Hourglass: Leveraging Transient Resources for Time-Constrained Graph Processing in the Cloud

Pedro Joaquim[†], Manuel Bravo[‡*], Luís Rodrigues[†], Miguel Matos[†]

[†]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
[‡]IMDEA Software Institute, Madrid, Spain

Abstract

This paper addresses the key problems that emerge when one attempts to use transient resources to reduce the cost of running time-constrained jobs in the cloud. Previous works fail to address these problems and are either not able to offer significant savings or miss termination deadlines. First, the fact that transient resources can be evicted, requiring the job to be re-started (even if not from scratch) may lead provisioning policies to fall-back to expensive on-demand configurations more often than desirable, or even to miss deadlines. Second, when a job is restarted, the new configuration can be different from the previous, which might make eviction recovery costly, e.g., transferring the state of graph data between the old and new configurations. We present Hourglass, a system that addresses these issues by combining two novel techniques: a *slack-aware provisioning* strategy that selects configurations considering the remaining time before the job's termination deadline, and a *fast reload* mechanism to quickly recover from evictions. By switching to an on-demand configuration when (but only if) the target deadline is at risk of not being met, we are able to obtain significant cost savings while always meeting the deadlines. Our results show that, unlike previous work, Hourglass is able to significantly reduce the operating costs in the order of 60 − 70% while guaranteeing that deadlines are met.

CCS Concepts • **Information systems → Graph-based database models**; • **Computer systems organization → Cloud computing**;

Keywords graph processing, time-constrained, transient resources, cloud computing

ACM Reference Format:
Pedro Joaquim, Manuel Bravo, Luís Rodrigues, Miguel Matos. 2019. Hourglass: Leveraging Transient Resources for Time-Constrained Graph Processing in the Cloud. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3302424.3303964

1 Introduction

The analysis of data modeled as graphs is increasingly relevant in many domains [21, 33, 47] and has driven the emergence of many specialized graph processing frameworks [4, 12, 15–17, 25, 27, 30, 32, 39, 48]. A key feature of many graph processing tasks is that they do not perform a "one-shot" computation. Instead, the dynamic nature of the target graphs [7, 23] often requires a recurrent analysis to keep results up-to-date [33]. Therefore, the cost of operating a recurring graph processing infrastructure over long periods of time can quickly become prohibitive, due to the continued use of a large number of resources. For example, based on the performance numbers and experimental setup reported by G-miner (a state-of-the-art graph mining system [11]), to allocate resources to perform a recurrent community detection computation [40] on a billion edges graph would cost on Amazon EC2 [1] more than $93K$ dollars per year.

Transient resources—resources with transient availability offered at a discounted price—present an opportunity to reduce such operational costs. For instance, running the same system as above on Amazon spot-instances costs approximately $13K$ dollars per year, resulting in a 86% cost reduction.[1] Unfortunately, transient resources *can* be unexpectedly revoked. This makes the provisioning of recurring graph processing tasks - that need to be executed periodically over a snapshot - very challenging: in order to avoid violating bounds on information staleness, and to ensure that the system is able to keep pace with the desired analysis frequency, it is crucial to guarantee that the analysis on a given snapshot terminates before the next one starts being processed. Given that evictions are not rare and that (re)starting a computation on spot-instances incurs significant delays [28] even when the interval between consecutive executions is not too tight, meeting execution time constraints becomes very hard.

*Work partially done as student at Universidade de Lisboa and Université Catholique de Louvain.

EuroSys '19, March 25–28, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303964

[1] https://aws.amazon.com/ec2/spot/pricing/

This creates a dilemma: *simple solutions that ensure termination within an execution deadline fail to provide the desired cost savings while solutions that only attempt to reduce costs often miss the deadlines [19, 34, 38].*

In this paper, we present Hourglass, a resource provisioning engine for graph processing that breaks this dilemma by offering substantial costs savings while ensuring that tasks meet their execution deadlines. Hourglass integrates two novel mechanisms:

1. A *slack-aware provisioning* strategy, that exploits the use of transient resources to reduce operational costs. It exploits the temporal slack available between the next deadline and the minimum time necessary to process the current task, to prioritize safer or riskier provisioning strategies, such that termination deadlines are met.
2. A *fast reload* strategy that allows to quickly re-partition the graph when new deployment configurations are selected: it combines an offline *micro-partitioning* technique that reduces the graph to a smaller graph by partitioning it into many small partitions, with an online *micro-partition clustering* policy that can quickly cluster these small partitions into a configuration tailored for the current deployment. We show that this strategy offers results that approximate the quality of offline partitioners, while avoiding the costs of precomputing *a priori* the partitioning for every possible configuration that may be selected in runtime.

We have built a prototype of Hourglass that integrates with the Amazon Web Services (AWS) ecosystem. Our prototype uses a custom version of Apache Giraph [4] (§7). We have experimented with three graph benchmarks: Graph Coloring (GC), PageRank, and Single-Source Shortest Paths (SSSP). Our results show that, unlike previous work, Hourglass is able to significantly reduce the operating costs in the order of 60 − 70% while guaranteeing that deadlines are met.[2]

2 The Practical Effect of the Dilemma

This section motivates our work. First, we illustrate our dilemma: previous state-of-the-art provisioning approaches, that only attempt to reduce costs with no concern for timing constraints, do miss deadlines; and naive adaptations of these approaches to ensure timely termination come short of providing the desired cost savings. Second, we illustrate the benefits of combining different mechanisms in Hourglass in order to break the dilemma.

To illustrate our point, we show results obtained when running a GC job [31] over a Twitter dataset [22] on transient resources, using Giraph [4] as the underlying graph processing system. The job can use multiple deployment configurations, each composed of a different number and type of machines. If executed using the fastest configuration, the job requires 4 hours to execute. In other available configurations it can take up to 10 hours. These are the observed execution times without the occurrence of evictions which further delay the job completion. Suppose that the computation needs to be re-executed 4 times a day, i.e, every 6 hours. This allows for a *slack* of 2 hours where one can use cheaper but slower configurations and still tolerate evictions (assuming the best configuration is selected as the last-resort configuration).

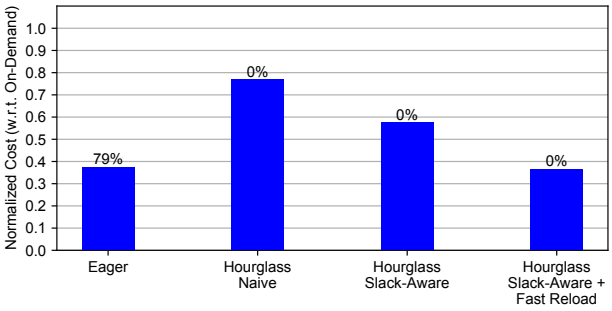

Figure 1. Cost and percentage of missed deadlines for an eager resource provisioning strategy and Hourglass.

We measured, for this scenario, the costs and probability of missing deadlines obtained using different provisioning approaches. To this end, and as in previous works [18, 19, 34, 35, 38], we use a public price trace of Amazon spot-instances [44] to simulate the execution of this job with real observed instance costs and eviction patterns (as it will be explained later, all simulations use values that have been extracted from real deployments). The sequence of deployments used in response to evictions or planned reconfigurations is selected by each of the provisioning strategies. We report the average cost obtained from executing the job over different randomly selected starting points in the trace (about 2000 simulations per strategy), while using the different provisioning strategies (the values are normalized with regard to the cost of running the job with on-demand resources). More details about the deployment configurations, price traces, fault-tolerance mechanisms, and partitioning strategies used, are given in §8.

Figure 1 shows the costs (y-axis) and the percentage of missed deadlines (numbers on top of each bar) for different provisioning strategies. The two leftmost bars illustrate the practical effect of our dilemma. The first bar (*eager*) shows the cost reductions obtained by using an eager strategy, similar to that of SpotOn [38], to acquire transient resources. This strategy chooses the deployment that reduces a cost per unit of work metric and is able to achieve an average cost reduction of 63%. However, this strategy missed the deadline in 79% of the runs, even with a 2 hours slack. The Hourglass

[2]Provided that our assumptions regarding the performance model used hold. See §5 for further details.

Naive bar shows the results of a naive approach to meet deadlines while using transient resources. This strategy uses first SpotOn and then reverts to on-demand instances once the time to reach the deadline becomes too short to tolerate further evictions. This approach is able to meet all deadlines but the cost savings drop to only 23% as, most of the times, the system has to fall back to on-demand resources before any substantial progress has been achieved.

