Clockwork: A Delay-Based Global Scheduling Framework for More Consistent Landing Times in the Data Warehouse

Martin Valdez-Vivas
Facebook
mvv@fb.com

Varun Sharma
Facebook
vasharma@fb.com

Nick Stanisha
Facebook
nickstanisha@fb.com

Shan Li
Facebook
shanharwayne@fb.com

Luo Mi
Facebook
miluo@fb.com

Wei Jiang
Facebook
jiangwei@fb.com

Alex Kalinin
Facebook
aykalinin@fb.com

Josh Metzler
Facebook
joshm@fb.com

ABSTRACT

Recurring batch data pipelines are a staple of the modern enterprise-scale data warehouse. As a data warehouse scales to support more products and services, a growing number of interdependent pipelines running at various cadences can give rise to periodic resource bottlenecks for the cluster. This resource contention results in pipelines starting at unpredictable times each day and consequently variable landing times for the data artifacts they produce. The variability gets compounded by the dependency structure of the workload, and the resulting unpredictability can disrupt the project workstreams which consume this data. We present Clockwork, a delay-based global scheduling framework for data pipelines which improves landing time stability by spreading out tasks throughout the day. Whereas most scheduling algorithms optimize for makespan or average job completion times, Clockwork's execution plan optimizes for stability in task completion times while also targeting efined pipeline SLOs. We present this new problem formulation and design a list scheduling algorithm based on its analytic properties. We also discuss how we estimate the resource requirements for our recurring pipelines, and the architecture for integrating Clockwork with Dataswarm, Facebook's existing data workflow management service. Online experiments comparing this novel scheduling algorithm and a previously proposed greedy procrastinating heuristic show tasks complete almost an hour earlier on average, while exhibiting lower landing time variance and producing significantly less competition for resources in the cluster.

CCS CONCEPTS

• **Software and its engineering** → **Distributed systems organizing principles**; Abstraction, modeling and modularity; **Scheduling**; • **Theory of computation** → **Scheduling algorithms**; *Theory and algorithms for application domains.*

KDD '21, August 14–18, 2021, Virtual Event, Singapore.

ACM ISBN 978-1-4503-8332-5/21/08.
https://doi.org/10.1145/3447548.3467119

KEYWORDS

data pipeline scheduling; delay-based scheduling; global cluster scheduling; data warehouse management; completion time stability; systems data science

ACM Reference Format:
Martin Valdez-Vivas, Varun Sharma, Nick Stanisha, Shan Li, Luo Mi, Wei Jiang, Alex Kalinin, and Josh Metzler. 2021. Clockwork: A Delay-Based Global Scheduling Framework for More Consistent Landing Times in the Data Warehouse. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '21), August 14–18, 2021, Virtual Event, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3447548.3467119

1 INTRODUCTION

Data is used by almost all internet applications. Many apps are backed by machine learning models trained on millions to billions of training examples, and the new wave of big data analytics-driven business intelligence is a thriving industry. As a large social media platform with a global presence, Facebook processes data on an immense scale, in the previous decade its data warehouse that has grown in size from hundreds of petabytes [22] to several exabytes. While this growth has unlocked innumerable new opportunities for innovation, it comes with an ever-increasing demand on computing resources, which if not managed properly can cause strain on Facebook's infrastrucutre.

At a high level, data arrives to Facebook's data warehouse in few different ways: raw ingestion, streaming ETL (Extract, Transform, and Load) apps, and batch scheduled ETL pipelines, with pipelines representing the bulk of the computational workload. Data pipelines are executed by Dataswarm, which is Facebook's internal data workflow automation and scheduling platform [21]. Dataswarm is centered around coordinating a directed acyclic graph (DAG) of tasks, with a pipeline being defining as a set of tasks and their dependencies (which may be either on intra-pipeline tasks or on data artifacts produced by other pipelines). Other relevant features of Dataswarm is that it allows users to define tasks for different data processing engines, as well as the flexibility to define periodic jobs (daily, hourly, weekly, etc.). An open-sourced variant which is very similar to Dataswarm is Apache Airflow [2].

The growth of Facebook's data warehouse has raised new scheduling challenges for Dataswarm. Historically, Dataswarm has used a very straightforward dispatch mechanism—kicking off a task as soon as its upstream dependencies are satisfied. In practice this leads to launching a significant volume of equal-priority pipelines at periodic times of the day (principally around midnight), and with the number of pipelines now growing well into the tens of thousands this has invariably given rise to demand spikes and resource bottlenecks. Due to this resource contention, tasks may need to wait in queue for resources to become available, and the queueing delay for a given task can vary significantly across days depending on variable start and landing times for tasks occurring upstream. These factors introduce an element of randomness in the actual starting time for the given task, and subsequently in the landing times of the Hive tables and other data artifacts these tasks produce, which affects the landing times of tasks further downstream, and so on. In other words, the randomness in landing times gets compounded by the DAG structure of the overall Dataswarm workload, with downstream tasks being affected not just by the variance in their own queueing delays and execution times, but also by the variance in the landing times of the tasks they depend on upstream.

