Caerus: NIMBLE Task Scheduling for Serverless Analytics

Hong Zhang, *UC Berkeley;* Yupeng Tang and Anurag Khandelwal, *Yale University;* Jingrong Chen, *Duke University;* Ion Stoica, *UC Berkeley*

https://www.usenix.org/conference/nsdi21/presentation/zhang-hong

This paper is included in the Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation.

April 12–14, 2021

978-1-939133-21-2

Open access to the Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

Caerus: NIMBLE Task Scheduling for Serverless Analytics

Hong Zhang UC Berkeley
Yupeng Tang Yale University
Anurag Khandelwal Yale University
Jingrong Chen Duke University
Ion Stoica UC Berkeley

Abstract

Serverless platforms facilitate transparent resource elasticity and fine-grained billing, making them an attractive choice for data analytics. We find that while server-centric analytics frameworks typically optimize for job completion time (JCT), resource utilization and isolation via inter-job scheduling policies, serverless analytics requires optimizing for *JCT* and *cost of execution* instead, introducing a new scheduling problem. We present Caerus, a task scheduler for serverless analytics frameworks that employs a fine-grained NIMBLE scheduling algorithm to solve this problem. NIMBLE efficiently pipelines task executions within a job, minimizing execution cost while being Pareto-optimal between cost and JCT for arbitrary analytics jobs. To this end, NIMBLE models a wide range of execution parameters — pipelineable and non-piplineable data dependencies, data generation, consumption and processing rates, etc. — to determine the ideal task launch times. Our evaluation results show that in practice, Caerus is able to achieve *both* optimal cost and JCT for queries across a wide range of analytics workloads.

1 Introduction

Serverless platforms [1–3] fulfill the promise of transparent resource elasticity in the cloud [4–6]. Under the Function as a Service (FaaS) serverless model, users decompose their applications into short-lived stateless *functions* that read and write data from an external storage service. The sub-second startup latencies and virtually unlimited parallelism in FaaS platforms permit fine-grained compute elasticity, while sub-second billing granularities afford cost-efficiency.

These benefits have driven many recent efforts to port *data analytics* applications to serverless platforms [7–18]. Analytics jobs typically comprise multiple stages of execution organized as directed acyclic graphs (DAGs) based on their data dependencies, with each stage comprising several parallel tasks. While traditional server-centric deployments use clusters provisioned with a fixed pool of storage and compute resources to execute these jobs, serverless deployments implement tasks as serverless functions [7–13] that exchange state via external storage [14, 15]. Since analytics workloads typically have widely varying resource needs over time, both across and during job lifetimes [12, 14], server-centric deployments can frequently suffer from resource under- or over-provisioning [12, 14, 19, 20], leading to resource wastage or performance degradation, respectively. In contrast, serverless compute [1–3] and storage [15, 21–24] platforms facilitate *fine-grained scaling* of resources to match application needs, making them an attractive choice for data analytics [7–18].

We find that the shift from server-centric to serverless analytics results in a shift in goals for schedulers in analytics frameworks. Since the FaaS platforms manage allocation of compute resources *across* jobs, schedulers need no longer be concerned with the conventional goals of maximizing cluster resource-utilization and enforcing fairness across jobs via *inter-job* scheduling policies [25–28]. Instead, under the FaaS billing model, schedulers must now consider the *cost* of each job's execution, which is proportional to the aggregated runtimes across its component tasks. This highlights the need for *inter-task* scheduling policies for serverless analytics jobs to minimize both execution cost and job completion time (JCT).

Unfortunately, task-level scheduling policies employed by server-centric analytics today expose a hard-tradeoff between cost and JCT in serverless platforms. Figure 1 shows a simple map-reduce job where reduce tasks consume and aggregate data generated by map tasks. Traditional analytics frameworks [29–32] typically employ one of the two following extremes: (1) a *lazy* approach that launches a reduce task only when all the map tasks have finished (Figure 1 (a)), and (2) an *eager* approach that launches a reduce task as soon as any map task produces data for it to consume (Figure 1 (b)).

Intuitively, the lazy approach is *cost-efficient*: since reduce tasks waste no time waiting for upstream map tasks to generate data, individual task durations (which governs cost in serverless settings) is always minimized. However, its JCT can be far from optimal since there is no *pipelining* of map and reduce task executions. The eager approach, on the other hand, is *JCT-efficient* since it maximally pipelines the execution of map and reduce tasks. However, its can introduce a much higher cost: reduce tasks can waste a lot of time waiting for upstream map tasks to generate data, which increases reduce task durations and, consequently, execution cost. We discuss this example further in §2, but note for now that this trade-off between execution cost and JCT is even more extreme for multi-stage jobs seen in production workloads [27, 28].

Note that in an ideal solution (Figure 1 (c)), a task would be launched *late enough* to minimize task durations (and therefore, execution cost), but *early enough* to minimize JCT. In this work, we propose a NIMBLE scheduling algorithm that builds on this intuition: at its core, NIMBLE scheduling combines the cost-efficiency of lazy and JCT-efficiency of eager approaches and breaks the tradeoff between them (Figure 1 (d)), by scheduling tasks to run *at just the right time*.

Designing such an optimal scheduling strategy, however, is non-trivial. First, a precise description of the pipelinablity across different job stages is crucial to determine the optimal schedule — task-level DAGs typically used for representing

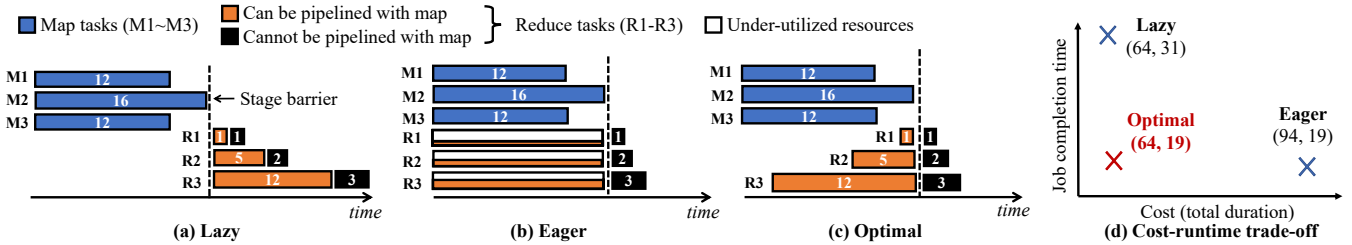


Figure 1: (a, b) Lazy and eager approaches expose a hard trade-off between JCT and cost; numbers within bars correspond to task runtimes. (c, d) Fine-grained scheduling in serverless infrastructures provide opportunities to break this tradeoff with optimal scheduling strategies. The JCT is simply the finish time of the last reduce task, while its cost is calculated as the aggregated durations of all its component tasks.

job executions in existing job schedulers are insufficient. Even for the simple map-reduce example in Figure 1, while parts of reduce task execution can be pipelined with map tasks (orange bars), some parts can only start after map stage finishes (black bars), *e.g.*, when map output must be aggregated at the reduce task before further processing. To this end, we develop a fine-grained *step dependency model* that captures data dependency and pipelinablity information at *sub-task* granularity (§3).

Second, in contrast to the map-reduce example above, tasks in general analytics jobs can have significantly more complex pipeline dependencies. Specifically, a task can consume data from multiple upstream tasks, and tasks across the job's execution DAG may have cascading dependencies. Coupled with time-varying data generation and consumption rates, this makes identifying task launch times for JCT- and cost-efficient job execution challenging. In fact, our analysis shows that even with perfect models for all of the above constraints, it is *impossible* for a task scheduling algorithm to always be able to optimize both execution cost and JCT for arbitrary analytics jobs. Fortunately, we show it *is* possible for a scheduling algorithm to be cost optimal, while being *Pareto-optimal* between execution cost and JCT. We realize this in NIMBLE, a scheduling algorithm that carefully models data produce and consume rates across stages, computes launch times for tasks across them based on both inter- and intra-task data dependencies, and schedules tasks greedily across dependent stages (§4).

Finally, we incorporate the NIMBLE algorithm into Caerus, a new fine-grained task-level scheduler for serverless analytics frameworks (§5). Caerus translates the theory developed for NIMBLE to practice, by extracting step dependencies from user queries via a step annotation API, and estimating NIMBLE algorithm inputs using a combination of job execution histories and information profiled at runtime. Caerus easily integrates with existing serverless analytics frameworks [11, 12, 14] — we implement Caerus in a prototype serverless SQL engine built atop Locus [14], and evaluate its performance on AWS Lambda for a wide range of analytics workloads including TeraSort, TPC-DS and Big-Data Benchmark (§6). Our results show that in practice, Caerus optimizes *both* cost and JCT, outperforming the lazy approach by 1.08–2.2× in JCT, and eager approach by 1.21–1.57× in cost across these workloads.

In summary, we make three main contributions:

- Formulation of a new task-level scheduling problem for serverless analytics to minimize execution cost and JCT. We show that schedulers used in server-centric frameworks expose a hard tradeoff between cost and JCT (§2).
- Design of a new NIMBLE scheduling algorithm, that launches each task in a job at just the right time to optimize *both* cost and JCT. NIMBLE employs a new *step model* to capture sub-task level pipelinablity and data dependencies, and guarantees cost optimality while being *Pareto-optimal* between cost and JCT for any analytics job (§4).
- Design, implementation and evaluation of Caerus, a fine-grained task-level scheduler for serverless analytics frameworks that enables NIMBLE scheduling in practice (§5, §6).

2 Motivation

In this section, we provide a brief background on server-centric and serverless analytics (§2.1). We then describe how serverless analytics introduces a new task scheduling problem (§2.2) and new opportunities to address it (§2.3).

2.1 Background

Server-centric Analytics. Traditional server-centric deployments for data analytics [30, 31, 33–35] operate atop a fixed pool of compute and storage resources, *e.g.*, clusters of provisioned servers or pools of provisioned virtual machines (VMs)[1]. Consequently, such deployments employ a cluster-wide job scheduler to efficiently share the fixed resource-pool among multiple jobs with three key goals: minimizing job runtime, maximizing resource utilization and ensuring resource isolation (or fairness) across jobs. Given the resource demands of each job, the scheduler achieves all or a subset of goals via *inter-job* (i.e., job-granularity) scheduling policies [25–28].

Within a job, the execution is broken down into a DAG of stages, each comprising multiple parallel tasks (see Figure 1 for an example). A task scheduler launches tasks across the compute resources allocated to the job. Tasks in a stage read their initial input from and write their final output to persistent storage (*e.g.*, HDFS [37]), while data exchange between consecutive stages occurs over the network (*e.g.*, shuffle, broadcast, etc.). Existing frameworks typically apply one of two popular approaches to decide *when* to launch tasks: (i) *lazy* (*e.g.*, Spark [30]), which launches a task only when *all*

[1] One can add/remove VMs to scale VM clouds, but at coarse time granularities, *e.g.*, resizing an AWS EMR cluster takes $\sim 6-45$ minutes [36].

tasks in upstream stages have completed, and (ii) *eager* (*e.g.*, MapReduce Online [29]), which launches a task as soon as *any* output from its upstream stages is ready.