The remaining bars of Figure 1 represent the following strategies: "Hourglass Slack-Aware" shows the effect of first applying our *slack-aware provisioning* strategy; and "Hourglass Slack-Aware + Fast Reload" shows the effect of adding the *fast reload* strategy to the equation. The slack-aware strategy, by itself, is able to achieve 43% cost reductions while missing no deadlines. By adding the *fast reload* mechanism, Hourglass is able to meet the timely constraints of the application with a cost reduction of 63%, the same as the eager strategy but without missing any deadline.

3 Background and Related Work

In this section, we briefly cover related work and highlight the main limitations of previous solutions.

3.1 Leveraging Transient Resources

A substantial number of techniques have been recently proposed to allow the usage of transient resources for different types of jobs in the cloud, such as SpotOn [38], Flint [34], SpotCheck [35], Proteus [19], or Tributary [18]. Most of these systems have been designed for applications that are substantially different from graph processing, and use techniques to recover from evictions that are not efficient in our context. SpotCheck [35], for instance, proposes a derivative cloud platform on the spot market that relies on VM migration mechanisms. However, these are intended for single machine applications as they lack coordination and have a limited state size that can be efficiently handled [38]. Also, most of these systems use greedy cost selection policies [19, 34, 38] that select the deployment that is expected to reduce the cost per unit of work at the provisioning moment. As we show in §8, these strategies are not a good fit for time-constrained jobs as they do not take the application timeliness into consideration, leading to missed deadlines and sub-optimal provisioning choices.

Some approaches (e.g. [18, 34, 38]) rely on over-provisioning (i.e., they use more transient machines than necessary) and explore eviction correlation metrics to provide bounds on latency SLOs. Although they succeed in mitigating the effect of a single eviction, their increased resource usage limits the potential cost reductions in the cases where (a few) evictions may be tolerated. Also, these solutions may end up spreading machines across different markets (to reduce eviction correlation), that may be geographically distant, a strategy that has a negative impact on graph processing jobs, that require frequent communication and coordination among workers [27].

3.2 Graph Processing

The relevance of graph processing has spurred the development of frameworks that are specifically tailored to the characteristics of these jobs. Two main approaches have been followed in the literature. One relies on hardware/software architectures that allow to process large graphs in a single machine. The other relies on distributed deployments, based on multiple machines with standard configurations. A notable example of the former approach is Mosaic [26] that, using fast storage media and massively parallel co-processors, is able to process a trillion-edge graph in a single machine. This indicates that with a sufficient powerful machine one can match, or outperform, distributed solutions. However, not all users have access to specialized hardware and the size of graphs is growing fast (for instance, the number of Facebook users has been reported to grow from 1.39B in 2014 [13] to more than 2B in 2017), which makes distributed solutions relevant.

A key issue that must be addressed to make distributed solutions effective is the fact that graph processing applications are iterative and require exchange of information among workers at the end of each iteration. Pregel [27] was one of the first graph processing systems, followed by several other systems such as [4, 15, 16, 30, 39]. These introduce optimizations such as: (i) partitioning the state of large vertices across several partitions [30]; (ii) asynchronous execution models [15, 16]; (iii) models centered in edges to better distribute work among workers for scale-free graphs [15]; and (iv) subgraph abstractions to bypass message passing overhead in local partitions [39].

Particularly relevant for the design of Hourglass is the fact that graph partitioning and loading, a task that needs to be executed whenever a new deployment configuration is selected, can consume a non-negligible amount of time. Also, in most cases, intelligent partitioning techniques [15, 20] significantly reduce the computational time [30, 39]. Unfortunately, previous graph partitioning schemes, such as METIS [20], are computationally slow and have been designed to be run only once, before graph processing starts. However, when using transient resources, the graph may need to be (re-)partitioned often due to evictions. Hourglass includes a novel partitioning and loading strategy that addresses these concerns.

4 Hourglass

We now provide an overview of Hourglass, a resource provisioning engine for time-constrained graph processing jobs that execute in Infrastructure as a Service (IaaS) platforms. Hourglass has two primary concerns: (i) minimizing the operational cost of executing the target graph processing

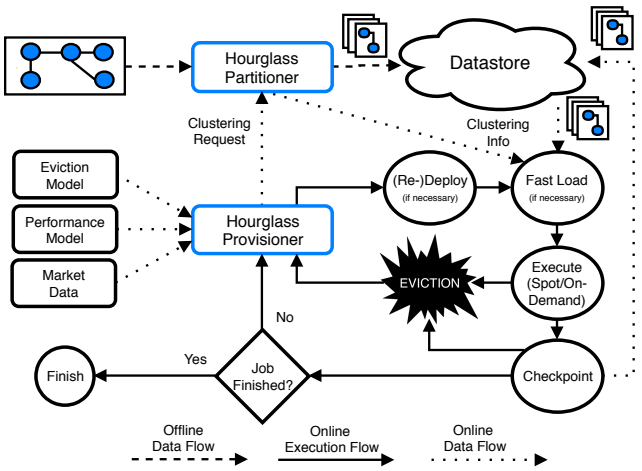

Figure 2. Hourglass overview.

job; and (ii) enforcing user specified temporal constraints. Figure 2 depicts a high level view of Hourglass and of its operational environment. The system is composed of two main components, a *Resource Provisioner* and a *Partitioner*. In a typical run, the execution flow is as follows:

1. The graph to be processed is first partitioned offline and stored in a persistent datastore [6, 36].

2. Hourglass selects the deployment configuration (transient or on-demand) to be used for the processing task using a slack-aware resource provisioning strategy (§5). This strategy aims at reducing the expected cost of running the job, while taking into consideration its temporal restrictions. The algorithm is fed with: (i) a model for estimating the performance of a given deployment configuration; (ii) a model for estimating the eviction probability of a given type of machine; and (iii) the current resource market prices.

3. When the requested machines are ready, the target graph processing system is deployed and, in turn, loads the graph data from the datastore. This step benefits from a novel micro-partitioning technique (§6) that significantly reduces the system's booting time, which is key to achieve greater cost reductions when leveraging transient resources, as we later show.

4. After loading the graph, the system executes until either an eviction occurs or it stops to checkpoint its progress. In the former case, the provisioner selects a new deployment to continue the job execution. In the latter case, if there is still work to perform, the provisioner decides if it is better to change or to keep the current deployment. Re-configuration may be required due to changes in the transient market prices or to comply with the application timely requirements. In any case, the system cycles between step 2 and 4 until the job completes.

5 Slack-Aware Provisioning Strategy

In this section, we describe the slack-aware provisioning strategy used by Hourglass. The goal of this strategy is to choose a deployment configuration that: (i) ensures that the job can complete before the deadline, even if evictions occur when transient resources are used; and (ii) minimizes the expected cost of running the job through completion.

Intuitively, our approach rests on two key observations: there is a deadline to execute each job, and there is a last-resort configuration that is able to complete the job before the deadline elapses (see Figure 3). The difference between the available time (given by the deadline) and the time required to execute the rest of the job using the last-resort configuration is denoted the slack-time. We exploit the slack-time available to make progress in the job using transient resources and thus reduce costs. If at any time during the execution of the job we run out of slack, we switch to the last-resort configuration to ensure that the job can be completed within the deadline. Note that depending on the slack available we might never need to switch and thus run the job to completion using exclusively transient resources.

We start by describing the system model, then we define precisely the optimization criteria that our strategy aims at achieving. Finally we describe a heuristic to find an approximate solution efficiently.

5.1 System Model

In this subsection, we introduce a number of concepts, terminology, and definitions that will be used to describe the Hourglass provisioning strategy. These are summarized in Table 1 and presented with greater detail in the following paragraphs.

On-demand and Transient Resources. We assume that the cloud providers offer two classes of resources, namely on-demand resources (D) and transient resources (T). On-demand resources are typically more expensive but are assumed to be reliable. Transient resources are cheaper but can be revoked without notice.

Deployment Configurations. Cloud providers have an extensive offer of different types of machines, which have distinct hardware characteristics and sizes. A deployment configuration c is a set of machines where the graph processing job can be executed. A configuration is characterized by the number and type of machines that are selected. We assume the set of possible configurations, denoted C, to be known a priori. We distinguish the set C^T of configurations that include transient resources from the set C^D of configurations that use only on-demand resources ($C = C^T \cup C^D$).