Several processes at Facebook revolve around when data from pipelines are expected to land, and variance in these landing times causes disruption resulting possibly even in lost revenue and fines. In fact, stakeholders with time-sensitive service-level objectives (SLOs) routinely indicate a preference for *stable* landing times over *earlier* average landing times in internal surveys, suggesting that in practice a highly stable scheduler that reliably lands data by the same time every day is more desireable than other objectives which traditionally have dominated the scheduling literature such as average job completion time and makespan. In this paper we introduce Clockwork, a new global scheduling framework for recurring data pipelines designed to meet data SLOs while optimizing for landing time stability. In short, Clockwork aims to accomplish this by generating a global plan which strategically spreads its workload throughout the day using delayed task dispatch times, thereby tempering demand spikes which are the main root cause for resource contention. Additionally, we observed more stable resource utilization in the cluster throughout the day, as a desireable byproduct of our dependency-aware variance reduction strategy.

We frame Clockwork in the context of the broader scheduling literature in section 2. We provide an overview of the relevant concepts in Dataswarm and Facebook's data warehouse to define the scope for Clockwork, and present a formal statement of our mathematical optimization problem in section 3. Resource and runtime estimation for data pipelines as well as characterizing the global dependency structure for the Dataswarm DAG presents its own domain of interesting challenges, and we discuss how we approached these for Clockwork in section 4. In section 5, we perform analysis of our optimization problem and discuss how we leveraged these insights in Clockwork's scheduling algorithm. In section 6, we detail how we designed the Clockwork planner as a service and integrated it with Dataswarm. We evaluated Clockwork in live, controlled experiments on subsets of Dataswarm's regularly-scheduled pipelines, comparing the new algorithm's performance against the status quo as well as a greedy procrastinating heuristic which was prevoiusly proposed in [5], and these results are presented in section 7.

2 RELATED WORK

There is an extensive literature on graph scheduling algorithms. Graph scheduling problems traditionally optimize for makespan (i.e., the difference between the start and end time for the entire workload) or average job completion time, both of which are typically cast as a mixed integer linear program (MILP) [4, 16]. As is true for combinatorial optimization problems in general, MILPs tend to be solved primarily through various approximation methods or heuristic solutions, which has been a running theme through a growing body of applied work on scheduling for pipelines [1, 3, 12].

Clockwork is conceptually related to reservation-based scheduling [5]. In this framework tasks are guaranteed a set amount of resources at a specific time based on a declarative language framework which also allows users to specify a dependency structure and SLOs. The authors formulate the scheduling problem as a MILP and propose a greedy procrastinating heuristic, effectively placing the reservation as close to the deadline as possible. This is shown to perform reasonably well in meeting task SLOs while maintaining high cluster utilization and low latency for best effort jobs. Similarly, Morpheus [11] uses the same declarative language with an explicit goal to increase workload predictability and average cluster utilization. They propose a bin-packing heuristic to place periodic tasks with SLOs that are inferred from historical data.

More broadly, Clockwork can be compared to work on budget-constrained workflow scheduling with fixed deadlines, otherwise known as Quality of Service (QoS)-constrained workflow scheduling [15]. A persistent challenge in this class of problems—in particular for the online setting—is framing and modeling the stochastic nature of task runtimes, resource usage, or both [9, 10, 15]. Another frequent source of complexity in this type of scheduling is handling a tradeoff across multiple resource types [6, 7]. As we discuss in the following section, Clockwork largely sidesteps the latter by considering each underlying data processing engine separately and leveraging an understanding of its bottlenecked resource for scheduling.

Deep reinforcement learning has been advanced as an alternative proposal for workflow scheduling [13, 14]. One complication with this approach is the curse of dimensionality (i.e., performing policy iteration over a combinatorial state space), however Decima [14] addressed this through the use of a graph neural network which encodes the state information in a set of embedding vectors. Moreover, Decima demonstrated improvements in average job completion time compared to established heuristic-based schedulers including Graphene [7] and Tetris [6]. On the other hand, Decima leans on a tradeoff between performance and interpretability which is currently in vogue within the deep learning community [8, 17]. Especially for operational systems running at scale, pragmatic consideration such as the ability to perform effective triage and root-cause analyses usually favor methods whose outputs are easier to process and debug in practice.