Serverless Analytics. In serverless platforms, users no longer provision or manage resources: this is the cloud provider's responsibility. Users simply pay for resources they use. Serverless compute platforms [1–3] allocate and charge for compute resources at function invocation granularity: invoking more functions permits scaling up at a higher cost, and vice versa.

Existing approaches to serverless analytics deploy tasks within a stage as serverless function invocations. Since cloud providers disallow direct communications between serverless functions [7, 11, 14], data is exchanged between functions via external storage [8, 14, 15]. A job is charged for both function execution and external storage, with the former typically dominating the cost[2]. With *sub-second granularity billing* for serverless functions, the job execution cost is proportional to the cumulative runtimes across all tasks of the job.

2.2 Serverless Scheduling: A New Problem

Since the cloud provider is responsible for resource management in serverless platforms, user goals in serverless analytics are different from server-centric deployments. In particular, while minimizing JCT is still a primary goal, metrics like resource utilization and isolation are now the onus of the cloud provider. Instead, the user must now optimize the *cost* of each job's execution, which is proportional to the cumulative task runtime as outlined in §2.1. This shift in goals exposes a new task-level scheduling problem for serverless analytics:

> **Problem Statement:** *Given the execution plan for an analytics job comprising tasks with arbitrary dependencies, can we find a task-level schedule that optimizes for both job execution cost and JCT on a serverless platform?*

Limitations of existing approaches. As we saw in §1, the existing server-centric lazy and eager task scheduling approaches, when applied for serverless analytics, expose a hard tradeoff between cost and JCT. Recall the job execution example in Figure 1, which comprises a map and a reduce stage, each with three tasks — each bar represents the execution of one task over time (numbers in bars show task runtimes).

The lazy approach (Figure 1 (a)) is *cost-optimal* in the serverless model, with a cost of 64 units[3] — starting reduce tasks any later would not affect their runtime (and therefore, cost), while starting them sooner can cause them to stall for more data to be generated by upstream map tasks, increasing cost. However, the lazy approach also leads to high JCT (31 units), since it does not pipeline the execution of map and reduce tasks at all. Similarly, eager scheduling (Figure 1 (b)) is *JCT-optimal* (19 units) since the first part of reduce execution

[2]Cost of AWS Lambda execution is ∼$0.20/hour [38], while Amazon S3 storage is ∼$0.02/GB/month [39], with no data transfer cost between them.

[3]The cost is computed as the cumulative sum of the runtimes of the tasks in the job, and assuming unit cost per unit runtime.

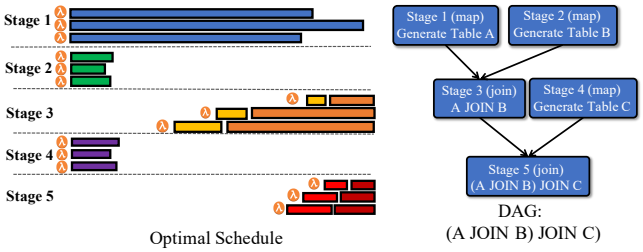


Figure 2: **Optimal schedule (left) and execution DAG (right) for a multi-stage job.** See §2.3 for details.

(orange bar) can be completely pipelined with the map stage. However, its cost is significantly higher (94 units), since reduce tasks often wait for data to be generated by upstream map tasks, increasing their runtime, and therefore, cost.

This tradeoff can be much more severe for multi-stage jobs. Production traces from Microsoft [27, 28] show that jobs in their workloads have 13 and 121 stages at 50th and 95th percentiles, making it likely for them to have far more opportunities for pipelining tasks across stages. Ignoring these opportunities (*e.g.*, following the lazy approach) would lead to JCTs that are significantly longer than optimal. On the other hand, jobs can also have heavy skew in task runtimes [40–42] — 10% of tasks take more than 10× the median task duration in Microsoft's workloads [40]. Starting tasks across all stages early to maximize pipelining (*e.g.*, following the eager approach) would force most downstream tasks to stall due to slower upstream tasks, significantly increasing execution cost.

2.3 Opportunities & Challenges

New opportunities in serverless scheduling. Serverless frameworks provide new opportunities to break the hard tradeoff between cost and JCT exposed by lazy and eager solutions — on-demand invocation of functions at fine-grained timescales permits the design of *fine-grained task-level schedulers*. Figure 1 (c) shows the optimal schedule for the job in Figure 1 — with fine-grained scheduling, it is possible to achieve such a schedule by launching each task *at just the right time*, minimizing both cost (64 units) *and* JCT (19 units) (Figure 1 (d)).

Moreover, these gains are likely to be even more significant in production workloads comprising multi-stage jobs with complex stage dependencies. For example, Figure 2 shows a multi-stage SQL job which performs join across three tables (A, B and C) using shuffle hash join (SHJ) algorithm [43]. The figure shows the job's execution plan as a DAG of stages on the right, and the corresponding optimal task schedule on the left. The optimal schedule can efficiently pipeline all the five stages in this multi-stage join example, resulting in much higher gains in JCT and cost than for simple two stages map-reduce jobs.

Challenges. Figure 2 also indicates that calculating the optimal launch time for each task is non-trivial due to a number of reasons. First, a task may include multiple parts, where each part may or may not be pipelineable with some part of one of its upstream stages. In Figure 2 (left), Stage 3 is composed of two parts. The first part, which reads Table B from Stage 2

and uses it to build a hash table, can be pipelined with Stage 2 execution. The second part, which reads Table A from Stage 1 and performs online join with the hash table constructed in the first part, can pipelined with Stage 1 execution. Second, the runtime of one task depends on the processing rate of all tasks in its previous stage, and these dependencies cascade to upstream stages. In Figure 2, the execution of Stage 5 depends on Stage 3 and Stage 4, and Stage 4 is further determined by Stage 1 and Stage 2. As such, the first challenge lies in identifying parts of the execution that can be pipelined, and the dependencies between such pipelinable components — we address this in §3. The second challenge lies in using this information to determine ideal launch times for tasks in jobs with complex DAGs, which we address in §4.

Why serverless? Intuitively, the fine-grained task-level scheduling shown in Figures 1 (c) and 2 can also be extended to server-centric settings to optimize average JCT. Moreover, the reduction in per-job resource usage (i.e., cost in serverless settings) enabled by this approach may improve resource utilization via bin-packing more jobs onto the same number of servers. However, while cost improvements in serverless analytics are obvious, achieving improvements in resource utilizations with theoretical guarantees in server-centric deployments is not straightforward, since it is unclear how the resources saved by delaying task launch times can be utilized by other jobs. Specifically, the optimal in Figures 1 (c) and 2 is likely to create staggered task launch times across stages to optimize *each individual* job, and they may not be optimal for bin-packing *across* jobs. Thus, while the clear decoupling from inter-job resource allocation ensures cost and JCT-optimality, extending it to server-centric settings for optimal JCT and resource utilization requires a careful co-design of inter- and intra-job scheduling. We leave this study to future work.

3 Step Dependency Model

As discussed above, a key challenge in identifying ideal task launch times for a job is modeling pipelineable and non-pipelineable dependencies across tasks. In this section, we discuss how we model such dependencies and the flow of data across them, using a new *step dependency model*. We employ this model to design our NIMBLE scheduling algorithm in §4.

Stage dependencies in traditional analytics. As outlined in §2.1, job execution in traditional analytics frameworks [27, 28, 30, 34] is represented as a DAG, where nodes are execution *stages* (comprised of multiple parallel tasks) and edges denote *data dependencies* between them. Figure 3 (a, left) shows the DAG for the map-reduce example from Figure 1, while Figure 3 (b, left) shows a SQL query that performs shuffle hash join (SHJ) on tables generated by two map stages.

Unfortunately, the stage model is not fine-grained enough to capture the information required to determine the ideal launch times for tasks in serverless analytics jobs. To see why,

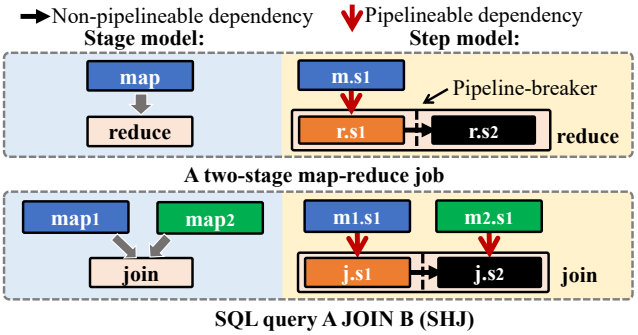


Figure 3: **Stage *vs.* step dependency model** for (a) map-reduce job, and (b) SQL query that joins two tables A and B after applying a map function on each. In the step model, red arrows show dependencies across steps that can be pipelined, while black arrows show dependencies that prevent pipelining. See §3 for details.

consider the map-reduce example from Figure 3 (a, right)[4], where the reduce stage (and therefore, all tasks in the stage) has two distinct parts, shown as orange and black boxes. While the first part (`r.s1`), where reduce tasks read map data, can be pipelined with map execution (`m.s1`), the second part (`r.s2`), where the reduce tasks aggregate and output data, cannot — since final aggregation can only occur after all map data has been read. Clearly, stage dependencies, shown in Figure 3 (a, left), cannot capture such fine-grained information regarding pipelineable and non-pipelineable components of task, nor capture the data dependencies between them. This information is crucial in determining the optimal start time for reduce tasks — early enough to maximally overlap `r.s1` with `m.s1`, but not too early, since pipelining `r.s2` with `m.s1` is impossible.

Modeling pipeline dependencies using *steps*. To precisely model how stages can be pipelined, we refine the stage model into a fine-grained step model to precisely describe how job execution can be pipelined across stages. In our model, the stages are decomposed into one or more *steps*, which are separated by *pipeline breakers* within the stage — operators that produce their first output only after all input have been processed. Pipeline breakers create barriers in execution, demarcating stretches of execution that cannot be pipelined with each other. As such, steps within a stage must be executed sequentially, since pipeline breakers prevent subsequent steps from starting before its upstream step finishes. Across stages, however, steps with data dependencies between them can be pipelined. As a concrete example, consider the step model for the map-reduce job in Figure 3 (a, right) — `m.s1` corresponds to the single step in map stage, while `r.s1` and `r.s2` correspond to two steps in the reduce stage, with a pipeline breaker separating them. The step `r.s1` which consumes data can be pipelined with the upstream step `m.s1` in the map stage that generates the data. We refer to such cross-stage pipelineable step pairs (*e.g.*, (`m.s1`, `r.s1`)) as *parent-child* step pairs. Note that the while above description focuses on the decomposition of a stage into steps, each task within the stage shares the same

[4]This is the same example as the one depicted in Figure 1.

step-level decomposition — we will use the term step to refer to parts of a stage or its tasks interchangeably and clarify the distinction whenever needed.