Performance Model. In order to select the right deployment configuration, Hourglass makes use of a *performance model* that can provide an estimate of the time t^c_{exec} required to run the job in a given configuration $c \in C$. How the

C	Set of all deployment configurations
t^c_{exec}	Estimated time configuration c takes to execute the target job
t_{boot}	Estimated time for the cloud provider to boot the requested machines
t^c_{load}	Estimated time configuration c takes to load the target graph dataset
t^c_{save}	Estimated time configuration c takes to checkpoint the current job state
t^c_{fixed}	Estimated time configuration c takes to boot, load the target graph and checkpoint the current job state
lrc	The last-resort configuration, which is the fastest on-demand configuration
$t_{deadline}$	The temporal deadline to finish a given job
$slack(t)$	Time between the deadline and the time necessary to complete the job in the fastest on-demand configuration started at time t
ω_c	The normalized capacity of configuration c
t^c_{ckpt}	Optimal checkpoint frequency of configuration c
$useful(c,t)$	Computation time left at time t for configuration c to execute
$expected\_progress(c,t)$	Expected job progress if configuration c is selected to execute at time t and no eviction happens
$EC(t,w)\|c$	Expected cost of running the remaining work w at time t when c is selected

Table 1. Notation.

model is constructed is orthogonal to the contributions of this paper. There is currently extensive research on methods for efficiently building performance models for the cloud (e.g. [8, 43]). In Hourglass we make the following approximation to make the problem of finding the best deployment tractable: we assume that the work progresses at uniform pace during the job execution. Say t^c_{exec} is the time required to execute the job on configuration c; our assumption is that by time $t^c_{exec}/2$ half of the work has been done. The performance model must also provide an estimate of the time it takes to load the graph to memory (denoted t^c_{load}) and of the time to checkpoint the progress (denoted t^c_{save}, see discussion on checkpointing below).

Last-Resort Configuration. We denote as the *Last-Resort Configuration* ($lrc \in C^D$) the on-demand configuration that can complete the graph processing job in the smallest amount of time, t^{lrc}_{exec}. Whenever possible, the provisioning strategy avoids using the last-resort configuration and, instead, attempts to use transient resources.

Deadline and Slack. We assume an existing *temporal deadline* to finish the graph processing job (denoted $t_{deadline}$). Naturally, we must have $t_{boot} + t^{lrc}_{load} + t^{lrc}_{exec} + t^{lrc}_{save} \leq t_{deadline}$, where t_{boot} is the time necessary to boot machines, measured as the elapsed time since a resource request is issued and the time the resource is ready to use. We denote the fixed costs of booting the lrc machines, loading the graph, and storing it to safe storage as $t^{lrc}_{fixed} = t_{boot} + t^{lrc}_{load} + t^{lrc}_{save}$.

The available slack is therefore the difference between the deadline and the time necessary to complete the job in the last-resort configuration, which is given by the sum of the fixed time and the variable time to finish the current job. More formally, at time t, with $w(t) \in [0,1]$ representing the percentage of the work left to be completed, and $horizon(t) = t_{deadline} - t$ the time left to the deadline, the available slack at that time t (depicted in Figure 3) is given by:

$$slack(t) = horizon(t) - t^{lrc}_{fixed} - w(t) * t^{lrc}_{exec}$$

Normalized Capacity. To assess the performance of a given configuration c, we often compare its performance with the performance of the last-resort configuration. We denote ω_c the normalized capacity of configuration c, i.e. $\omega_c = t^{lrc}_{exec}/t^c_{exec}$.

Eviction Model. Hourglass also needs to estimate how likely are transient resources to be revoked. This is provided by an *eviction model*. Although most cloud providers do not disclose information regarding the probability of eviction of a given resource, an eviction model can be constructed based on empirical evidence by studying the behavior of different resources, in different time periods and availability zones. A number of eviction models have been constructed in this way [18, 19, 28, 34, 38, 45]. Without loss of generality, we assume that the eviction model provides a cumulative distribution function (CDF) of the probability of being revoked before reaching a certain uptime. This offers a good tradeoff between complexity [18] and tractability [19, 34, 38].

Cost. The cost per unit of time of a configuration c ($cost^c$) is simply the price charged by the service provider at the provisioning moment.

Checkpoint Interval. Given that transient resources can be evicted, we require a mechanism to avoid losing all progress done upon an eviction. Most graph processing systems [4, 11, 15, 16, 25, 27, 30, 39] rely on checkpointing to achieve fault tolerance and, therefore, we leverage this mechanism. The

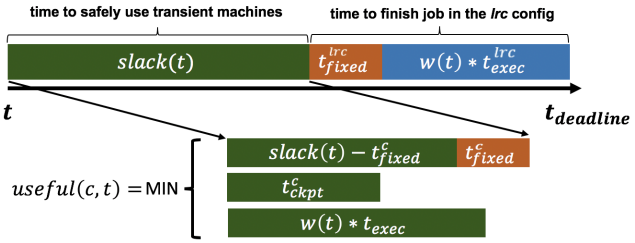

Figure 3. Available slack at time t and respective useful compute interval available for a configuration c.

optimal checkpointing interval, denoted t^c_{ckpt}, is dependent on the configuration c selected, because different configurations make progress at different pace and there is a minimum amount of progress that is worth being checkpointed. Like previous work [34], we rely on the result presented in [14] to select the checkpointing interval: it computes the optimal checkpointing frequency using the time necessary to checkpoint (t^c_{save}) and the mean time to fail (MTTF). For configuration c, the optimal checkpoint frequency is:

$$t^c_{ckpt} = \sqrt{2 * t^c_{save} * MTTF^c}$$

Useful Interval. We define $useful(c,t)$ as the time left (counting from instant t) before the computation on c is stopped either because: the job finishes, the slack is over, or a checkpoint of the application state needs to be performed. Please note that when a new configuration is selected to replace an existing one, the maximum execution time available becomes $slack(t) - t^c_{fixed}$ instead of just $slack(t) - t^c_{save}$. To avoid cluttering the notation, we do not further distinguish these two cases (the implementation accurately considers both cases). The useful interval for deployment c at time t (also depicted in Figure 3) is therefore given by:

$$useful(c,t) = \text{MIN}(w(t) * t^c_{exec},\ slack(t) - t^c_{fixed},\ t^c_{ckpt})$$

Progress. When the job runs uninterrupted for a period in a given configuration, it makes some progress. We define $expected\_progress(c,t)$ as the work that will be performed during the next useful interval if no evictions occur, i.e.:

$$expected\_progress(c,t) = \frac{\omega_c * useful(c,t)}{t^{lrc}_{exec}}$$

Thus, if at a given time t^i the work that remains to be performed is $w(t^i)$, at time $t^{i+1} = t^i + useful(c,t^i) + t^c_{save}$ we expect $w(t^{i+1})$ to be:

$$w(t^{i+1}) = \begin{cases} w(t^i) - expected\_progress(c,t^i), & \text{if no evictions} \\ w(t^i), & \text{otherwise} \end{cases}$$

5.2 Provisioning Criteria

With the definitions above, we can now define precisely the optimization criteria that our provision strategy should satisfy. When the strategy is invoked, it is fed with the current time (t) and the amount of work that remains to be done (w). The deployment strategy attempts to make progress using transient resources if this strategy is viable; otherwise, it terminates the job using on-demand resources. The choice of a configuration c to be used depends on the *Expected Cost* of executing the remainder of the job starting from that configuration, denoted $EC(t,w)|c$. Therefore, the expected cost of finishing the current job at time t, denoted $EC(t,w)$, is the expected cost of c_{best} that satisfies:

$$EC(t,w)|c_{best} \leq EC(t,w)|c_i : \forall c_i \in C$$

The expected cost of running work w at time t when c is selected to execute next is provided by:

$$EC(t,w)|c = \begin{cases} 0, & \text{if } w = 0 \\ \infty, & \text{if fails deadline} \\ cost^c \cdot (w \cdot t^c_{exec} + t^c_{save}), & \text{if } c \in C^D \\ cost^T(c,t,w), & \text{if } c \in C^T \end{cases}$$

The first and second cases correspond to the stop conditions, i.e., when the work has finished and when the current deployment is no longer able to be selected without compromising the job deadlines, respectively. By construction, the last-resort configuration never falls in the second branch and we always have at least one deployment able to be selected. The third and fourth cases are used when the selected configurations are either on-demand or transient and do not match any of the above cases. For configurations with transient resources, the expected cost is computed recursively. Let $Pe_c(t)$ be the probability of any of the transient resources of configuration c being evicted at time t, $Fe_c(t)$ the respective CDF function and $t^c_{int} = useful(c,t) + t^c_{save}$ the total time configuration c will be executing until it is able to checkpoint progress, the expected cost of c is:

$$cost^T(c,t,w) = Fe_c(t + t^c_{int}) * cost^T_{fail}(c,t,w) + (1 - Fe_c(t + t^c_{int})) * cost^T_{success}(c,t,w)$$