3 CONCEPTS AND DEFINITIONS

We offer a brief review of the core concepts in workflow scheduling, and use them to formally define Clockwork's scheduling problem.

3.1 Data pipelines

A Dataswarm pipeline is comprised of the following elements:

- A *query* is a command to read from or write to a database, and is executed by a query engine. At Facebook, the two most commonly used query engines are Presto [23] and Spark [24]. There are other query engines that can be accessed through Dataswarm, but these technically execute outside of the data warehouse.
- A *task* represents an atomic unit of work. A task can be defined with a set of dependencies on other tasks or data artifacts yielded by other pipelines, effectively permitting inter-pipeline dependencies.
- An *operator* is an interface for defining a task. An operator is analogous to a function within a program. A Spark or Presto operator usually launches a single query, and produces a single table or table partition. There are different operators for submitting queries to different query engines, as well as custom operators to launch other processes like Python scripts or Bash commands.
- A *pipeline* is a directed acyclic graph (DAG) of tasks. An individual pipeline may produce one table or several tables, and frequently invokes a hetereogenous set of operators. A schedule for how often the tasks should run (daily, hourly, weekly, etc.) is usually defined at the pipeline level, though it is possible to set individual tasks to run at a different cadences. For example, a task which trains a model may only need to run weekly, while an evaluation task for the model is set to run daily.

3.2 Query and cluster taxonomy

Something to note about Dataswarm is that it technically does not execute queries, it only dispatches these jobs to different query engines running on separate clusters (as an aside, this is why true reservation-based scheduling is not possible in our current setup—Dataswarm controls when tasks are dispatched but does not manage the underlying capacity). From a resource planning perspective, it is important therefore for Clockwork to incorporate a high-level awareness of how different distributed query engines allocate queries and manage their cluster capacity. This behavior is slightly different for Presto and Spark, which are the relevant components to Facebook's data warehouse planning as explained above.

In Spark, an individual query gets broken down into contiguous stages, with each stage requiring a different number of containers. The number of containers for Spark queries is almost always determined by the amount of *memory* that is required at each stage. In order to estimate the resource requirements of a Spark query for planning, we compute a "skyline" for each query, describing its resource usage over time based on data from historical runs. The details of our approach are covered in section 4.

Resource accounting for Presto is slightly simpler, Presto scheduling is premised on a hierarchical queueing structure with each queue in the hierarchy capped at some number of concurrent queries. Internally, Presto does more fine-grained allocation of worker threads which is scaled dynamically based on the current cluster utilization, however we chose to abstract away this level of complexity for Clockwork's budget tracking purposes.

3.3 Mathematical formulation

We frame our formulation of Clockwork's optimization problem using Spark tasks, the generalization to include Presto tasks should be straightforward. As discussed in section 3.2, the resource requirements for a Spark task can be characterized using a skyline, with the skyline for task i denoted as

$$S_i = [(\Delta_i^1, m_i^1), \ldots, (\Delta_i^M, m_i^M)]$$

where (Δ_i^j, m_i^j) specifies one stage of random duration Δ_i^j requiring m_i^j units of memory. Let c denote the total capacity available in the cluster (in units of memory for Spark and units of queries for Presto, see section 3.2 for more details) and d_i denote the absolute, fixed-time deadline or SLO for the task.

The decision variable for Dataswarm can be expressed as a vector $a = (a_1, \ldots, a_N)$, where a_i is the dispatch time for task i. We also define a random variable τ_i as the actual starting time of the task, where

$$\tau_i = \max\left(a_i, \max_{k \in \mathcal{U}_i}(\tau_k + \sum_j \Delta_k^j)\right)$$

using $\mathcal{U}_i$ to indicate the set of tasks upstream of task i and $(\tau_k + \sum_j \Delta_k^j)$ corresponds to the landing time for task k. In other words, task i will start running at its dispatch time or the time at which all its upstream dependencies are satisfied, whichever is later.