Figure 3 (b) contrasts the step and stage DAGs for a simple join query. Each of the two map stages comprise a single step, while the hash join stage is divided into two steps. The step `j.s1` reads the left table (Table A generated by `m1.s1`) to create a hash table of unique entries, while step `j.s2` reads the right table (Table B generated by `m2.s1`), joins it with the hash table and writes the output. Each of the two steps can be pipelined with their parent steps (the two map stages), but these two steps have to be executed sequentially within the join stage, since the hash table must be created before the second join step can proceed (pipeline-breaker).

We discuss the details of how the step dependencies can be extracted from user code in §5, but note for now that this model is expressive enough to capture the pipeline dependencies across a wide range of evaluated analytics applications (§6).

Modeling flow of data across steps. We now describe parameters that are used to model the flow of data across steps in the step dependency model. While we discuss how these are estimated in §5, we note for now that these parameters are used as inputs to the NIMBLE algorithm. Consider a stage comprising n steps, `s1`-`sn`, some of which may have a parent step, while some may not. If step `si` receives data from a parent step, then (1) parent *produce rate* (r_p) is the aggregated data output rate across all tasks of its parent step (referred to as *produce rate* for brevity); and (2) *full consume rate* (r_c) is the rate at which data can be read and processed by step `si` when there is sufficient data for it to consume. If step `si` does not have a parent step, then its execution duration $d_{\mathtt{s}i}$ is independent of when the task is launched, allowing us to model $d_{\mathtt{s}i}$ as a constant.

Since the produce rate is determined by the aggregate data output rate across all upstream tasks, each with potentially different start and end times, we model r_p as an arbitrary function of time t. Note that the cumulative area under the $r_p(t)$ curve corresponds to the total input data for the step under consideration; we denote this as P. The full consume rate, on the other hand, is tied to how fast the step can read and process data, and we found it to be stable throughout a the step's execution in our evaluation (§6), allowing us to model r_c as a constant. Note that the parent step may not always produce data as fast as it can be consumed, i.e., the *actual consume rate* (r_{ac}) for the step may be lower than r_c.

4 NIMBLE Scheduling

Armed with the step dependency model, we are now ready to describe our NIMBLE scheduling algorithm. NIMBLE builds on the intuition outlined in §2.3, and combines the cost-optimality of lazy and JCT-optimality of eager approaches to schedule tasks in an analytics job to run at *just the right time*. We first describe NIMBLE scheduling for a simple two-stage map-reduce job (§4.1), and then extend it to general analytics jobs with arbitrary execution DAGs (§4.2).

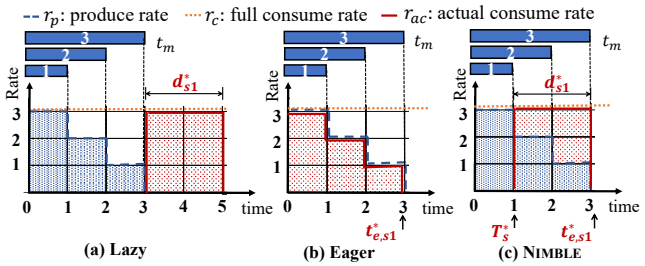


Figure 4: **Optimal launch time for a two-stage map-reduce job.** (a) The total volume of data to be consumed by the reduce step `r.s1` ($P = 6$) is the area under the produce rate ($r_p(t)$) curve. The lazy approach allows us to compute the optimal task runtime ($d^*_{\mathtt{s1}}$) as $P/r_c = 2$. (b) The optimal task finish time ($t^*_{e,\mathtt{s1}} = 3$) is obtained by emulating the eager approach, where the finish time is the maximum of P/r_c ($= 2$) and the map finish time t_m ($= 3$). (c) The optimal launch time ($T^*_s = 1$) is computed as the difference of the optimal finish time and optimal duration. See §4.1 for details.

4.1 NIMBLE for Two-stage Map-Reduce

Consider the step model for the simple two-stage map-reduce job in Figure 3 (a). Note that the JCT of the job is the same as the finish time of the last reducer, and the total cost of the job is proportional to the aggregated duration of all map and reduce tasks. As such, optimizing for the finish time and execution duration of individual tasks also ensures optimality for JCT.

Since map tasks do not have any upstream dependencies, their execution duration is independent of their launch times, and only depends on how fast they can read data from persistent storage and process it. Meanwhile, optimal finish time for map tasks can be achieved by launching them as early as possible (at $t = 0$). On the other hand, due to the parent-child step dependency between the map and reduce tasks (Figure 3 (top)), the *data consumption* in `r.s1` step of reduce tasks can be pipelined with the *data generation* in step `m.s1` of map tasks for minimizing reduce task finish times and execution durations. In particular, a reduce task should be launched early enough to ensure `r.s1` overlaps with `m.s1` as much as possible to minimize finish time, but late enough to ensure that it can always consume data at full rate throughout its execution without stalling, to optimize cost. Our NIMBLE scheduling approach can always find such a "perfect" launch time using the following three steps (Figure 4):

Step 1: Calculate optimal task duration D^*. Since step `r.s2` can only start after `r.s1` finishes, the optimal duration D^* of a reduce task is $d^*_{\mathtt{s1}} + d^*_{\mathtt{s2}}$, where $d^*_{\mathtt{s1}}$ and $d^*_{\mathtt{s2}}$ are the optimal durations of steps `r.s1` and `r.s2` respectively. Note that since `r.s2` does not have a parent step, its duration is independent of when the reduce task is scheduled. As such, the optimal duration D^* depends only on step `r.s1`.

Recall from §2.2 that the lazy approach always ensures optimal duration for reduce tasks — since the entire input is available before the reducer starts, `r.s1` can always consume the input data at consume rate r_c without ever stalling. As such, $d^*_{\mathtt{s1}}$ is simply P/r_c, where P is the total amount of input data for `r.s1`. In Figure 4 (a), $P = 6$ is the area under the curve

$r_p(t)$, which gives $d^*_{\texttt{s1}} = 3$.

Step 2: Calculate optimal finish time T^*_e. The optimal finish time T^*_e for a reduce task is simply $t^*_{e,\texttt{s1}} + d^*_{\texttt{s2}}$ where $t^*_{e,\texttt{s1}}$ is the optimal finish time of `r.s1`. Again, since duration $d^*_{\texttt{s2}}$ is independent of when the task is launched, T^*_e depends only on when step `r.s1` finishes.

We leverage the eager strategy of starting the reduce task at $t = 0$ to compute the optimal step finish time $t^*_{e,\texttt{s1}}$ — intuitively, starting the task any sooner cannot reduce the step finish time any further. Note that since all the map tasks are started at $t = 0$, the produce rate r_p is non-increasing in time. Consequently, if the full consume rate r_c of the reduce task is lower than the *average* produce rate, then the finish time of the step will be bottlenecked by r_c, i.e., $t^*_{e,\texttt{s1}} = P/r_c$. On the other hand, if r_c is higher than the average produce rate, the bottleneck shifts to r_p, and the reduce task can only finish when the map tasks finish generating data at time t_m. Figure 4 (b) shows the latter scenario, where $r_c = 3$ is higher than the average produce rate ($= 2$), and therefore $t^*_{e,\texttt{s1}} = t_m = 3$.

Step 3: Calculate optimal launch time T^*_s. We find that launching the reduce task at $T^*_s = T^*_e - D^*$, where D^* and T^*_e are computed via the lazy (Step 1) and eager (Step 2) approaches, respectively, ensure that the task is optimal in both execution duration and finish time. This is shown in Figure 4 (c), where starting the reduce task at $T^*_s = 3 - 2 = 1$ ensures optimal duration ($D^* = 2$), as well as finish time ($T^*_e = 3$). At first glance, this may seem obvious, since D^* and T^*_e already correspond to optimal task duration and finish time, respectively. But we note that since D^* and T^*_e were computed for two separate approaches, it is not obvious if an approach that starts the task at $T^*_s = T^*_e - D^*$ will always ensure the task takes exactly D^* time to finish. Fortunately, for two-stage map-reduce jobs, we have the following theorem:

Theorem 4.1 *For a reduce task, we can always achieve both optimal execution duration and finish time by launching it at time $T^*_s = T^*_e - D^*$, where T^*_e is the optimal finish time and D^* is the optimal duration computed using Steps 1 and 2 above.*

Proof Since the duration of step `r.s2` is independent of when the reduce task is scheduled, we only need to prove the optimality of finish time and duration for step `r.s1`.

We first show that we can always achieve optimal finish time $t^*_{e,\texttt{s1}}$ if we launch the reduce task at time $T^*_s = T^*_e - D^*$. We prove this by contradiction: assume that a reduce task that is started at T^*_s does not finish executing its first step `r.s1` at $t^*_{e,\texttt{s1}}$. This must be because at some time point $\in [T^*_s, t^*_{e,\texttt{s1}}]$, the task was unable to consume data at full consume rate r_c. We denote the last time instant where this was true as t'. Note that the data produced until time t' (say, $P_{t \le t'}$) must be less than the data that can be consumed by time t' at full consume rate r_c, i.e., $P_{t \le t'} < (t' - T^*_s) \times r_c$. Since the total amount of data produced is $P = d^*_{\texttt{s1}} \times r_c$, the data produced *after* t' must be $P_{t > t'} = P - P_{t \le t'} > (t^*_{e,\texttt{s1}} - t') \times r_c$, and the reduce task will take more time than $(t^*_{e,\texttt{s1}} - t')$ to consume it (since it can consume data at a rate no faster than r_c).

Note that the data produced after t', $P_{t > t'}$, is independent of the reduce task's launch time. This implies that regardless of how early the task is launched, no solution could have achieved optimal finish time $t^*_{e,\texttt{s1}}$ for the step `r.s1`. However, this contradicts with the fact that the eager solution can achieve the optimal finish time by launching the task at $t = 0$. Therefore, our initial assumption must have been false: a reduce task that is started at T^*_s *does* finish executing its first step `r.s1` at $t^*_{e,\texttt{s1}}$.

Proving $T^*_s = T^*_e - D^*$ results in optimal task duration is then trivial: since step `r.s1` finishes at $t^*_{e,\texttt{s1}}$ with start time T^*_s, the corresponding duration $T^*_s - t^*_{e,\texttt{s1}}$ will always be $d^*_{\texttt{s1}}$. ■

Note that the T^*_s for different reduce tasks may be different, since the produce rate r_p to different reduce tasks may vary (*e.g.*, due to data skew). Recall that the finish time of the job is the same as the finish time of the last reduce task, and the total cost of the job is proportional to the aggregated duration of all tasks. As such, Theorem 4.1 shows that we can simultaneously achieve both optimal cost and finish time for the entire job, as long as each reduce task is optimal in duration and finish time, i.e., is launched at T^*_s.