In this formula, the cost is calculated as the sum between the follow-up cost in case of failing before checkpointing ($cost^T_{fail}$) and in case of succeeding to do so ($cost^T_{success}$), weighted by the probability of each event occurring. In both cases, we compute the follow-up cost recursively with the respective time and work left to be finished. In particular, the follow-up cost in case of failure is calculated as:

$$cost^T_{fail}(c,t,w) = \int_t^{t+t^c_{int}} \frac{Pe_c(x)}{Fe_c(t+t^c_{int}) - Fe_c(t)} \cdot (cost^c \cdot x + EC(t+x,w))\, dx$$

This cost is given by the integral of the follow-up costs in case of failing in each possible moment before completing the useful computation time, therefore no progress is achieved, weighted by its probability. In case of succeeding to checkpoint, the follow-up cost is also computed recursively as:

$$cost^T_{success}(c,t,w) = cost^c \cdot t^c_{int} + EC(t+t^c_{int}, w - expected\_progress(c,t))$$

5.3 Approximating *EC* Efficiently

As explained before, Hourglass takes provisioning decisions in key moments of the job execution. One relevant aspect is the time necessary to reach a decision. The system should be able to reach good decisions at key moments of execution (for instance, after evictions occur) without requiring the entire system to stop for a significant amount of time. One could consider computing the provisioning decisions ahead of time in an offline manner. However, the provisioning policy takes into consideration parameters that are only known to the system at run time (e.g. current market prices, exact moment in which evictions occur, etc.).

Since *EC* needs to be computed online, it needs to be computed fast. However, computing *EC* by solving the integral of $cost^T_{fail}$, which requires an approximation by a finite sum, is prohibitively slow for any reasonable discretization of time (we illustrate this in §8). Therefore, we now propose the following simplifications to approximate $cost^T$ efficiently. We assume that, if a configuration is not evicted, it remains in use for consecutive checkpointing intervals, assumption supported by empirical evidence that deployment reconfigurations not due to evictions are rare. Under this assumption, the expected follow-up cost in case of success ($cost^T_{success}$) is only computed recursively for the current configuration rather than for all possible configurations. Additionally, the follow-up cost in case of failure ($cost^T_{fail}$) is only computed at the *MTTF* of a given configuration rather than for all instants between the beginning and end of the computation.

6 Fast Reload

Distributed graph processing systems are characterized by an initial phase in which the target graph is first *partitioned* and then *loaded* by the workers (machines) before starting the execution phase. Typically, the partitioning is based on the number of machines that will be available during execution phase. This step may be costly when chasing high-quality partitionings but its cost is amortized as it is only executed once. Nevertheless, when using transient resources, one may have to re-deploy the system frequently due to evictions. Given that the number of machines across deployments may vary, the system may be forced to re-partition the graph such that a new partitioning—more suitable for a newly selected configuration—is computed.

Next, we describe why the current graph partitioning and loading approaches are not suitable for Hourglass. Then, we describe a novel strategy, which combines a micro-partitioning (aka over-sharding) and a clustering technique, that circumvents the limitations of previous work.

6.1 Limitations of Current Strategies

Previous work can be grouped into three main classes: strategies that assign vertices to partitions based on hashing [27]; strategies that create graph partitions offline using some optimization criteria that improves the execution phase online (for example, METIS [20]); and strategies that perform the graph partitioning during loading time using stream-based approaches [37, 41], which usually also follow some intelligent partitioning strategy. All these approaches offer a trade-off between partitioning time and partition quality, which makes them well suited for graph processing jobs with different characteristics. Nevertheless, all strategies have limitations whose impact is more significant when applied to our dynamic context and that we must address in order to efficiently leverage the available slack time.

Partitioning Phase. All approaches partition the graph based on the number of worker machines that are available in the execution phase. In our dynamic setting, this translates into possibly having to re-partition the target graph every time the system changes to a new configuration. When using hashing, this has no impact because there is no partitioning phase. The partitioning step is implicitly hidden behind the hash function that translates vertices into partitions (usually the modulus of an integer vertex id and the target number of partitions). On the other hand, when using offline and stream-based partitioners, which seek partitionings of better quality,[3] this is specially problematic because they require increased partitioning times. If we increase the partitioning time (by executing it multiple times for different deployment configurations) we are not only increasing the overall job cost but also further reducing the situations in which it is worth using one of these approaches instead of hashing, therefore also reducing potential cost benefits.

Loading Phase. When hash and offline partitioners are used, the loading of the graph can be parallelized by assigning different graph chunks to all machines available [4]. These machines read and parse the data into in-memory entities (vertices or edges) that are then forwarded over the network

[3] Good-quality partitionings are those that minimize the number of edges between vertices belonging to different partitions, as this implies communication among machines during the execution phase, which affects performance negatively.

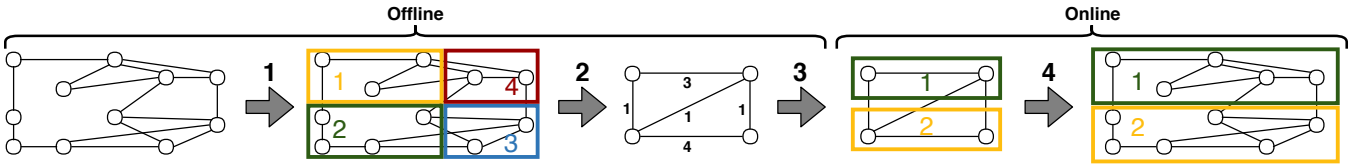

Figure 4. Micro partition generation process and subsequent partitioning steps. First the original graph is partitioned into the desired number of micro partitions (step 1) and reduced (step 2) offline. The reduced graph is then partitioned online (step 3) instead of the original graph. Finally, the vertices of the reduced graph are then mapped to the micro-partitions of the original graph (step 4).

to the workers to which they were assigned in the partitioning phase. As we will show in §8, the exchange of data over the network during the loading phase has a significant impact in the overall time needed to perform such a task. In the best case scenario, workers only load graph entities that are assigned to them, therefore avoiding the exchange of data over the network. One could achieve this by storing the graph in the same order in which it was partitioned such that workers could be assigned chunks that only contain their assigned data. However, if we happen to change the deployment configuration, therefore requiring a new partitioning phase, the data of a single partition would be again scattered, falling back to the first scenario. Regarding stream-based partitioners [37, 41], these often use a centralized partitioning logic that requires a single machine to load the entire graph dataset, thus preventing potential performance benefits that can be achieved by allowing a parallel loading of the graph.

6.2 Hourglass Partitioner

The Hourglass partitioner avoids the limitations of previous approaches thanks to a novel hierarchical approach that combines both offline and online steps, as illustrated in Figure 4. Offline, Hourglass partitions the graph based on a micro-partitioning strategy. Online, the system merges and creates partitions that fit the target deployment configuration needs. This way, the system avoids the need to perform a new partitioning phase every time a new configuration is selected. Still, when clustering micro-partitions, one must seek to preserve the partition quality obtained if running the baseline partitioner from scratch for the target deployment configuration. This is achieved by our clustering policy, which is used in the online phase to select which micro-partitions to merge. The following paragraphs give additional details regarding the offline and online steps.

Micro Partitioner. In the offline phase, the graph is partitioned in micro-partitions by using any state-of-the-art partitioning scheme. In the current prototype, we allow the user to decide between using METIS [20], FENNEL's [41] one-pass streaming algorithm, and the standard hash partitioning algorithm [27]. The choice of the partitioning algorithm depends on the size and properties of the target graph (results for different approaches are discussed in §8). Nevertheless, our approach does not preclude the use of other partitioners, tailored for a specific graph. A key design issue is to decide how many micro-partitions to create. Hourglass selects the number of micro-partitions as the *least common multiple* of the number of worker machines used by configurations in C. This ensures that, whatever configuration is selected in run-time, it is possible to generate equally-sized clusters, which favours a balanced work among workers.

Clustering. One key observation is that micro-partitions can also be represented as a graph: there is an edge between two micro-partitions if there is one or more edges among the vertices of these micro-partitions; the weight of the edge between the two micro-partitions is the number of edges that cross the border between them. Logically, the offline step reduces the original graph to another graph, that is orders of magnitude smaller. By creating a micro-partitions graph, as depicted in Figure 4, the problem of clustering can be modelled recursively as a partitioning problem. We rely on METIS to solve the recursive problem formulation and find an optimal clustering of micro-partitions to macro-partitions that are suitable for a given configuration We chose METIS because it provides good results and, for the problem sizes we used in our evaluation, we were able to obtain a solution in few milliseconds. Nevertheless, Hourglass is independent of METIS and any other partioner could be used. As we will show in §8, our clustering policy is able to adapt the micro-partitions (created in a single partitioning step) to multiple partitioning solutions with very reduced quality degradation. Additionally, due to this micro-partitioning strategy, we are also able to attain *parallel recovery* [42, 46], a powerful recovery property that makes eviction recovery more efficient by allowing Hourglass to paralellize the loading step after an eviction, without worker communication. This is possible because graph data remains partitioned in the same way across different configurations, only micro partitions are assigned to different workers that can load them in parallel and independently.