Clockwork's objective is to simultaneously minimize the variance in task landing times as well as the penalties for missing their SLOs. By aiming to reduce an expected cost and variance, the formulation resembles the conditional value-at-risk scheduling problem [18, 19]. The objective function can be formalized as

$$\min_a \sum_i \mathbb{E}(\tau_i + \sum_j \Delta_i^j - d_i)^+ + \frac{\beta}{2} \sum_i \mathrm{Var}(\tau_i + \sum_j \Delta_i^j)$$

with $\mathbb{E}(\cdot)^+$ is the expected value taken over the positive part function, $\mathrm{Var}(\cdot)$ is the variance, and $\beta/2$ is a weighting factor (which can alternatively be defined for each task separately). This function is to be minimized subject to nonnegativity and cluster capacity constraints, which we include here for completeness introducing $\mathbf{1}(\cdot)$ as notation for the indicator function and $[0, T]$ for the planning time horizon:

$$\tau_i \geq a_i, \ i = 1, \ldots, N$$

$$\tau_i \geq \tau_k + \sum_j \Delta_k^j, \ \forall k \in \mathcal{U}_i$$

$$\sum_{i,j} m_i^j \mathbf{1}(\tau_i + \sum_{\ell < j} \Delta_i^\ell \leq t \leq \tau_i + \sum_{\ell \leq j} \Delta_i^\ell) \leq c, \ \forall t \in [0, T]$$

$$a \geq 0$$

4 RESOURCE ESTIMATION AND DEPENDENCY DATA

One practical challenge in workflow scheduling is estimating task durations and resource requirements, which are generally unknown a priori [9, 10]. We provide some detail of our solutions on these fronts, as well as other practical design considerations for the optimization problem defined in section 3.3 such as defining task deadlines and processing a global DAG representation at scale.

4.1 Task metrics and deadlines

Recurring data pipelines provide some advantage over scheduling for adhoc workloads in that it is possible to mine rich telemetry data from their run history. Some metrics that we collected from historical data for scheduling and evaluation purposes included:

- *Task state transition timestamps* such as the start and end times for each task, in addition to ready time–which is the time at which all of the upstream dependencies for a task have been satisfied. These data can be used to derive task duration and queueing time.
- *Query execution times and memory usage* which are obtained from query engine logs. See section 4.3 for more details on our approach for estimating resource usage for Spark.

The distributions for the above quantities were approximated for each task by collecting quantiles over a 4-week aggregation window.

For defining task SLOs, we relied on a custom metric for Facebook's data warehouse called healthy landing time, which is assigned to each Dataswarm task. The healthy landing time for a task is calculated by taking the 90th percentile of a median-filtered and winsorized sequence of its end times over a specified time period.

4.2 Global dependency structure

Global scheduling requires global awareness of precedence constraints, and maintaining an accurate representation of the dependency structure for Dataswarm presents another challenge especially given its scale, which stands at over 1 million tasks. Dataswarm itself does not have awareness of its global task dependency structure. As discussed in section 3.1, Dataswarm allows users to specify dependencies on other data artifacts, which may be produced by other pipelines, but the origin of the data is not visible to Dataswarm.

We obtained the global task dependency structure for Dataswarm by crawling a separate, internal data lineage tool which tracks metadata and stdout logs from clients to continually audit data flows between all data warehouse endpoints (Dataswarm tasks runs, adhoc queries, Hive table partitions, etc.). In the process, we collapsed intermediate nodes between task runs and dropped tasks which are only set to wait for data artifacts, resulting in a complete precedence ordering of the workload-bearing tasks in Dataswarm.

Because of its large size and complexity, performing a full crawl of the data lineage through the provided APIs lasted over 36 hours, making it infeasible under this approach to build a new precedence ordering for scheduling on a daily basis. Instead, recognizing the fact that the precedence ordering is expected to change marginally day-over-day for recurring pipelines, we cache the precedence graph from the previous day and re-crawl lineages only for those tasks which are known to have been edited, added, or deleted. This brings the processing time to within 1 hour.

4.3 Resource estimation for queries

As alluded to in section 3.2, one challenge pertaining to Spark queries in particular is characterizing their resource requirements in terms of memory units over time, corresponding to the input variable S_i in section 3.3. We are referring to this quantity as a *skyline*. The skyline is difficult to estimate compared to other query telemetry metrics cited in section 4.1 because the observed resource usage over time for a single run of a repeated Spark query is determined not only by the complexity of the query, but also the contention in the Spark cluster at the time. For example, if the stage of a Spark query requests 1000 containers and only 10 are available, it will start running with the available containers and accumulate the remaining containers as they are released by other queries. Consequently, both the observed resource usage over time and total duration of the query can vary significantly day-over-day based on the other workload running in the Spark cluster at the time. For global planning purposes, our goal is to take noisy observations of previous query runs and recover the signal of the skyline in the absence of resource bottlenecks—which is how we expect it run under a schedule produced by Clockwork.