4.2 NIMBLE for General Analytics

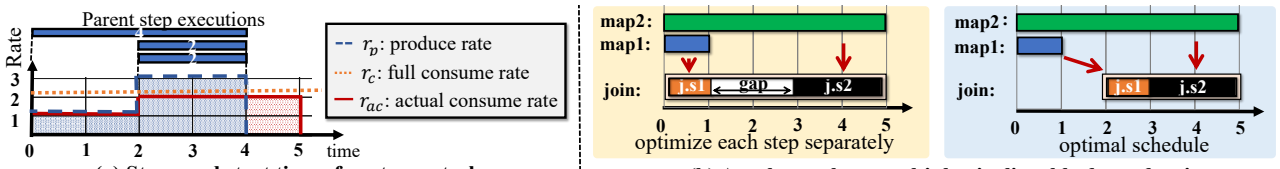
We now extend our analysis to general analytics jobs. We first outline the steps in computing the optimal launch time for tasks in jobs with arbitrary execution DAGs, and then describe how NIMBLE scheduling can be generalized to such DAGs.

4.2.1 Optimal launch time for individual tasks

General analytics jobs with arbitrary execution DAGs introduce two main challenges in determining the optimal task launch time as defined in §4.1. First, unlike two-stage map-reduce jobs, the start times of a step's parent steps need not start at $t = 0$ and can be staggered in time, as shown in Figure 5 (a). This breaks our assumption of a nonincreasing $r_p(t)$ from §4.1, and necessitates a more nuanced treatment of the eager approach to compute the optimal task finish time.

Second, unlike two-stage map-reduce jobs, general analytics jobs, a stage may contain multiple parent-child step pairs, *e.g.*, the join stage in Figure 3 has two steps, and each step has a parent step from a different map stage. For such dependencies, we find that optimally overlapping each parent-child step pair is insufficient to ensure optimal task duration and finish time. Specifically, the optimal task launch time depends not only on the *inter-stage* dependency between parent-child step pairs, but also on the *intra-stage* dependency between steps in the same stage. Figure 5 (b, left) shows a join example where the optimal launch for each step in the task is computed independently. Although each child step is optimally pipelined with its parent, the gap between their execution corresponds to time where no useful work is done, resulting in sub-optimal task duration and, therefore, cost of execution. Figure 5 (b, right) shows how this can be avoided by *deferring* the start time of the first step.

We exploit the above insights to extend NIMBLE scheduling approach from §4.1 to general analytics jobs. We consider the



(a) Staggered start time of upstream tasks **(b) A task may have multiple pipelineable dependencies**

Figure 5: **NIMBLE scheduling for general analytics (§4.2)** (a) Unlike two-stage map-reduce, the produce rate for a task may not be nonincreasing, since the parent step in different tasks can have different start times. (b) Unlike two-stage map-reduce, tasks may have multiple pipelineable dependencies, which requires careful handling to ensure optimal task duration. See §4.2.1 for details.

general case where a task comprises n steps, s1-sn, and make two assumptions to simplify our analysis: (i) each step in a task has at most one parent step, and (ii) steps within a task are executed sequentially in a fixed order. Both assumptions hold for a wide range of analytics jobs, including the join example above, and all of our evaluated workloads (§6). Similar to two-stage map-reduce (§4.1), the optimal launch time T_s^* for a task in the execution DAG is calculated in three steps:

Step 1: Calculate optimal task duration D^*. The optimal task duration D^* is simply the sum of individual optimal step durations $d^*_{\mathtt{s}i}$, $1 \le i \le n$. As in §4.1, the duration for steps without a parent is independent of task launch time, while the optimal duration for steps with a parent is computed using the lazy approach, i.e., P/r_c for the corresponding step. In Figure 5 (a), $P = 8$ and $r_c = 2$, so $d^*_{\mathtt{s}i} = 4$.

Step 2: Calculate optimal finish time T_e^*. Since T_e^* is bound by the finish time of the last step sn, we first compute the optimal finish time of a step as computed by the eager approach, similar to §4.1. As noted earlier, however, unlike two-stage map-reduce where the parent step across all the map tasks start at time $t = 0$, the parent step across different tasks in a general DAG may start and end at arbitrary times. This is depicted in Figure 5 (a) where the parent step across two upstream tasks start at time $t = 2$, while the third starts at $t = 0$. Consequently, the produce rate is no longer non-increasing. As such, the optimal step finish time (based on the eager approach) can only be determined by tracking the actual consume rate r_{ac} over time. In the example, r_{ac} is bound by r_p $(= 1)$ between $t = 0-2$, lower than r_p $(= 3)$ and bound by r_c $(= 2)$ between $t = 2-4$, and equals to r_c $(= 2)$ between $t = 4-5$ to clear the surplus data generated between $t = 2-4$. As such, the finish time yielded by the eager approach is 5.

In order to generalize the above example, we discretize time into slots t_1, t_2, ..., t_m, such that the produce rate is constant within a time slot. We introduce a new function $S(t_i)$ to identify time slots where the step accumulates surplus data, i.e., $S(t_i) = 0$ if all the input data produced until t_i has been consumed by time t_i, and 1 otherwise. It is easy to see that when there is no surplus data $(S(t_i) = 0)$, the actual consume rate $r_{ac}(t_i)$ is upper-bounded by the produce rate $r_p(t_i)$. When there is surplus data $(S(t_i) = 1)$, the actual consume rate increases to the full consume rate $(r_{ac} = r_c)$ to clear the surplus. Formally,

$$r_{ac}(t_i) = \begin{cases} \min\left(r_p(t_i), r_c\right) & \text{if } S(t_i) = 0 \\ r_c & \text{if } S(t_i) = 1 \end{cases} \quad (1)$$

For each t_i, we can calculate $S(t_i)$ based on $S(t_{i-1})$, $r_{ac}(t_{i-1})$ and $r_p(t_{i-1})$, and $r_{ac}(t_i)$ based on Equation 1. The time slot t_n where the cumulative data consumed so far equals P, the total input data for the step, corresponds to the optimal finish time; we formally prove optimality in Appendix A.

Unlike the two-stage map-reduce job in §4.1, we have to consider one more constraint — step si can only start after step si-1 has finished, i.e., the finish time of step i is no less than $t^*_{e,\mathtt{s}i\text{-}1} + d^*_{\mathtt{s}i}$. Let the optimal step finish time for si as computed above (which only considers its parent step) be $t'_{e,\mathtt{s}i}$, then the actual optimal finish time of step i is:

$$t^*_{e,\mathtt{s}i} = \begin{cases} \max\left(t'_{e,\mathtt{s}i}, t^*_{e,\mathtt{s}i\text{-}1} + d^*_{\mathtt{s}i}\right) & \text{if s}i \text{ has a parent} \\ t^*_{e,\mathtt{s}i\text{-}1} + d^*_{\mathtt{s}i} & \text{otherwise} \end{cases} \quad (2)$$

We compute the optimal task finish time by iteratively calculating the optimal finish time for each step s1−sn. Figure 5 (b) shows an example for this computation: the task finish time equals the $t'_{e,\mathtt{s}2}$ $(= 5)$, since $t^*_{e,\mathtt{s}1} + d^*_{\mathtt{s}2}$ $(= 1+2)$ is smaller.

Step 3: Calculate optimal launch time T_s^*. As in §4.1, the optimal launch time is computed as $T_s^* = T_e^* - D^*$. Consider the example in Figure 5 (b); the optimal launch time is calculated as $T_s^* = T_e^* - D^*$ $(= 5-3)$ for the two steps. Compared to Figure 5 (b, left), doing so automatically delays the first step and removes the gap (Figure 5 (b, right)).

Indeed, Theorem 4.1 extends to general analytics jobs:

Theorem 4.2 *For a task in an analytics job with an arbitrary execution DAG, given the execution (produce rate) of all its parent steps, we can always achieve both optimal execution duration and finish time by launching it at $T_s^* = T_e^* - D^*$, where T_e^* is the optimal finish time and D^* is the optimal duration computed using Steps 1 and 2 above.*

We defer the formal proof to Appendix A, but note here that it employs induction on the number of steps: we assume the statement holds for a task with $n-1$ steps, and use Theorem 4.1 to show that it still holds on adding one more step.

4.2.2 Optimal schedule for the entire job

Algorithm 1 shows NIMBLE scheduling for the entire job based on Theorem 4.2. Stages in the job are scheduled iteratively based on their dependencies: a stage is scheduled when all of its parent stages in the execution DAG have been scheduled. For each task within a scheduled stage, we first calculate the produce rate from its parent stages, and then calculate its optimal launch time as described above.

Algorithm 1 ensures that *each task* achieves optimal duration and finish time given its parent execution (due to Theo-

Algorithm 1 NIMBLE scheduling for a job

```
Launch all stages with no parent stages.
U ← Set of unscheduled stages
while U ≠ ∅ do
    for each stage S ∈ U, whose parent stages are scheduled do
        for each task in stage S do
            Calculate r_p for each using parent stage schedules
            Calculate T_e^* and D^* based on r_p and r_c of each step
            Calculate T_s^* = T_e^* − D^*
```

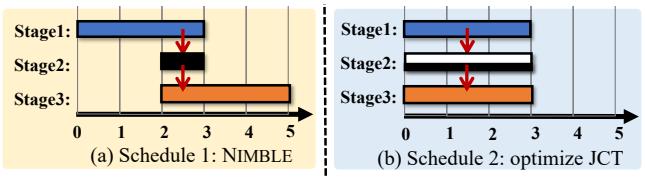


Figure 6: **Example of a job that cannot achieve both (a) optimal cost and (b) finish time simultaneously.** Each stage comprises a single task/step. Stage 2 has a produce rate of 1 and consume rate of 3. Stage 3 has a produce rate of 3 and consume rate of 1.

rem 4.2). However, it still leaves the question: does the algorithm also ensure optimal finish time and cost for the *entire job*? We find that the answer is in the affirmative for jobs with DAGs of depth two, including the map-reduce and SQL jobs in Fig 3. Intuitively, since the stages in the first level of the DAG do not have parent steps, their optimal start time is $t = 0$. As such, given the execution of the stages in the first level, Theorem 4.2 ensures optimal duration and finish time for the stages in the second level of the DAG.