7 Implementation

Our prototype is integrated with the AWS ecosystem but it could be easily extended to support other providers. It

consists of 3k lines of Java code. We have selected the widely used Apache Giraph [4] (version 1.2.0) as our graph processing engine. We opted to use Giraph for several reasons. First, it is one of the few solutions publicly available. Second, it has been widely used and its performance has been reported in several works. Finally, and most importantly, its source code is also available; by using Giraph, we can illustrate our contributions without risking to have the results polluted by our own (potentially flawed) implementations of other systems. Giraph runs on top of Hadoop [5], therefore we decided to integrate Hourglass with the Amazon Elastic MapReduce service [2] (release 5.11.1) that provides an AWS optimized version of Hadoop. We have also modified the checkpointing mechanism of Giraph such that it reads/stores checkpoints from/to Amazon S3 [3], a highly available cloud storage service, rather than to the underlying distributed filesystem. This allows a recovery from a full system failure that may occur due to evictions. Regarding spot-instance requests, which require a bidding price representing the maximum price one is willing to pay for a given machine (not the actual paid price), we simply bid the on-demand price. Although previous work has given special focus on how to place the ideal bid [18, 19], recent changes in the spot-instances pricing model[4] turned the submitted bid irrelevant to determine when spot-instances are terminated. This effect has been confirmed empirically by recent work [28].

Regarding micro-partitions, we have modified Giraph to perform the graph reduction step (§6) while the graph is being first loaded, with a few seconds overhead. From the three supported partitioning approaches, only METIS requires an offline step to generate micro-partitions. When FENNEL is used, micro-partitions are generated in the first loading phase. For hashing, micro-partitions are also created during the first loading phase by assigning chunks of the graph dataset (e.g. file blocks from the target storage system) to workers that load them and become owners of all the vertices in the assigned file blocks. In this setting the system benefits from a loading step that is parallelized across workers and requires no communication to exchange out-of-place data. Independently of the partitioning approach used, after micro-partitions are created, any subsequent load of the graph is done in parallel and without coordination among workers. Also, there is no need for an additional online partitioning phase, even if a different deployment configuration is selected.

8 Evaluation

In this section, we first evaluate Hourglass as a whole, comparing it with state-of-the-art approaches in §8.2, and then study the impact of each individual contribution in the subsequent sections.

[4] https://aws.amazon.com/blogs/compute/new-amazon-ec2-spot-pricing/

	# Vertices	# Edges	Network Type
Human-Gene [29]	22 283	12 323 680	Biological
Hollywood [29]	1 069 126	56 306 653	Collaboration
Orkut [24]	3 072 626	117 185 083	Social
Wiki [29]	5 115 915	104 591 689	Web Pages
Twitter [22]	52 579 678	1 614 106 187	Social
RMAT-N [10]	2^N	2^{N+4}	Synthetic

Table 2. Graph datasets.

8.1 Experimental Setup

Similarly to previous works, we resort to both real deployments on Amazon EC2 and simulations [18, 19, 34, 35, 38, 45]. Simulations are used for long running experiments, that aim at assessing cost savings over long periods of time and that would be impractical and costly to perform in a real deployment. All the simulations are fed with parameters, such as application execution times, extracted from real deployments. Simulations use a price trace from the Amazon's *us-east-1* region on November 2016 [44]. Thus, when running the simulation, both the changes in prices and the evictions that result from these changes follow exactly what would happen if Hourglass was executed in that period of time. Using this methodology, the experiments can be reproduced and allow us to compare the different strategies under exactly the same conditions. Additionally, a trace from October 2016 was used to derive the historical statistics, such as eviction probabilities and average market prices per instance type, necessary in the provisioning decisions of the tested approaches.

Graph Datasets. Table 2 describes the graph datasets used in the experiments. We use a mix of real world graphs from different domains and synthetic datasets with different scale parameters.

Graph Applications. We use the following applications in our evaluation: *SSSP*, which calculates the distance from all vertices to a single source vertex; PageRank [9], which estimates the relevance of each vertex based on the existing connections (edges); and *GC*, that assigns the least possible number of colors to vertices such that no two adjacent vertices have the same color, following the approach proposed in [31]. These applications are representative of typical graph processing jobs that can go from a few minutes (SSSP) to hours (GC).

Instance Types. We consider a total of 9 deployment configurations where each uses a single type of machine from the "memory optimized" family and either all resources are transient or none. The use of homogeneous deployments is justified by the fact that Giraph adopts a synchronous execution model, therefore (i) the execution inevitably halts if some resources are evicted, even when some resources are still available, and (ii) no gains are achieved by having machines that make computational progress at different rates.

More precisely, we use r4.2xlarge, r4.4xlarge, and r4.8xlarge instances in deployments that have a total of 16, 8 and 4 worker machines.

Simulation Parameters. To build an accurate simulation, we first identified the key parameters that affect the system which include, among others, the time to boot machines, the time to read/store data from/to the external storage, and the execution time of each of the tested graph processing applications across the different graph datasets and in each of the possible deployments considered. We then performed the actual deployment of each configuration and measured each parameter. These measurements are used to calibrate the simulator.

8.2 Overall Performance and Comparison with the State-of-the-Art

We start by assessing the number of missed deadlines and the cost reductions achieved by Hourglass. For this purpose, we executed different graph processing jobs on Amazon using Hourglass, and compared it to two state-of-the-art provisioners, namely Proteus [19], and SpotOn [38], which we integrate in our prototype.

Proteus, greedily selects the deployment expected to reduce the cost per unit of work produced at each moment. SpotOn, which is also targeting batching systems, chooses between: (i) using a single transient deployment and checkpoint the state periodically to safe storage; or (ii) replicating the system across multiple transient deployments and avoid the checkpoint overhead. In both cases, SpotOn uses a greedy policy similar to the one used by Proteus. None of these provisioners attempts to satisfy deadlines. Therefore, we have derived, implemented, and experimented a straightforward extension to these systems that is able to enforce deadlines. The extension, denoted deadline protection (DP) mechanism, simply switches the deployment to use on-demand resources once the slack available to tolerate further evictions is exhausted. The variants augmented with the DP mechanism are identified as Proteus+DP and SpotOn+DP.

Overview of Results. Figure 5 shows the results for 30 different scenarios. We use the three graph applications SSSP, PageRank (30 iterations), and GC, which exhibit different execution times: 3 minutes, 20 minutes and 4 hours in the last-resort configuration, respectively. Note that these execution times for the last-resort configuration include not only the computation time (with checkpointing disabled) but the time to: bootstrap the system (start Hadoop and then Giraph), download the datasets from the external storage (S3 in our case), load the datasets into Giraph and finally write the computation output back to the external storage after finishing the computation. For each job we use 10 different deadlines, which vary the slack available to finish the job from 10% to 100% of the execution time. The bars in the figure depict the total average cost of running the jobs (including both offline and online phases) normalized by the cost of running them in the on-demand baseline configuration. The baseline cost is calculated as a single execution of the target job in the last-resort configuration from the moment the graph dataset starts being retrieved from the external storage until the computation output is safely stored. In the on-demand baseline configuration, the checkpointing feature was disabled (no overhead storing checkpoints into the external storage). The numbers above the bars denote the percentage of runs that missed deadlines. Therefore, lower bars represent lower costs and smaller numbers above the bars represent less missed deadlines (always 0 for Hourglass). The costs measured for each strategy are the average over 2000 simulations of the target job (with the starting moment selected at random). All experiments have been executed using the Twitter dataset, our largest dataset.

We first analyze the performance of the different jobs for a given slack and later discuss the overall trend when the slack increases. A detailed explanation of why Hourglass achieves these results is postponed for §8.3, where we zoom into some of these results and, with the help of additional micro-benchmarks, provide insights on the contribution of the different Hourglass components to the overall performance in different conditions.