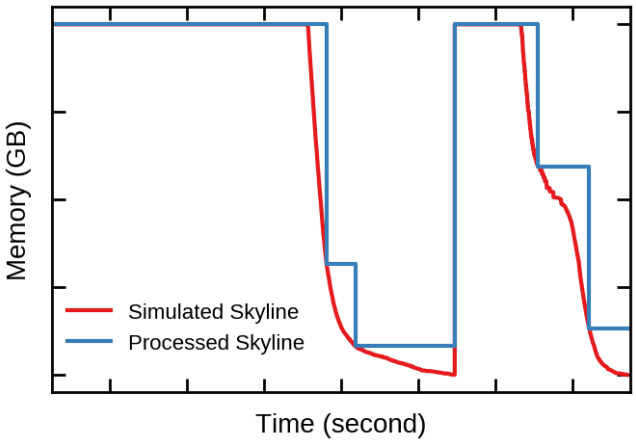

Figure 1: A sample skyline showing the simulation-based expected resource usage over time for a two-stage Spark query. Skylines are simulated across several days and aggregated to their outer contour, which is passed as an input to our scheduling algorithm for capacity planning.

The solution we designed is based on discrete-event simulation. From the query metadata, we obtained the number of concurrent and non-concurrent stages, the number of data partitions processed at each stage and the distribution of their durations, and from the Spark engine configuration we obtained the memory per worker and limit on the number of workers per stage. Each set of concurrent Spark stages was simulated with a separate shared-resource model, where the number of shared resources was set at the per-query worker limit and the individual events were the execution duration of each data partition generated from an empirical CDF. The skyline for each stage were conjoined to form the skyline of the query, and these were further aggregated by taking the outer contour of the query-level skylines over a historical time interval. The final aggregated skyline rendered a sufficiently conservative estimate of the task's resource requirements, and was used to monitor the cumulative cluster usage in our scheduling routine. A sample Spark skyline is shown in figure 1.

5 SCHEDULING ALGORITHM

To our knowledge, the problem defined in section 3.3 represents the first instance of a stochastic graph scheduling problem which explicitly prioritizes minimizing the variance of task completion

times. Morevoer, this formulation exhibits some compelling structure which lends intuition for suggesting an effective planning heuristic.

5.1 Analytic properties of the objective function

For notational convenienece, we introduce $\Delta_i = \sum_j \Delta_i^j$ as shorthand for the cumulative duration of the stages for task i.

Claim 1. *Let $X_i = \max_{k \in \mathcal{U}_i}(\tau_k + \Delta_k)$ represent the completion time of all the predecessors to task i, and further assume the duration of all the tasks are pairwise independent. Setting $\frac{\partial}{\partial a_i}\mathbb{E}(\tau_i + \Delta_i - d_i)^+ + \frac{\beta}{2} Var(\tau_i + \Delta_i) = 0$ yields the relation*

$$\mathbb{P}\big(\Delta_i > d_i - a_i\big) = \beta \int_{a_i}^{\infty} \mathbb{P}\big(X_i > t\big) dt$$

for all i with $|\mathcal{U}_i| > 0$.

Claim 2. *In the absence of resource constraints, the values $\hat{a}_1, \ldots, \hat{a}_N$ which satisfy the relation in claim 1 represent an upper bound on the optimal dispatch times $a_1^*, \ldots, a_N^*$.*

Claim 3. *Define $[0, T]$ as the planning horizon for the problem formulated in section 3.3. In the absence of resource constraints, if $\mathcal{U}_k = \emptyset$ and $\mathbb{P}\big(\Delta_k > \hat{a}_i\big) > 0$ for any i such that $k \in \mathcal{U}_i$, and $\hat{a}_i$ satisfies the relation in claim 1, then the optimal dispatch time for task k is $a_k^* = 0$.*

The proof of the above claims are provided in the supplemental material. Taken collectively, claims 1–3 suggest the optimal solution weighs a tradeoff between scheduling tasks early to fulfill their SLOs and scheduling tasks *distantly apart from their dependencies* to reduce the variance of the landing times. For instance, in laymen's terms, claim 3 states that if a task with no upstreams has downstream dependencies, and there is a nonzero probability of finishing past the upper bound dispatch time—as derived in claim 1—for any of its downstreams, the upstream task should be dispatched as early as possible so as to minimize the uncertainty in the effective start time of the downstream task. Likewise in the relation in claim 1, we clearly see this tradeoff: opting for an earlier dispatch time a_i *improves the likelihood of landing the task before its deadline* and decreases the quantity on the left hand side, but also *compresses it nearer to the stochasticity of its upstream dependencies* consequently increasing the value of the integral on the right hand side. Rephrasing it succinctly, it is desireable to dispatch jobs early, but not so early that the variance of its actual start time is heavily influenced by the upstream variance.