Unfortunately, for general analytics jobs with arbitrary DAGs, the answer is in the negative. In fact, we find that for some jobs, it is *impossible* to find a schedule that achieves both cost and JCT optimality. The key insight behind this observation is that the launch time of a task affects not only itself, but also its downstream tasks — the optimal launch time for one task (Theorem 4.2) may negatively affect tasks in its downstream. We illustrate this with the example in Figure 6, that shows a job with three stages, each with only one step and one task. Stage 2 has a produce rate of 1 and consume rate of 3. Stage 3 has a produce rate of 3 and consume rate of 1. The arrows denote parent-child step pairs. Figure 6 (a) shows NIMBLE approach, which greedily optimizes the duration and finish time from Stage 1-3 based on the produce rate of each stage's parent steps; the end-to-end execution time of the entire job is 5, while its cost is 7. Figure 6 (b) shows an alternative strategy, that launches tasks across all three stages (Stage 1-3) at $t = 0$. This increases the duration of Stage 2 from 1 to 3, and consequently, the cost of execution of the job from 7 to 9. However, doing so also reduces the produce rate for Stage 3 to 1, allowing Stage 3 be completely pipelined with Stage 2. As such, the finish time of the entire job reduces from 5 to 3. Note that no schedule can achieve both a JCT of 3 and a cost of 7, since optimal JCT can only be achieved if Stage 2 and 3 are started at $t = 0$, which ensures a sub-optimal cost.

Cost optimality & cost-JCT *Pareto-optimality*. Despite the negative result above, NIMBLE scheduling efficiently navigates the cost-JCT tradeoff for jobs with arbitrary DAGs:

Theorem 4.3 *For a job with arbitrary DAG,* NIMBLE *scheduling in Algorithm 1 is (1) optimal in cost; and (2) Pareto-optimal between cost and JCT.*

Proof We first consider cost-optimality: since Algorithm 1 ensures optimal execution duration for each task in a job (Theorem 4.2), the aggregated duration across all tasks in the job, and therefore, the job execution cost, is also optimal.

Since the cost is always optimal, for Pareto-optimality we only need to show that no solution can further reduce job finish time without also increasing its cost. Our proof builds on the intuition developed for the example in Figure 6. First, we note that delaying the start time beyond T_s^* for any task cannot reduce its completion time; the only possibility to reduce JCT is to pick a start time earlier than T_s^*. As per Theorem 4.2, starting a task any sooner than T_s^* must increase its duration. Moreover, doing so will not reduce the duration of any other task, since they are already optimal. Thus, even if starting the task before T_s^* did improve JCT, it would only do so by increasing the aggregate duration across all tasks in the job, and therefore, its cost. ■

As an interesting aside, we note that for the example in Figure 6, we face this hard tradeoff between JCT and cost optimality because Stage 3 has a larger duration compared to Stage 2. Instead, if Stage 3 had a duration of 0.5, starting Stage 2 any sooner than $t = 2$ (at higher cost) would not have made stage 3 finish any faster. In practice, downstream stages often have a shorter duration compared to the upstream stages, since frequently used operators such as reduce, filter and join often significantly reduce the output data volume to downstream stages. In such cases, NIMBLE can achieve both optimal cost and JCT simultaneously — evaluation results on a wide range of analytics jobs in §6 validate this argument.

5 Design Details

In this section, we describe how we incorporate NIMBLE scheduling into Caerus, a new fine-grained task-level scheduler for serverless analytics frameworks. We first describe Caerus design components and application workflow (§5.1), and then describe its implementation details (§5.2).

5.1 Caerus System

We now describe Caerus system components and how they fit together (Figure 7). Before describing these components, we first briefly summarize the design employed by existing serverless analytics frameworks.

Primer on serverless analytics frameworks. Recent proposals on serverless analytics frameworks [8, 12, 14] share similar designs. Figure 7 depicts this design (adapted from [12]). The framework takes as input a job execution plan (DAG) that captures dependencies between stages and the number of tasks within each stage. It uses this to generate code for the individual tasks, compiles it and packages it with necessary dependen-

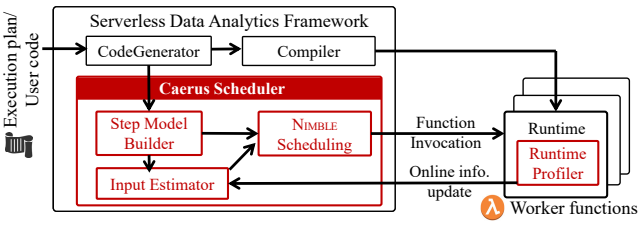


Figure 7: **Caerus system components & workflow (§5.1).**

cies. To execute a job, a scheduler launches tasks as serverless functions and monitors their progress. Pywren [8, 14] is similar, but omits the code-generation and compilation steps and directly takes task code and execution plan as input.

Caerus integrates with these analytics frameworks by simply replacing their task-level scheduler, and taking over the task launching and monitoring responsibilities. We next describe the major components of Caerus scheduler (Figure 7) in detail.

The step model builder is responsible for extracting the fine-grained step dependency model that NIMBLE scheduling expects, either from the job's execution plan or the user code. If the input is user code (e.g., a Python function in PyWren [8, 14]), Caerus provides a step annotation API that users can employ to specify the step information Caerus expects:

```
s = createStep() # Create a step object
s.start() # Notify system about step start
s.end() # Notify system about step end
s.addParent(stageID, stepID) # Specify parent step
```

If the query code is generated by a CodeGenerator based on the execution plan (as in Starling [12]), the step dependencies can be extracted during code generation. Most popular query execution engines (*e.g.*, SparkSQL) generate code based on the Volcano [44] iterator or `WholeStageCodegen` [45] model, which fuses operators as much as possible to maximize pipelining. As such, the generated code in such models is already composed of blocks separated by pipeline breakers, where each block corresponds exactly to a step in our step model. Caerus augments the CodeGenerator to additionally generate step-annotations at the start and end of blocks, along with step dependencies, using the step annotation API outlined above.

Input estimator & runtime profiler. Recall from §3 that NIMBLE scheduling relies on estimates of step produce rate (r_p) and consume rate (r_c) for steps with parents, and duration (d^*_{si}) for steps without parents, to make scheduling decisions. To facilitate accurate estimates, we leverage the observation that task and job-level statistics can be accurately estimated by tracking profiled information from prior job runs, since such analytics jobs in production workloads tend to be recurring in nature [26–28, 46, 47]. In particular, the input estimator in Caerus is responsible for collecting information for prior executions for each job (i.e., the job history) and maintaining estimates for various r_p, r_c and d^*_{si} values.

For higher accuracy, the input estimator continuously refines its r_p, r_c and d^*_{si} estimates based on realtime task progress. To facilitate this, a runtime profiler (similar to [27, 40]) periodically profiles and reports such metadata to the input estimator. Our runtime profiler is incorporated into the function runtime in serverless frameworks [8] that is shared across all tasks.

Caerus workflow. For each job, the step-model builder first extracts the step model from code generator or directly from the annotated function code. The input estimator maintains estimates of algorithm inputs (r_p, r_c and d^*_{si} values) for each step based on prior job runs. The NIMBLE scheduling module then calculates launch time based on both the algorithm inputs and step model, and launches each task at the calculated time. Launched tasks periodically report their progress to the input estimator via the runtime profiler, which is leveraged to refine the input estimates for future runs.

Caerus scalability. Caerus's scheduling performance scales well with the number of available CPU cores due to two main reasons. First, tasks within a stage have independent launch times, which permits parallel calculation and launching. Second, while the number of online update messages from runtime profiler grows linearly with the number of tasks, it can be served in parallel by partitioning input estimates for different tasks across different CPU cores.

Fault-tolerance. Caerus handles task failures by restarting them. For controller fault-tolerance, Caerus relies on traditional primary-backup mechanisms [30, 48]. The backup maintains consistent copies of the job's step model, launched and queued tasks, and runtime profiled information from prior job runs. During recovery, Caerus fetches this metadata from the backup and resumes scheduling queued tasks using NIMBLE.

5.2 Caerus Implementation

Our Caerus prototype is implemented atop Pywren [8], a serverless data analytics engine that runs on AWS Lambda [1]. We use Amazon S3 [21] for persistent data storage and Jiffy [24] for intermediate data storage.

SQL analytics with Caerus. We implement a SQL query execution framework atop Locus to highlight the benefit for Caerus scheduling for SQL analytics workloads. We employ Apache Spark's query planner to generate the query plan from the original SQL query, and then use Pandas to implement the SQL operators. Pandas' current implementation for SQL operators (like `JOIN` and `GROUPBY`) employs the lazy approach, *e.g.*, for the join example in Figure 3 (bottom), Pandas would only start the `JOIN` operation after all the data in both input tables are ready. We therefore modify the operator implementations in Pandas to conform to the step dependency model required by NIMBLE scheduling.

As a concrete example consider the implementation of the SQL job in Figure 3 (bottom) in Caerus. For the first two map stages, each task keeps reading input data from S3. After reading each small chunk of input data, it performs the map function, and partitions the output data into *chunks* based on key hashes. To implement shuffle, we maintain a FIFO queue for each join task in the intermediate storage. Once an output

chunk is ready, the map task pushes it to the corresponding join task's queue. Note that a join task receives data from two shuffles (i.e., from map1 and map2). As such, each join task has two receiver queues: queue A (for data from `m1.s1`) and queue B (or data from `m2.s1`). After being launched, each join task fetches data from queue A and builds the hash table incrementally in step `j.s1`. Once the hash-table is built, the step `j.s2` fetches data from queue B, performs a join of the fetched data with the data in the hash-table, and writes the output to persistent storage.

Identifying pipeline-breakers. In Caerus, we implement all commonly used SQL operators (*e.g.*, `FILTER`, `JOIN`, `SORT`, `GROUP BY`, aggregates, etc.), employ the widely-used Volcano iterator model [44] to identify pipeline-breakers, and specify them using the step annotation API. As such, Caerus can run all TPC-DS and Big-data benchmark queries — we evaluate a representative subset in §6.

Accurate parameter estimations. NIMBLE scheduling relies on accurate estimation of various parameters (r_p, r_c, d^*_{si}), which can be complicated due unpredictable variations stemming from a range of sources. Fortunately, we found a majority of these sources had little to no variation across AWS Lambda executions, including (1) processing time for various operators; (2) function launch times[5]; and (3) function ingress/egress bandwidth to intermediate storage.

However, we did observe unpredictable performance variations for Amazon S3 reads and writes, particularly with larger number of parallel tasks (≥ 100). To minimize parameter estimation errors caused by these variations, we adopt a straggler mitigation technique for S3 reads and writes similar to [12] — Caerus tasks proactively establishes a new connection to S3 when a transfer takes longer than expected, and uses the response from whichever connection performs the read or write first. Moreover, we found larger S3 reads/writes to have unpredictable durations, so we break them into multiple smaller chunks. We show in §6 how these modifications ensure negligible estimations errors for a wide range of evaluated workloads.

6 Evaluation

We now evaluate Caerus implementation (§5.2) using three analytics workloads: TeraSort benchmark (§6.1), TPC-DS Benchmark (§6.2) and BigData Benchmark (§6.3). All of our experiments use Lambda instances with 3GB memory and deploy Jiffy on 6 m4.16xlarge EC2 instances.