Different Jobs with the Same Slack. We start by discussing the performance of SSSP (the shortest job) with a 20% slack. Interestingly, Hourglass offers the best cost savings. Proteus and SpotOn also achieve large savings (but not as expressive as Hourglass) but miss many deadlines. With the deadline protection mechanism enabled, both systems (Proteus+DP and SpotOn+DP) avoid missing deadlines, but they no longer achieve cost savings. We now look at the performance of GC (the longest job) with the same slack. In terms of cost savings, Hourglass is slightly worse than Proteus and SpotOn. However, Hourglass avoids missing deadlines while the latter miss the deadlines in most of the runs. In this job, the simple DP mechanism is able to provide some savings, but these savings are relatively modest when compared with Hourglass. The results for PageRank (the medium duration job) show a similar trend.

Changing the Slack. We now discuss the trends when the available slack changes. An interesting observation is that, in almost all scenarios, Hourglass, which never misses deadlines, is able to approximate, or even outperform, Proteus and SpotOn, which miss the deadlines in a large fraction of runs. Not surprisingly, the larger the slack (i.e., when more time is available to recover from evictions), the easiest it is to achieve savings without missing deadlines. Therefore, for large slacks the simple DP mechanism also offers reasonable results. However, Hourglass is able to achieve savings when the slack is smaller, in scenarios where DP fails to be effective. What is somehow surprising is that, even with very large slacks, Proteus and SpotOn still miss the deadlines frequently,

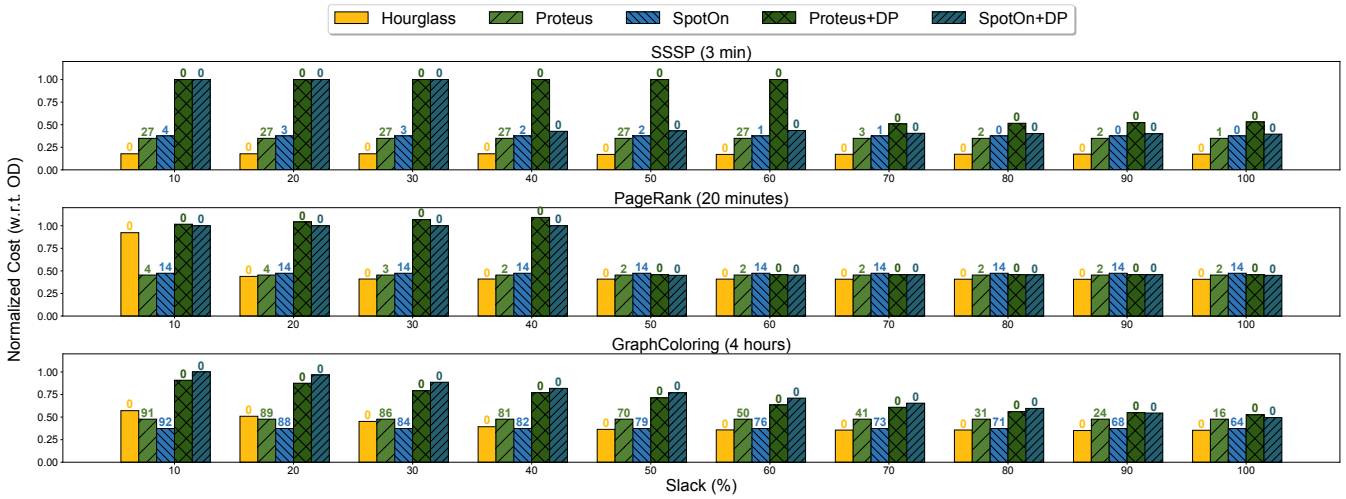

Figure 5. Cost reductions achieved (bars) and percentage of deadlines missed (number above the bars) by different resource provisioners with different temporal slacks. Lower bars, with smaller numbers above them is better.

in particular in longer jobs (such as GC). This highlights the importance of the slack-aware provisioning strategy. It is also interesting to notice that Hourglass is able to avoid missing deadlines without a large cost penalty. This happens because Hourglass is still able to make significant progress using transient resources and, when it switches to the last-resort configuration, only a small fraction of the job remains to be executed. Thus, Hourglass only pays an additional cost when it really needs to. Furthermore, as shown later, our fast reload mechanism is very effective, which also helps Hourglass to reduce costs.

Relaxing the Deadlines. Although we have always run Hourglass such that it would never miss a deadline, the reader might wonder what would be its behaviour if we would configure Hourglass in a way that deadlines could be missed (let us call this version *relaxed*-Hourglass). One way of implementing *relaxed*-Hourglass is to setup the standard Hourglass with a target larger than the real deadline. In this case, *relaxed*-Hourglass would operate using an inaccurate (larger) slack and, in face of evictions, would switch to the on-demand configuration too late, risking missing deadlines. Thus, the performance of *relaxed*-Hourglass is the same of standard Hourglass with larger slacks. As discussed before, under this setting the relevance of the slack-aware provisioning strategy becomes lower and most gains come from the use of micro-partitioning.

8.3 Micro Benchmarks

We now study in more detail the contribution of the different Hourglass mechanisms to the results of the previous section. To this end, we resort to both a set of micro-benchmarks and experiments that zoom in on some of the results presented in Figure 5.

8.3.1 Effects of Fast Reload in Short Jobs

Because SSSP and PageRank are short jobs (complete in 3 and 20 minutes), it is rare to observe evictions while they run. In this case, provisioning decisions do not play a significant role. Also, because jobs are short, it does not payoff to use expensive data partitioning schemes. Therefore, the best results with these systems are achieved with hashing: there is no partitioning phase, this is implicitly hidden behind the hash function, and the graph can be loaded in parallel. Still, Hourglass outperforms Proteus and SpotOn. In these short jobs, the key contributor to these results is the Hourglass's micro-partitioning and fast reload mechanisms, which bring advantages even when hashing is used. In fact, as discussed in §7, the hashing micro-partitioner is able to support the effective parallelization of the loading step without requiring communication among workers. This optimization significantly reduces the time necessary to load the graph, with a visible impact on the final cost of the entire run.

To give further insights on the performance of our fast-loading mechanism, we measure the loading times that different loading strategies exhibit when using different datasets and configurations. Figure 6 shows the results. In particular, we measure the loading time for the fast reload approach (Micro Loader), for the Hash Loader, and for the Stream Loader. With the Micro Loader, workers are assigned micro partitions that are loaded in parallel. With the Hash Loader, workers load in parallel chunks of the target graph dataset but are required to exchange data subsequently. With the Stream Loader, the entire graph dataset is loaded first by the master node and then assigned to the workers. For the latter, we report only the time necessary for the master to load the dataset and ignore the partitioning time. In the figure, the size of the dataset doubles from left to right. For each

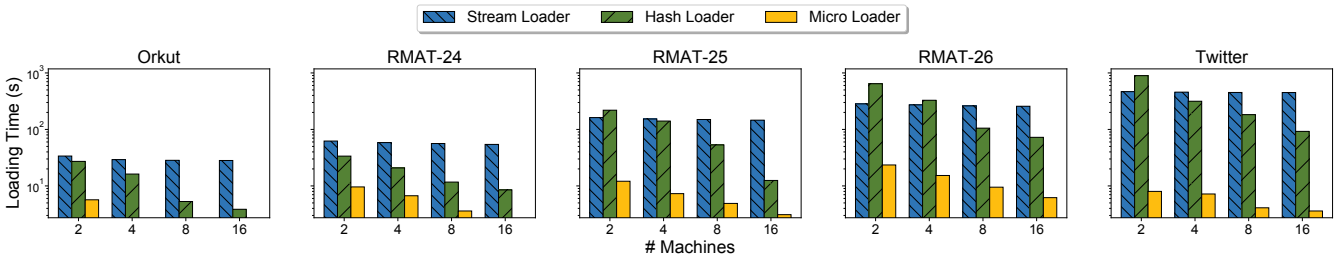

Figure 6. Loading times for different strategies. Y-axis in log scale. The size of the dataset doubles from left to right.

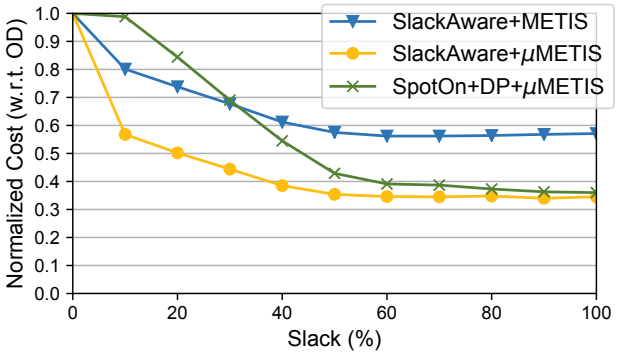

Figure 7. Zoom in the cost reductions achieved in the GC application.

dataset, we varied the number of machines performing the loading phase between 2 and 16 machines. Also, for each dataset, we used a single machine type that was the smallest machine type able to load the target dataset with the desired number of machines. In detail, for the Orkut and RMAT-24 dataset we used r4.xlarge machines, for the RMAT-25 and RMAT-26 datasets we used r4.2xlarge machines and r4.4xlarge machines for the Twitter dataset.