This analysis formalizes the intuition motivating the greedy procrastinating heuristic which has been proposed in other work [5], in which the authors suggest that scheduling task reservations close to their deadlines minimizes the likelihood the task will miss its reservation window due to upstream delays. Also, while very few formal results exist for the general graph scheduling problem with stochastic job processing times, the best theoretical performance guarantees that we are aware of are premised on delayed list scheduling, which similarly remedy the uncertainty in processing times by inserting delays between tasks [20]. As has been shown here, introducing landing time variance as an explicit term in the objective function makes delaying some tasks an even stronger imperative, and it is further likely that this novel reformulation of the stochastic graph scheduling problem carries interesting theoretical properties extending well beyond the initial claims cited above.

Algorithm 1 Clockwork scheduling algorithm

INPUT: planning horizon $[T_0, T_{\max}]$; cluster capacity C; set of tasks $\mathcal{T}$ with attributes: earliest allowable start time ℓ_i, deadline d_i, task duration CDF $F_{\Delta_i}(\cdot)$, resource skyline S_i; global precedence graph $G = (\mathcal{T}, E)$
REQUIRE: $d_i > \ell_i + F_{\Delta_i}^{-1}(0.5)$
OUTPUT: a global plan, represented as an indexed array of dispatch times $a = (a_1, \ldots, a_N)$ for all $i \in \mathcal{T}$

```
function ScheduleTasks(G)
    PriorityQueue Q = ∅, PriorityQueue B = ∅
    for all i ∈ ReverseTopologicalSorted(T) do
        Δ̃_i := F^{-1}_{Δ_i}(0.5), q_i := ∫_{Δ̃_i}^{∞} F_{Δ_i}(t)dt
        Σ_Δ(i) ← Δ̃_i, Σ_q(i) ← q_i, d(i) ← d_i
        φ(i) ← Σ_q(i) / (d(i) − ℓ_i − Σ_Δ(i))
        for all j ∈ children(i) do
            η ← (Σ_q(j) + q_i) / (d(j) − ℓ_i − (Σ_Δ(j) + Δ̃_i))
            if η < 0 or η = ∞ then
                B.addWithPriority(i, ℓ_i)
                Σ_Δ(i) ← 0, Σ_q(i) ← 0, d(i) ← ℓ_i
            else if η ≥ φ(i) and i ∉ B then
                φ(i) ← η, d(i) ← d(j)
                Σ_Δ(i) ← Σ_Δ(j) + Δ̃_i, Σ_q(i) ← Σ_q(j) + q_i
        if i ∉ B then Q.addWithPriority(i, φ(i))
    return PlaceRankedTasks(Q, B)
function PlaceRankedTasks(Q, B)
    Array a, Set A = ∅, GlobalPlanSkyline S
    a_v ← ℓ_v ∀v ∈ T
    while |Q| + |B| > 0 do
        if |B| > 0 then v ← B.pop() else v ← Q.pop()
        t ← S.findNearestPlacement(S_v, a_v, ℓ_v, d_v)
        a_v ← t, S.addSkylineAtTime(S_v, a_v), A.add(v)
        for all v' ∈ (children(v) ∩ A^c) do
            ℓ_v' ← max(ℓ_v', a_v + Δ̃_v)
            if d(v') − ℓ_v' − Σ_Δ(v') ≤ 0 then
                a_v' ← ℓ_v'
                Q.remove(v'), B.addWithPriority(v', ℓ_v')
            else
                w ← d(v') − a_v − Σ_Δ(v') − Δ̃_v
                a_v' ← max(a_v', a_v + Δ̃_v + w * q_v / (q_v + Σ_q(v')))
                η ← Σ_q(v') / (d(v') − a_v' − Σ_Δ(v'))
                if η > φ(v') then
                    Q.updatePriority(v', η)
        for all v' ∈ (parents(v) ∩ A^c) do
            d_v' ← min(d_v', a_v)
    return a
```

5.2 Implementation

Algorithm 1 shows the scheduling algorithm for Clockwork in psuedocode. Guided by the intution gained from the analysis in section