Compared approaches. We compare Caerus with the eager and lazy scheduling approaches, implemented as a part of Caerus scheduler. These scheduling approaches correspond to the two extremes typically used in server-centric analytics frameworks for task level scheduling — lazy in Spark [30] and MapReduce [48], and eager in Dryad [32] and MapReduce Online [29]). Note that since our main contribution is a new scheduler for serverless analytics, our evaluation focuses on comparing scheduling approaches on a common analytics framework as opposed to comparing different analytics frameworks. As noted in §5, Caerus can integrate with any of existing serverless analytics frameworks [8, 11, 12, 14] and inherit their specific performance optimizations.

[5] We ensure function invocations are warm to avoid cold-start delays.

	Lazy	Eager	NIMBLE
JCT(s)	124	105	107
Cost(s)	10776	15756	11169

Table 1: **Comparison of NIMBLE against lazy and eager approaches for TeraSort on a 100GB dataset (§6.1).**

Performance metrics. We focus on two main metrics: JCT and cost of job execution. The former is measured as the time between job's first task's launch time to the last task's finish time. For the latter, we measure the aggregated duration across all tasks in the job as a proxy for cost. We avoid reporting precise dollar values, since these depend on the cloud provider and can change with market economics.

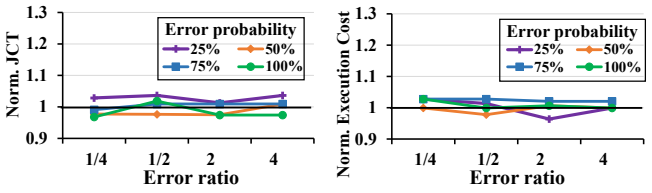
6.1 TeraSort

We port the TeraSort algorithm [49] implementation from Locus [14] to our framework for sorting large datasets. The algorithm operates in two stages: a partition stage that range partitions input data to intermediate storage, and a merge stage that reads these partitions, merges, sorts and writes them out as output. The sort job in our experiments uses 100 lambdas for both the map and reduce stage to sort 100GB of data generated using the Sort benchmark tool [50].

Table 1 compares the results of eager, lazy and NIMBLE scheduling approaches for the sort job. We observe little data skew for the TeraSort benchmark during both the partition and merge stages, and the ideal launch time for merge tasks identified by NIMBLE scheduling is roughly in the middle of the execution for partition stage. As such, NIMBLE achieves $1.16\times$ lower job completion time compared to the lazy scheme, $1.41\times$ lower cost than the eager approach. The results validate our analysis in §4, that NIMBLE scheduling can achieve near-optimal JCT and cost simultaneously for two stage map-reduce jobs in practice ($< 4\%$ in Table 1). NIMBLE's slight departure from optimal is due to delays in launch times introduced by the analytics framework (i.e., PyWren).

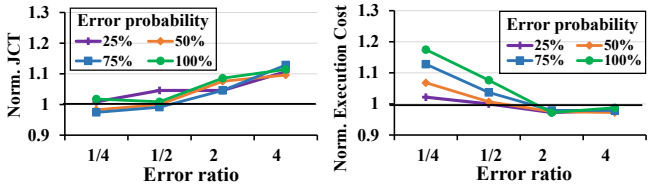
Impact of estimation errors. Caerus's JCT and cost-efficiency is gated on being able to estimate parameters like produce rate (r_p) and consume rate (r_p) accurately. Since Caerus's estimation errors are quite small in practice ($< 4\%$), we study their impact by introducing errors artificially.

To inject errors in produce rate estimation, we randomly select map tasks in our TeraSort job with probability p_e, and for each of them, incorrectly estimate the data output rate by Caerus's offline estimation as $k\times$ the actual value. We denote p_e as error probability and k as the error ratio. Since the produce rate r_p is the aggregated data output rate across all map tasks, our stochastic approach effectively injects errors



(a) JCT (b) Cost

Figure 8: **Impact of produce rate estimation errors (§6.1).** The results are normalized against the performance with no injected errors.

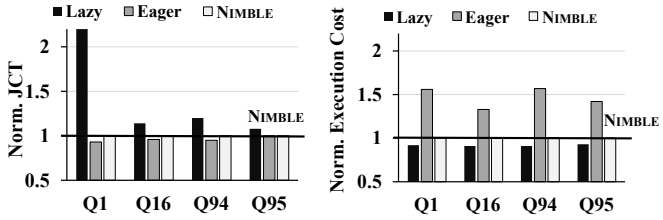


(a) JCT (b) Cost

Figure 9: **Impact of consume rate estimation errors (§6.1).** The results are normalized against the performance with no injected errors.

to the r_p estimate as well. Figure 8 shows the impact of the injected estimation errors on NIMBLE's performance, i.e., JCT and execution cost, with the corresponding metrics normalized against a run with no injected errors. We observe that NIMBLE's performance is minimally affected — across various combinations of error probability and error ratio, the JCT and execution cost is always within $\sim 4\%$ of the run with no injected errors. We attribute this to the runtime profiler, which tracks the real-time progress of each map task and refines the produce rate estimation by continuously re-estimating the task output rates. As such, the runtime profiler is able to correct the offline estimations in produce rate before launching the reducers, minimizing the impact of errors.

To study the impact of consume rate estimation errors, we employ a similar error rate and error ratio driven approach for reduce tasks. Note that runtime profiler is unable to correct for estimation errors in this case, since it can re-estimate the consume rate only *during* reduce task executions, which is *after* the reduce tasks have already been launched. Figure 8 shows the impact of injected errors on NIMBLE performance. For error ratio > 1 (i.e., estimated rate > actual rate), NIMBLE incorrectly estimates that the reduce task would finish faster than it actually does, while for error ratio < 1, it assumes the opposite. As expected, for the former case, Caerus launches reduce tasks later than it should, resulting in a longer JCT, while for the latter scenario, it launches them sooner than necessary, resulting in increased cost. Figure 9(a) shows that the normalized JCT increases from $1.07\times$ to $1.12\times$ as error ratio is increased from 2 to 4, while Figure 9(b) shows that the cost increases from $1.08\times$ to $1.18\times$ as error ratio increases from $1/2$ to $1/4$. Moreover, at higher error probability, the cost increase is greater since more reducers are launched earlier than necessary; while the JCT increase is largely unaffected since it only depends on the *slowest* task. Note that even with extreme estimation errors ($\frac{1}{4}\times$ and $4\times$), the increase in cost or JCT is only 12–18%.



(a) JCT (b) Cost

Figure 10: **NIMBLE performance for TPC-DS queries (§6.2).** Its JCT is comparable to eager and 1.08–2.2× lower than lazy, while its cost is comparable to lazy and 1.33–1.57× lower than eager.

6.2 TPC-DS Benchmark

The TPC-DS benchmark [51] has a set of standard decision support queries based on those used by retail product suppliers. The queries vary widely in terms of compute, storage and network I/O load variations. We evaluate Caerus on TPC-DS with scale factor of 1000, which results in a total input size of 1TB across various tables. Similar to Locus [14], we evaluate four representative queries (in terms of performance characteristics) from the TPC-DS Benchmark, specifically, queries `Q1`, `Q16`, `Q94` and `Q95`. All selected queries have complex DAGs comprising six to eight stages, with each query operating over a subset of the 1TB input — varying from 33GB to 312GB. Note that some late stages in the selected queries process much less data compare to early stages (after several `join` and `groupby` operations) — we adjust the degree of parallelism for these stages based on the amount of data they process.

Figure 10 compares the performance for NIMBLE with the lazy and eager approaches. The results indicate that Caerus can efficiently navigate the JCT-cost trade-off for all evaluated queries. Specifically, NIMBLE achieves JCT comparable to eager for all the queries, while outperforming lazy by 1.08–2.2×. For cost, NIMBLE matches the lazy approach while outperforming eager by 1.33–1.57×.

6.2.1 Diving deeper into NIMBLE benefits

In order to better understand the gains enabled by NIMBLE scheduling, we zoom in on the performance for Query `Q1` of the TPC-DS benchmark. Figure 11 shows the step-level dependencies for `Q1`, while Figure 12 shows the breakdown of execution time across different stages. Note that compute and network I/O take up most of the execution time, highlighting potential gains from pipelining. Figure 14 shows the job execution breakdowns with lazy, eager and NIMBLE scheduling.

Optimal pipelining across stages. We now walk through `Q1`'s execution with Caerus (Figure 14(c)). Caerus identifies 7 step dependencies (i.e., parent-child step pairs) as pipelineable, shown as red arrows in Figure 11.

While all map stages are launched at time $t = 0$, (since they do not have upstream dependencies), Caerus launches tasks across subsequent stages in a manner that ensures child steps in the above parent-child step pairs are optimally pipelined with the parent step, which corresponds to a large portion of the query execution. This is highlighted in Figure 14(c): when con-

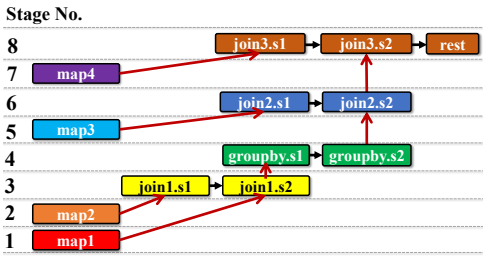


Figure 11: **The step dependency model for Q1.**

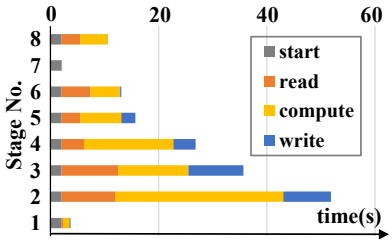


Figure 12: **Time breakdown for Q1.**

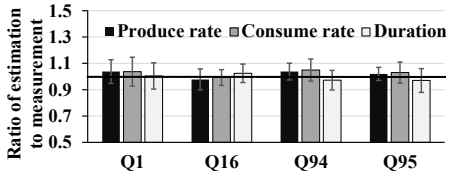
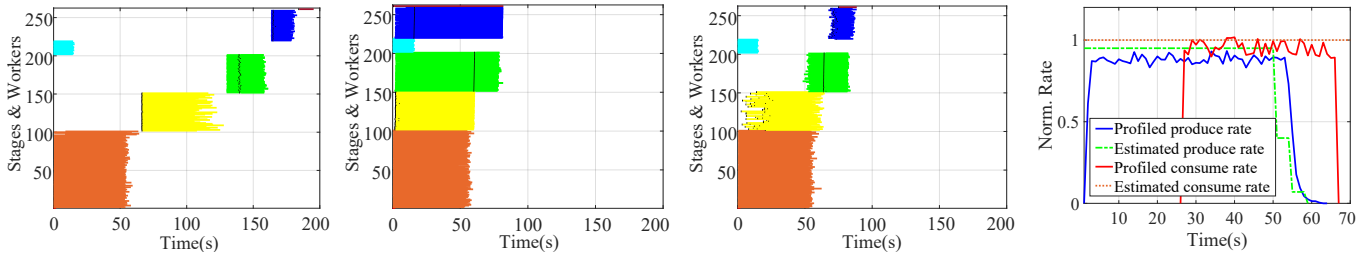


Figure 13: **Ratio of estimated & measured parameters for TPC-DS queries**. Error bars denote standard deviation across all tasks.