From the obtained results, we can draw several conclusions. First, the loading time of the Stream Loader increases with the dataset size, as expected. The Hash Loader performs worse for a small number of machines, due to network bottlenecks. The Micro Loader is on average, across all configurations, 11.34×, 10.5×, 22.2×, 19.7× and 79.6× faster than the Single Loader approach in the Orkut, RMAT-24, RMAT-25, RMAT-26 and Twitter datasets, respectively, and 4.98×, 3.39×, 15.5×, 21.1× and 64.8× faster than the Hash Loader in the same datasets.

8.3.2 Hourglass Gains in Long Jobs

Long duration jobs (e.g. GC) are more susceptible to suffer evictions or price fluctuations that may require reconfigurations. In these scenarios, both the fast reload mechanism, its clustering policy, and the slack-aware provisioning approach are important to outperform the cost reductions achieved by previous provisioners.

To get an insight on the relative importance of each mechanism, we now look in detail at the performance of the GC application. For this application, METIS is the partitioning approach that yields the best results with Hourglass. The cost savings achieved by Hourglass result from both the micro-partitioning and the slack-aware provisioning strategy. To illustrate this, we show in Figure 7 the performance of Hourglass with and without micro-partitioning and also the effect of micro-partitioning on SpotOn (that does not use the slack-aware scheduler). Note that SlackAware+μMETIS combines both components and corresponds to the values previously presented in Figure 5.

The results show that the micro-partitioner is always useful, independently of the slack used. The difference between the cost reductions achieved by the slack-aware approach with and without micro-partitioning is on average of 23%. This is mainly due to the smaller offline costs incurred by the micro-partitioning approach, which requires running METIS only once, while the default strategy requires running METIS for the multiple configurations. As expected, our slack-aware approach performs significantly better than SpotOn+DP for smaller slack sizes, where the bad provisioning decisions have a greater impact. Namely, in this experiment, with just 10% slack time, the slack-aware approach already achieves significant cost reductions while SpotOn+DP offers no improvements.

8.3.3 Effects of Fast Reload in Partition Quality

We now analyze how the micro-partition clustering policy impacts partition quality. Partition quality is given by the percentage of edges cut between all the partitions created [37, 41], which can be used to estimate the percentage of communication that will be done across different machines.

Figure 8 shows how our micro-partitioning clustering technique impacts the partition quality when clustering 64 micro-partitions into macro-partitions (from 2 to 32 partitions) of five different types of graph datasets. In the first experiment (top row of plots) we use METIS to generate the initial micro-partitions and, in the second, we use FENNEL

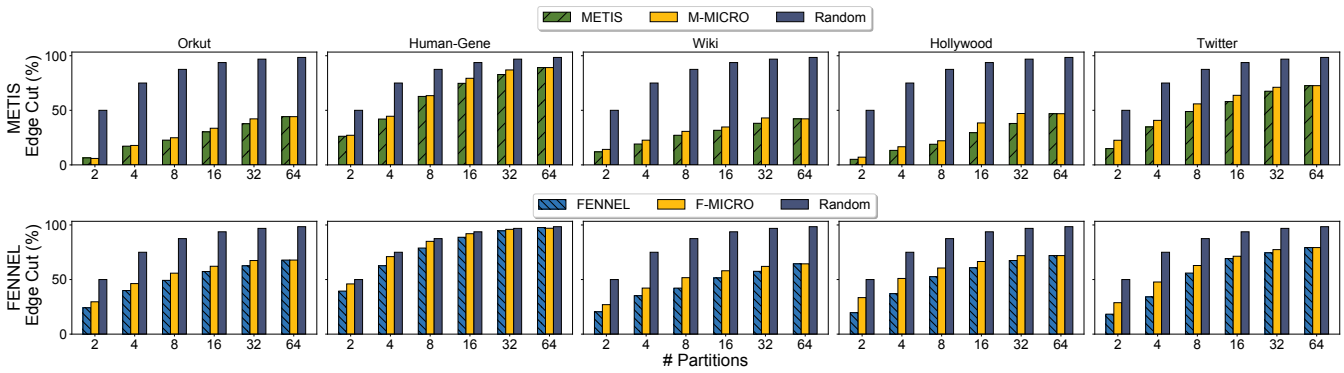

Figure 8. Partition quality analysis.

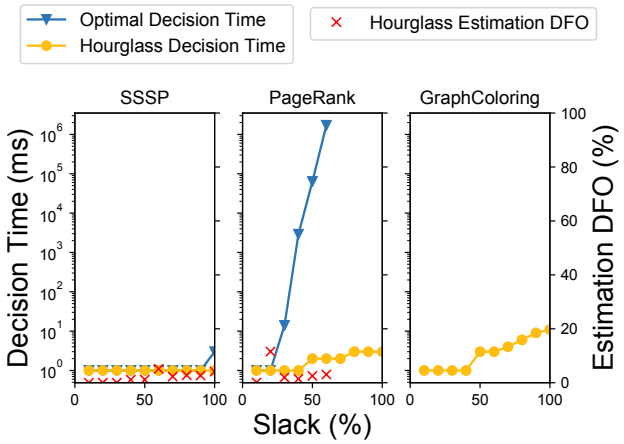

Figure 9. Comparison of decision time and estimation error of the Hourglass provisioning strategy against the optimal cost estimator. Optimal did not finish for slacks larger than 60% for the PageRank application and for any slack size in the GC application.

(configured with the same parameters as in [41]). We set both partitioners to balance the total number of edges assigned to the different partitions, similar to [30]. In each experiment, we compare the quality of the partitions obtained with our micro-clustering (M-Micro or F-Micro) against the quality obtained when using the base algorithm (METIS or FENNEL). For comparison, we also plot the partition quality obtained when assigning vertices randomly to partitions (Random), given by $1 - 1/n$. With METIS, the partition quality obtained by our clustering policy has, on average across all the number of partitions from 2 to 32, only more 1.68%, 2.16%, 2.94%, 4.43% and 4.97% of edges cut than the baseline approach in the Orkut, Human-Gene, Wiki, Hollywood and Twitter datasets, respectively. With F-Micro, the number of edges cut in our micro-partitioning approach increases, on average, 4.65%, 4.18%, 5.7%, 7.66% and 6.02% for the same datasets as before. The practical effects of the quality degradation were measured empirically for the Twitter dataset across all the applications used in §8.2. The observed impact in the application running time by using our micro-partitioning approach rather than the baseline partitioner were mostly negligible (at most 2 – 3% degradation) and often inside the natural fluctuations in the execution time observed when running the same experiment in the same exact conditions.

8.3.4 Accuracy of the EC approximation

The Hourglass provisioner requires the computation of the expected cost (EC) of a configuration. This is computed using the approximation introduced in §5.3, given that the precise formulation presented in §5.2 (which requires computing the integral) does not scale for realistic problem sizes. Here we show that solving the integral is, in fact, not scalable but that Hourglass approximates EC efficiently and without significant loss of accuracy.

Figure 9 shows the time necessary to reach a provisioning decision (depicted in log-scale on the y-axis) as a function of the slack. Note that the higher the slack, the largest the search space. The figure shows the time it takes to derive the provisioning strategy when EC is computed using the optimal formulation, and the approximation detailed in §5.3. Both lines computed EC using a discretization of time based on one second intervals, as this is the minimum time unit in which we observed price changes in the price traces,[44]. The figure also plots the approximation distance from optimum (DFO), which is calculated as $\frac{|\hat{cost}_H - \hat{cost}_O|}{\hat{cost}_O} * 100$.

The obtained results clearly show that computing EC by solving the integral is only able to provide a provisioning decision in reasonable time for the SSSP (all slack times tested) and the PageRank application (until a slack time of 40%). For slacks larger than 60% in the PageRank application, and for the GC application (all slacks), we are unable to get a single provisioning decision under one hour when computing the integral. However, when using the EC approximation

described in §5.3, it is possible to derive a provisioning decision in just a few milliseconds, for all the applications tested and all slack percentages. Interestingly, these improvements are achieved without a significant loss of accuracy: for the settings where we were able to reach an optimal solution, the average error of all the approximations is only 3%, on average.

9 Model Evolution and Future Work

We now discuss some of the extensions that we would like to address in the future.