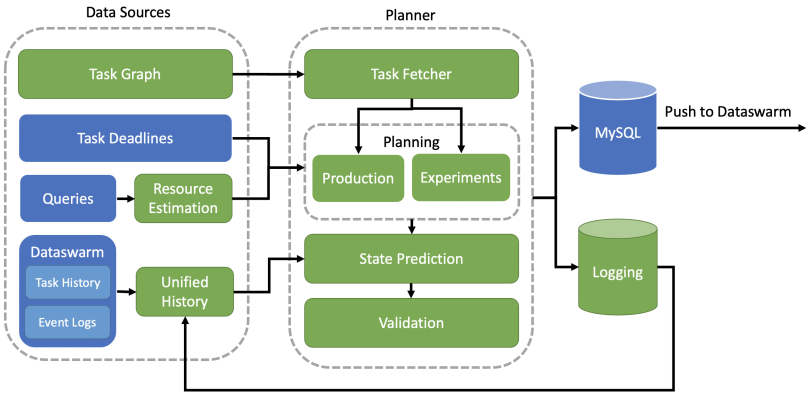

Figure 2: System design for the Clockwork planner service, and its integration with Dataswarm

5.1, at a high level the algorithm proceeds by a critical path-based scheduling approach which first ranks tasks based on a measure of their outgoing paths, and then spaces them proportionally between their upstreams and their deadline according to a measure of uncertainty in the task processing times. The measure of uncertainty we use is the area under the CDF curve above the median value, which resembles the right-hand quantity in claim 1 and is denoted as q_i in algorithm 1.

The task ranking step of the algorithm finds the longest path originating from each node, where the length of a path is defined as a ratio of the cumulative uncertainty measure along the path to *slack*. We define a path's slack here as the time remaining until the deadline of the last task if all tasks along the path run end-to-end with their median processing times. A path with less slack relative to its cumulative uncertainty is assigned higher priority. Intuitively, this leads to nodes along the paths with the most uncertainty packed within a fixed time budget to be scheduled first. If there is no slack along a path, the algorithm takes this as an indication that the tasks along the path should run best-effort, which is to say dispatched as soon as possible, and are given the highest priority. Tasks are scheduled in order of priority, and the amount of delay inserted after each task is in proportion to its contribution to the uncertainty measure along its outgoing paths. The time complexity of this ranking routine is $O\big((|\mathcal{T}| + |E|) \log |\mathcal{T}|\big)$.

Note we omit the implementation of $\mathcal{S}$ in algorithm 1 in the interest of brevity. The `GlobalPlanSkyline` is a bookkeeping construct to update the planned cluster usage and check for feasibility against the capacity constraint. Our implementation used discretized time increments and segment trees. The skyline values S_i which are inserted in $\mathcal{S}$ refer to the static quantities developed in section 4.3. With segment trees, `FindNearestPlacement` is a binary search, and `AddSkylineAtTime` is an $O(\log N)$ operation for each block of S_i, where N is the number of time increments in $\mathcal{S}$. Taken collectively, the complexity of all the skyline operations are $O\big((|\mathcal{T}| + B) \log N\big)$, with B defined as the total number of blocks across all task skylines.

6 DEPLOYMENT

We deploy the Clockwork planner as a stand-alone service which takes the task precedence graph, resource skylines, and deadlines as input. On every invocation, the planner defines a planning horizon relative to the current time and persists a global plan of dispatch times in a database. The planner itself has four main components

- *A data ingestion module* which fetches data for all tasks in the current planning horizon.
- *A configurable scheduling algorithm* which accepts the normalized data as input and computes a global plan based on the algorithm's specific objectives.
- *A ready state predictor module* which estimates the time at which we expect a given task to have all of its upstream dependencies satisfied (this time is specified as a task's *ready* time).
- *A validation and publishing module* which performs a series of quality checks on the planner output before publishing.

Dataswarm has its own scheduler service which, by default, dispatches tasks whenever they transition to the ready state. We integrate the Clockwork plan into the existing scheduler by adding a lightweight client to each worker (a single worker is responsible for managing a subset of tasks). Dispatch times are pre-fetched by the clients in the order determined by the ready state predictor. When a task is ready, the worker checks the client for a Clockwork-assigned dispatch time. If one exists, the task is enqueued for dispatching at that future time, otherwise it is dispatched immediately. With this prioritization and pre-fetching design, we are reliably able to execute all the intended Clockwork dispatch times while minimizing storage and network I/O in the existing Dataswarm scheduler.

7 EXPERIMENTS

We ran several initial trial runs with Clockwork by scheduling a subset of the Spark workload in a separate dedicated queue. Although this showed strongly positive results with respect to reducing both queueing delays and landing time variance, it is unclear the extent to which Clockwork benefitted from its isolation from the rest of the workload in the cluster. On the same token, we expect that a global scheduler like Clockwork will produce progressively better results relative to non-global heuristics when it is able to plan a larger share of the cluster workload, since otherwise the uncertainty posed by the landing times of any upstream tasks that are not planned by Clockwork will limit the variance reduction that is attainable.