(a) Lazy (b) Eager (c) NIMBLE (d) NIMBLE Input Parameters

Figure 14: **Diving deeper into NIMBLE benefits for TPC-DS query Q1 (§6.2.1).** (a, b, c) show Q1 execution breakdown for lazy, eager, and NIMBLE, respectively; the black dots inside a task denote pipeline-breakers between steps. The degree of parallelism for Stages 1-8 is: {1, 100, 50, 50, 20, 40, 1, 1}. Note that Stages 1 (red) and 7 (purple) contain only one very short task, making them hard to see. (d) NIMBLE input parameters as measured by Caerus runtime profiler (solid lines) and as estimated by input estimator (dashed lines) for part of Stage 3 (yellow).

trasted with the lazy approach in Figure 14(a), Caerus enables a JCT that is 2.2× lower than the lazy approach. Meanwhile, Caerus also ensures that the tasks are not launched too soon in order to minimize time spent waiting for input from the parent step to become available, and therefore, the end-to-end job execution cost. As a concrete example, since step `groupby1.s1` is much shorter than step `join1.s2` and cannot finish before `join1.s2`, tasks in the `groupby` stage are started after tasks in `join1` stage are started, but before they finish execution. Compared with Figure 14(b), this allows name to Caerus achieve a cost that is 1.56× lower than the lazy approach.

Decreasing duration across stages. Another interesting takeaway from Q1's execution is that downstream stages in general process smaller amounts of data than upstream stages (since operators such as filter and join significantly reduce the data to downstream stages), and consequently have shorter durations. As noted in §4.2, NIMBLE scheduling enables both optimal cost and JCT simultaneously for such DAGs, which is reflected in Figure 14. Moreover, this observation holds across all of our evaluated TPC-DS queries, ensuring cost and JCT optimality with Caerus for all of them.

Accurate profiling & estimation for NIMBLE inputs. Figure 14(d) shows the normalized produce rate and consume rate of of step `join1.s2` in Stage 3, as profiled by Caerus runtime profiler and as estimated by Caerus input estimator. We make two observations: (1) the consume rate is stable as a function of time, as modeled in §3, and (2) the estimated values are a close approximation of the actual produce and consume rates. We find these observations extend to all stages across query Q1, as well as to all other queries we evaluate in this section — Figure 13 shows that the average error in parameter estimations for r_p, r_c and d^*_{si} is within 4% across all queries. As we already saw in §6.1, NIMBLE scheduling is also robust to higher estimation errors.

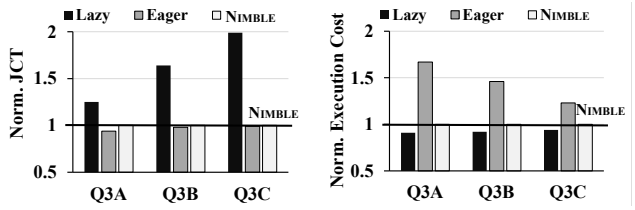
Data skew. We note that Stage 3 (yellow) experiences data skew across tasks (Figure 14(a)-14(c)) — our profiling indicates that some tasks process $> 1.6\times$ more data than others. Caerus captures the effect of such data skew in its NIMBLE scheduling algorithm, and launches tasks in Stage 3 at a time that still ensures JCT and cost optimality for the job execution.

Fast scheduling decisions. The query Q1 has over 250 tasks across 8 stages — Caerus schedules and launches each task in about 400μs (on average). In contrast, when the task launch request is issued to AWS Lambda, it typically takes an additional $\sim 25-320$ms to start the task's execution [52]. As such, despite making much more fine-grained (i.e., task-level) decisions than traditional job schedulers, Caerus is fast enough to not be the bottleneck in the analytics execution pipeline.

6.3 BigData Benchmark

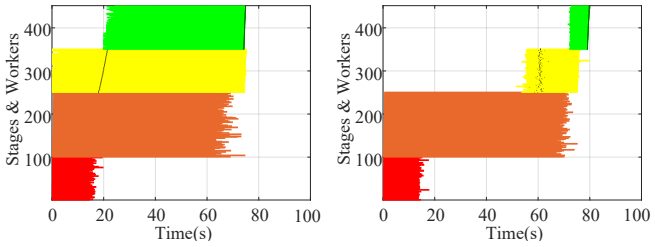
The Big Data Benchmark [53] is a query suite derived from production databases. We consider Query 3 (Q3), which is a join query with four stages, with a step dependency model similar to the first four stages of TPC-DS benchmark's Q1 (Figure 11). Our implementation uses shuffle hash join (SHJ), and efficiently pipelines the join stage with the map stages. Q3 reads in 123GB of input, and can perform joins with three different sizes: 485,312 rows in Q3A; 53,332,015 rows for Q3B; and 533,287,121 rows for Q3C. This allows us to understand the effect of join size on NIMBLE scheduling.

Figure 15 compares NIMBLE approach with both the lazy and eager approaches with different join data sizes (Q3A-Q3C).



(a) JCT (b) Cost

Figure 15: **NIMBLE performance for BigData Benchmark (§6.3).** NIMBLE's JCT improvement over lazy increases as join size increases (Q3A→Q3C), while its cost improvement over eager increases as join size decreases (Q3C→Q3A).



(a) Eager (b) NIMBLE

Figure 16: **Q3A execution breakdown** for (a) eager and (b) NIMBLE.

Interestingly, we observe that NIMBLE's relative JCT improvement compared to lazy increases from 1.29× to 1.99×, as join size increases from Q3A to Q3C. Meanwhile, NIMBLE's relative cost improvement compared to eager increases from 1.23× to 1.67×, as join size decreases from Q3C to Q3A.

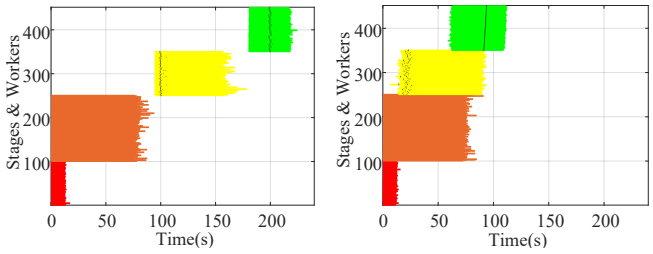
To better understand the differences in cost and JCT improvements due join sizes, Figure 16 shows the execution breakdown for Q3A. As Q3A's join input data size is small, the `join` (yellow) and subsequent `groupby` (green) stages are much shorter than the initial `map` stage (orange). As such, pipelining these shorter stages with the `map` stage does not improve the JCT by much (1.29×) compared to the lazy approach. However, the eager solution significantly increases the cost by starting these short tasks very early (Figure 16(a)). Caerus, on the other hand, improves cost relative to eager by 1.67× by launching them at just the right time (Figure 16(b)).

Figure 17 compares the execution of lazy and Caerus for Q3C: as the input data size for join is now much larger, the duration of the join stage and groupby stage is comparable to the map stage (orange). As such, the eager approach does not lose as much in terms of cost by launching these tasks early. However, the lazy solution increases the JCT significantly by running these relatively longer stages one after the other (Figure 17(a)). In contrast, Caerus improves JCT by 1.99× relative to lazy by efficiently pipelining the join and groupby stage with the map stages (Figure 17(b)).

7 Related Work

We already discussed related work on server-centric and serverless analytics frameworks in §2.1 and §5.1. We now discuss prior work related to Caerus in other areas.

Some databases [45, 54–56] and data processing frameworks [31, 57] support pipelined execution via an *iterator model* [44]. These approaches focus on maximally pipelining operators to minimize query completion time. Similar to these works, we leverages pipelined execution to achieve JCT-optimality for analytic jobs. In fact, our step dependency model draws inspiration from iterator models to identify regions of execution that can or cannot be overlapped with other regions. Unlike prior work, however, our approach also considers cost-optimality, a key concern in serverless analytics.



(a) Lazy (b) NIMBLE

Figure 17: **Q3C execution breakdown** for (a) lazy and (b) NIMBLE.

Caerus's scheduling problem is also related to the parallel query scheduling [58–60] in databases. Many of the proposed algorithms assign CPU and memory resources across operators considering both pipelinable and non-pipelinable dependencies across them. Unlike Caerus, however, these algorithms are designed for server-centric deployments and optimize for the query completion time under limited resource constraints.

Another related problem is Multi-Objective Query Optimization (MOQO), which searches for an query execution plan with an optimal trade-off between multiple conflicting cost metrics in databases [61–67]. While MOQO optimizes a query execution plan to determine its component set of operations and their orderings, Caerus takes the execution plan as input and specifically optimizes the task launch times for JCT and cost, i.e., Caerus approach is *complementary* to MOQO.

8 Conclusion

We have presented Caerus, a task scheduler for serverless analytics that uses a new NIMBLE scheduling algorithm. NIMBLE efficiently pipelines task executions across various stages in serverless analytics jobs, to ensure cost-optimality and Pareto-optimality between cost and JCT. We show that for a wide range of analytics workloads, NIMBLE is often optimal in *both* dimensions. This allows Caerus to outperform existing lazy scheduling approaches by 1.08–2.2× in JCT, and eager approaches by 1.21–1.57× in cost for these workloads.

Acknowledgments

We thank our anonymous reviewers and shepherd Srinivasan Seshan for their insightful comments. We also thank Qifan Pu, Zongheng Yang, Silvery Fu, Chenggang Wu, Danyang Zhuo, Joe Hellerstein and other members of RISELab for their constructive feedback. In addition to NSF CISE Expeditions Award CCF-1730628, this research is supported by gifts from Amazon Web Services, Ant Group, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware.

References

[1] AWS Lamda. https://aws.amazon.com/lambda/.

[2] Google Cloud Functions. https://cloud.google.com/functions.

[3] Azure Functions. https://azure.microsoft.com/en-us/services/functions.

[4] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, 2019.

[5] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[6] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Communications of the ACM*, 2019.

[7] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *NSDI*, 2017.

[8] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: distributed computing for the 99%. In *SoCC*, 2017.

[9] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *SoCC*, 2020.

[10] Youngbin Kim and Jimmy Lin. Serverless data analytics with Flint. In *CLOUD*, 2018.

[11] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *ATC*, 2019.

[12] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud function services. In *SIGMOD*, 2020.

[13] Qubole Announces Apache Spark on AWS Lambda. https://www.qubole.com/blog/spark-on-aws-lambda.

[14] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: scalable analytics on serverless infrastructure. In *NSDI*, 2019.

[15] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *OSDI*, 2018.

[16] Databricks Serverless: Next Generation Resource Management for Apache Spark. https://bit.ly/3cbLe3L.

[17] Amazon. Amazon Aurora Serverless. https://aws.amazon.com/rds/aurora/serverless.