First, some providers issue a warning before resources are evicted. Such warning event can be incorporated in our model, by considering that some progress is still possible even when there are evictions. Namely, in the current model, when computing $cost^T_{fail}(c, t, w)$ in a recursive manner, it is assumed that no progress is made if the eviction happens before the checkpoint (see §5.2). The model can be extended to account for the amount of work that has been performed before the eviction, in the case the warning is given early enough to allow a successful checkpoint [38]. This would improve Hourglass results by decreasing the work lost when evictions happen and by reducing the number of situations where Hourglass is forced to fall back to the last-resort configuration prematurely due to the conservative assumption that all work may be lost.

More interestingly, we would like to experiment with our slack-aware provisioning strategy on a wider range of applications. Note that our formulation of the provisioning strategy is quite general, and it has a number of parameters (such as the loading time, etc) that can be adjusted for different applications. The key assumption is that the application has some recovery mechanism, which prevents all work from being lost when an eviction occurs. In this paper, we have assumed that this mechanism is based on checkpointing and that all work after the last checkpoint is lost upon an eviction. In the previous paragraph, we have already discussed how to adjust the model to have a more accurate estimate of the work that can be lost in case of eviction for other cases (such as an eviction warning). A similar approach can be used to encompass other recovery mechanisms. Also, in the current paper, we assume that the provisioner is only called when there is a checkpoint or an eviction. However, nothing prevents the provisioning process from executing in other stages of the execution, when a relevant event occurs. For example, applications that execute in multiple phases, and where each phase requires a different amount of resources or impacts the computational progress differently in case of failure, may require the provisioner to be executed at the end of each phase.

10 Conclusions

We have presented Hourglass, the first resource provisioning engine that, by relying on transient resources, achieves significant savings in the deployment of time-constrained graph processing jobs in the cloud. Hourglass combines two novel techniques. First, a novel slack-aware provisioning strategy that permits it to achieve cost reductions on par with the state-of-the-art. Unlike previous work, our strategy considers the temporal slack available to select safer or riskier provisioning strategies such that termination deadlines are met. Second, a fast reload mechanism that optimizes the graph loading phase of existing graph processing systems. This mechanism is key to reduce the time necessary to recover from evictions. Our evaluation has shown that, when the two techniques are combined together, Hourglass is able to reduce the operating costs in the order of 60%–70% while guaranteeing that deadlines are met.

Acknowledgments

We thank our shepherd Ittay Eyal, and the anonymous reviewers for their comments and suggestions. This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) and Feder through the projects with references PTDC/EEI-SCR/ 1741/ 2014 (Abyss), PTDC/EEI-COM/29271/2017 (Cosmos), LISBOA-01-0145-FEDER- 031456 (Angainor) and UID/ CEC/ 50021/ 2019.

References

[1] Amazon EC2. https://aws.amazon.com/ec2/. Last Accessed: September 2018.
[2] Amazon EMR. https://aws.amazon.com/emr/. Last Accessed: September 2018.
[3] Amazon S3. https://aws.amazon.com/s3/. Last Accessed: September 2018.
[4] Apache Giraph. http://giraph.apache.org. Last Accessed: September 2018.
[5] Apache Hadoop. http://hadoop.apache.org. Last Accessed: September 2018.
[6] Apache Hive. https://hive.apache.org. Last Accessed: September 2018.
[7] Leman Akoglu, Hanghang Tong, and Danai Koutra. 2015. Graph Based Anomaly Detection and Description: A Survey. *Data Min. Knowl. Discov.* (May 2015).
[8] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association.
[9] S Brin and L Page. 1998. The anatomy of a large scale hypertextual Web search engine. *Computer Networks and ISDN Systems* 30 (1998).
[10] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining.
[11] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: An Efficient Task-oriented Graph Mining System. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. Porto, Portugal.
[12] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the Pulse of a Fast-changing and Connected

World. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, Bern, Switzerland.

[13] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-scale. *Proc. VLDB Endow.* (2015).

[14] J. T. Daly. 2006. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems* (2006).

[15] Je Gonzalez, Y Low, and H Gu. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (2012).

[16] Minyang Han and Khuzaima Daudjee. 2015. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *Proc. VLDB Endow.* (2015).

[17] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: A Graph Engine for Temporal Graph Analysis. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, Amsterdam, The Netherlands.

[18] Aaron Harlap, Andrew Chung, Alexey Tumanov, Gregory R. Ganger, and Phillip B. Gibbons. 2018. Tributary: spot-dancing for elastic services with latency SLOs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA.

[19] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. 2017. Proteus: Agile ML Elasticity Through Tiered Reliability in Dynamic Resource Markets. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, Belgrade, Serbia.

[20] George Karypis and Vipin Kumar. [n. d.]. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* ([n. d.]).

[21] Jon M. Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S. Tomkins. 1999. The Web As a Graph: Measurements, Models, and Methods. In *Proceedings of the 5th Annual International Conference on Computing and Combinatorics (COCOON'99)*.

[22] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. ACM, Raleigh, North Carolina, USA.

[23] Kathy Lee, Diana Palsetia, Ramanathan Narayanan, Md. Mostofa Ali Patwary, Ankit Agrawal, and Alok Choudhary. 2011. Twitter Trending Topic Classification. In *Proceedings of the 2011 IEEE 11th International Conference on Data Mining Workshops (ICDMW '11)*. Washington, DC, USA.

[24] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[25] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A New Framework for Parallel Machine Learning. *CoRR* (2010).

[26] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. Belgrade, Serbia, 527–543.

[27] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*.

[28] T. Pham, S. Ristov, and T. Fahringer. 2018. Performance and Behavior Characterization of Amazon EC2 Spot Instances. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*.

[29] Ryan Rossi and Nesreen Ahmed. 2013. Network Repository. http://networkrepository.com

[30] Semih Salihoglu and Jennifer Widom. 2013. GPS: A Graph Processing System. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management (SSDBM)*. ACM, Baltimore, Maryland, USA.

[31] Semih Salihoglu and Jennifer Widom. 2014. Optimizing Graph Algorithms on Pregel-like Systems. *Proc. VLDB Endow.* (2014).

[32] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. 2016. GraphIn: An Online High Performance Incremental Graph Processing Framework. In *Proceedings of the 22Nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833*. Springer-Verlag New York, Inc., New York, NY, USA.

[33] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: Real-time Content Recommendations at Twitter. *Proc. VLDB Endow.* (2016).

[34] Prateek Sharma, Tian Guo, Xin He, David Irwin, and Prashant Shenoy. 2016. Flint: Batch-interactive Data-intensive Processing on Transient Servers. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, London, United Kingdom.

[35] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. 2015. SpotCheck: Designing a Derivative IaaS Cloud on the Spot Market. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, Bordeaux, France.

[36] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. [n. d.]. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10)*. IEEE Computer Society, Washington, DC, USA.

[37] Isabelle Stanton and Gabriel Kliot. 2012. Streaming Graph Partitioning for Large Distributed Graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '12)*. ACM, Beijing, China.

[38] Supreeth Subramanya, Tian Guo, Prateek Sharma, David Irwin, and Prashant Shenoy. [n. d.]. SpotOn: A Batch Computing Service for the Spot Market. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM, Kohala Coast, Hawaii.

[39] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "Think Like a Vertex" to "Think Like a Graph". *Proc. VLDB Endow.* (2013).

[40] Etsuji Tomita and Tomokazu Seki. 2003. An Efficient Branch-and-bound Algorithm for Finding a Maximum Clique. In *Proceedings of the 4th International Conference on Discrete Mathematics and Theoretical Computer Science (DMTCS'03)*. Springer-Verlag, Berlin, Heidelberg.

[41] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining (WSDM '14)*. ACM, New York, NY, USA.

[42] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Shanghai, China, 374–389.

[43] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA.

[44] Rich Wolski, John Brevik, Ryan Chard, and Kyle Chard. [n. d.]. Probabilistic Guarantees of Execution Duration for Amazon Spot Instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, Denver, Colorado.

[45] Youngseok Yang, Geon-Woo Kim, Won Wook Song, Yunseong Lee, Andrew Chung, Zhengping Qian, Brian Cho, and Byung-Gon Chun. 2017. Pado: A Data Processing Engine for Harnessing Transient Resources in Datacenters. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, Belgrade, Serbia.

[46] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Farminton, Pennsylvania, 423–438.

[47] Song Zhang, Hu Chen, Ke Liu, and Zhirong Sun. 2009. Inferring protein function by domain context similarities in protein-protein interaction networks. *BMC bioinformatics* (2009).

[48] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. [n. d.]. Gemini: A Computation-centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association.