[18] Azure. Azure SQL Data Warehouse. https://azure.microsoft.com/en-us/services/sql-data-warehouse.

[19] Charles Reiss. *Understanding Memory Configurations for In-Memory Analytics*. PhD thesis, EECS Department, University of California, Berkeley, 2016.

[20] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient Memory Disaggregation with Infiniswap. In *NSDI*, 2017.

[21] Amazon S3. https://aws.amazon.com/s3.

[22] Introduction to object storage in Azure. https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction.

[23] Google Cloud Storage. https://cloud.google.com/storage/.

[24] Jiffy: A virtual memory abstraction for serverless architectures. https://github.com/resource-disaggregation/jiffy.

[25] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[26] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *SIGCOMM CCR*, 2014.

[27] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *OSDI*, 2016.

[28] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, 2016.

[29] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *NSDI*, 2010.

[30] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.

[31] Spark SQL. https://spark.apache.org/sql/.

[32] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *SIGOPS*, 2007.

[33] Apache Hive. https://hive.apache.org.

[34] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 2008.

[35] Cloudera Impala. http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html.

[36] Scheller Brandon. Best practices for resizing and automatic scaling in Amazon EMR. https://amzn.to/2ZJYY0D, 2018.

[37] Hadoop Distributed File System. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[38] AWS Lambda pricing. https://aws.amazon.com/lambda/pricing/.

[39] Amazon S3 pricing. https://aws.amazon.com/s3/pricing/.

[40] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, 2010.

[41] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. A study of skew in mapreduce applications. *Open Cirrus Summit*, 2011.

[42] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: mitigating skew in mapreduce applications. In *SIGMOD*, 2012.

[43] Cliff Engle, Antonio Lupher, Reynold Xin, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Fast Data Analysis Using Coarse-grained Distributed Memory. In *SIGMOD*, 2012.

[44] Goetz Graefe. Volcano, an extensible and parallel query evaluation system; cu-cs-481-90. 1990.

[45] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. 2011.

[46] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. Hug: Multi-resource fairness for correlated and elastic demands. In *NSDI*, 2016.

[47] Sameer Agarwal, Srikanth Kandula, Nico Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Reoptimizing data parallel computing. In *NSDI*, 2012.

[48] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[49] Owen O'Malley. Terabyte sort on apache hadoop. 2008.

[50] gensort Data Generator. http://www.ordinal.com/gensort.html.

[51] TPC-DS. http://www.tpc.org/tpcds/.

[52] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *ATC*, 2018.

[53] AMPLab. The BigData Benchmark. https://amplab.cs.berkeley.edu/benchmark/, 2018.

[54] Luc Bouganim, Daniela Florescu, and Patrick Valduriez. Dynamic load balancing in hierarchical parallel database systems. 1996.

[55] Li Wang, Minqi Zhou, Zhenjie Zhang, Yin Yang, Aoying Zhou, and Dina Bitton. Elastic pipelining in an in-memory database cluster. In *SIGMOD*, 2016.

[56] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *SIGMOD*, 2014.

[57] Yuan Yu Michael Isard Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, and Pradeep Kumar Gunda Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. *Proc. LSDS-IR*, 2009.

[58] Minos N Garofalakis and Yannis E Ioannidis. Multi-dimensional resource scheduling for parallel queries. *ACM SIGMOD Record*, 1996.

[59] Chandra Chekuri, Waqar Hasan, and Rajeev Motwani. Scheduling problems in parallel query optimization. In *SIGACT-SIGMOD-SIGART*, 1995.

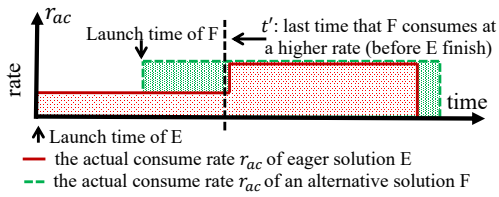


Figure 18: [Example for Lemma A.1] The actual consume rate of (i) the eager approach $\mathcal{E}$; and (ii) an alternate approach $\mathcal{F}$ with a later launch time.

[60] Minos N Garofalakis and Yannis E Ioannidis. Parallel query scheduling and optimization with time-and space-shared resources. *SORT*, 1997.

[61] Immanuel Trummer and Christoph Koch. Approximation schemes for many-objective query optimization. In *SIGMOD*, 2014.

[62] Immanuel Trummer and Christoph Koch. An incremental anytime algorithm for multi-objective query optimization. In *SIGMOD*, 2015.

[63] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. In *SIGMOD*, 1992.

[64] Sameer Agarwal, Anand P Iyer, Aurojit Panda, Samuel Madden, Barzan Mozafari, and Ion Stoica. Blink and it's done: interactive queries on very large data. 2012.

[65] Immanuel Trummer and Christoph Koch. Multi-objective parametric query optimization. *ACM SIGMOD Record*, 2016.

[66] Christos H Papadimitriou and Mihalis Yannakakis. Multiobjective query optimization. In *SIGMOD-SIGACT-SIGART*, 2001.

[67] Amol Deshpande and Lisa Hellerstein. Parallel pipelined filter ordering with precedence constraints. *TALG*, 2012.

A Theoretical Proofs

Lemma A.1 *The eager approach always optimizes the finish time for a reduce task.*

Proof Figure 18 shows an example of the actual consume rate $r_{ac}(t)$ for `r.s1` under two approaches: the eager solution $\mathcal{E}$ and an alternate approach $\mathcal{F}$ with a later launch time. We see that before $\mathcal{E}$ finishes, there may exist some time t such that $r_{ac}(t)$ of $\mathcal{F}$ is greater than $r_{ac}(t)$ of $\mathcal{E}$. Based on Equation 1, we know that this is because $\mathcal{E}$ has processed all its inputs by t ($S(t) = 0$ for $\mathcal{E}$), but $\mathcal{F}$ has not ($S(t) = 1$ for F) as it is launched later. Denote the last such time point before $\mathcal{E}$ finishes as t', we have: (1) by time t', $\mathcal{E}$ has processed all data generated before t'; (2) after time t', $r_{ac}(t)$ of $\mathcal{E}$ is no less than $\mathcal{F}$ until it finishes. The combination of these two observations shows that $\mathcal{E}$ always has an earlier finish time for `r.s1` than $\mathcal{F}$. Since $T_e^* = t_{e,\mathtt{s1}}^* + d_{\mathtt{s2}}^*$, $\mathcal{E}$ also ensures optimal task finish time. ■

Proof of Theorem 4.2 :

Theorem 4.2 *For a task in an analytics job with an arbitrary execution DAG, given the execution (produce rate) of all its parent steps, we can always achieve both optimal execution duration and finish time by launching it at $T_s^* = T_e^* - D^*$, where T_e^* is the optimal finish time and D^* is the optimal duration computed using Steps 1 and 2 described in §4.2.1.*

Proof We prove Theorem 4.2 by mathematical induction on the number of steps of the task.

Base case: We first show that Theorem 4.2 holds for a task with only one step. In this case, the problem reduces to the single parent-child step case for two-stage map-reduce jobs as in Theorem 4.1, which we have already shown to hold.

Inductive step: We now show that for any $n > 1$, if Theorem 4.2 holds for tasks with n steps, it also holds for tasks with $n+1$ steps. Consider a task with $n+1$ steps. For the first n steps, we denote the optimal finish time as $T_e^*(n)$, and cost as $D^*(n)$. Since Theorem 4.2 holds for any task with n steps, $T_e^*(n)$ and $D^*(n)$ can be achieved simultaneously by launching the $(n+1)$-step task at $T_s^*(n) = T_e^*(n) - D^*(n)$. Based on Equation 2 and the definition of the optimal task duration (Step 1 in §4.2.1), we have:

$$\begin{aligned} T_e^*(n+1) &= max(t'_{e,\mathtt{s}n+1}, T_s^*(n) + d_{\mathtt{s}n+1}^*) \\ D^*(n+1) &= D^*(n) + d_{\mathtt{s}n+1}^* \end{aligned} \tag{3}$$

Based on Equation 3, the launch time calculated from Theorem 4.2 for the $(n+1)$-step task is

$$\begin{aligned} T_s^*(n+1) &= T_e^*(n+1) - D^*(n+1) \\ &\geq (T_s^*(n) + d_{\mathtt{s}n+1}^*) - (D^*(n) + d_{\mathtt{s}n+1}^*) \\ &= T_e^*(n) - D^*(n) \\ &= T_s^*(n) \end{aligned} \tag{4}$$

Note that if a step j starts at time t_1 and executes at full load (i.e., it will never stall for data to become available), then it must also be able to execute at full load if it starts at any time $t_2 \geq t_1$. Since we have $T_s^*(n+1) \geq T_s^*(n)$ from Equation 4, if we launch the task at $T_s^*(n+1)$, we can always execute the first n steps at full load (i.e., with optimal duration $D^*(n)$). As such, with launch time $T_s^*(n+1)$, the corresponding finish time of the first n steps is $T_s^*(n+1) + D^*(n)$, which is also the start time of the last step $\mathtt{s}n+1$. Based on Equation 3 we have:

$$\begin{aligned} \text{Start time of step } \mathtt{s}n+1 &= T_s^*(n+1) + D^*(n) \\ &= (T_e^*(n+1) - D^*(n+1)) + D^*(n) \\ &= T_e^*(n+1) - d_{\mathtt{s}n+1}^* \\ &= max(t'_{e,\mathtt{s}n+1}, T_s^*(n) + d_{\mathtt{s}n+1}^*) - d_{\mathtt{s}n+1}^* \\ &\geq t'_{e,\mathtt{s}n+1} - d_{\mathtt{s}n+1}^* \end{aligned} \tag{5}$$

Equation 5 indicates that if we launch the task at $T_s^*(n+1)$, the start time of the step $\mathtt{s}n+1$ is no less than $t'_{e,\mathtt{s}n+1} - d_{\mathtt{s}n+1}^*$. Note

that $t'_{e,\mathsf{s}n+1}$ is the optimal finish time of step $\mathsf{s}n+1$ calculated only based on its parent. Just as in the proof of Theorem 4.1, which covers the single parent-child step pair case, we can then show that the step $\mathsf{s}n+1$ must execute at full rate if it is launched at $t'_{e,\mathsf{s}n+1} - d^*_{\mathsf{s}n+1}$.

Taken together, if we launch the task at $T^*_s(n+1)$, all $n+1$ steps can execute at full load, which indicates it achieves the optimal duration $D^*(n+1)$. Moreover, recall that $T^*_s(n+1) = T^*_e(n+1) - D^*(n+1)$. This means that if the task starts at $T^*_s(n+1)$ and has a duration of $D^*(n+1)$, it must finish at $T^*_e(n+1)$. As such, we can achieve both $T^*_e(n+1)$ and $D^*(n+1)$ by launching the task at $T^*_s(n+1)$.

Conclusion: Since both the base case and the inductive step hold, Theorem 4.2 holds by mathematical induction. ■