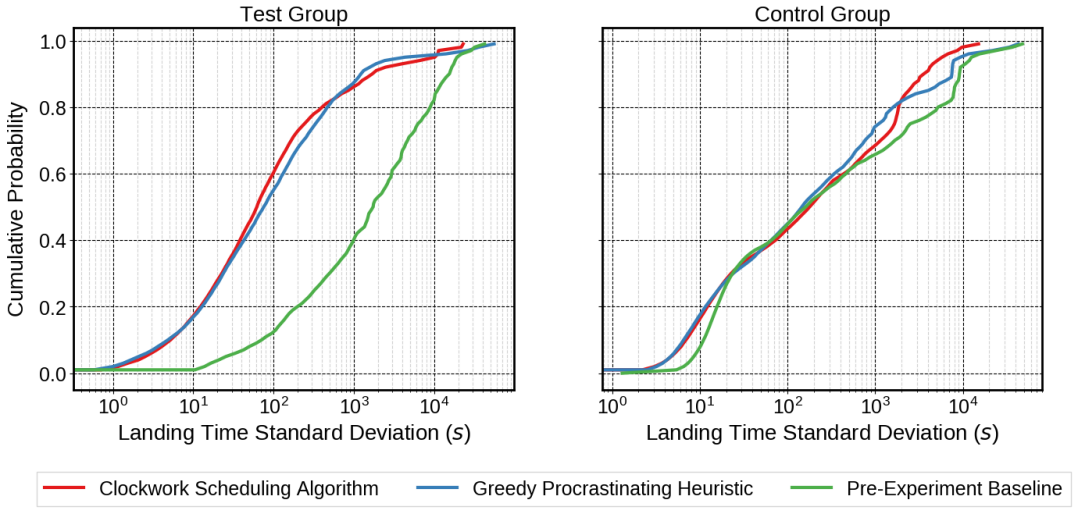

Figure 3: The cumulative distribution function of standard deviation in task landing times, shown for test and control groups. The distribution for the control group (right) remained constant through the three experimental phases, whereas the distribution for the test group (left) showed more stable landing times with our baseline greedy heuristic and Clockwork scheduling algorithm.

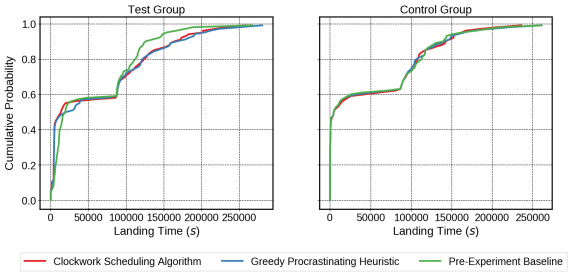

Figure 4: The cumulative distribution function of task landing times, relative to its generation time. The baseline is able to deliver earlier landing times in the aggregate compared to the other two approaches, at the cost of significantly less stability. Clockwork delivers earlier landing times in aggregate compared to the procrastinating heuristic, while offering the greatest predictability among the three approaches.

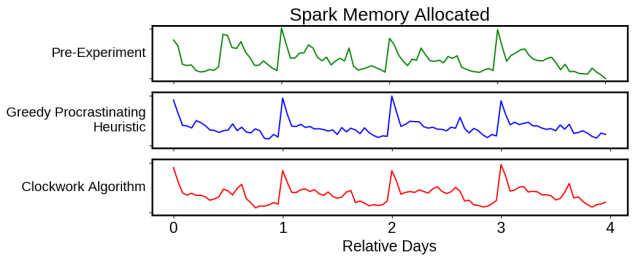

Figure 5: In addition to more stable landing times, Clockwork achieves lower resource contention and more stable and predictable cluster utilization. Looking at memory allocation for Spark (above), we see that the cluster tends to be used more evenly during off-peak times with Clockwork compared to the baseline cluster behavior.

For both of these reasons, the rollout plan for Clockwork relied on testing the algorithm in an online experiment over a sufficiently large pool of tasks, while also avoiding any modifications to our existing queue and cluster configurations to eliminate these changes as potential confounding factors. The test group was defined by selecting the existing Dataswarm workload in one of Facebook's data centers, which roughly maps to one Spark and one Presto cluster. In all, the test group amounted to tens of thousands of daily and hourly Dataswarm tasks. Dataswarm tasks are assigned to data centers in a way that minimizes interactions with tasks outside of the cluster, so even though cross-cluster dependencies are still a possibility, the workload in a data center approximately constitutes a self-contained DAG of tasks.

To confirm our observed effect sizes were not attributable to other widespread changes to the warehouse during our experiment time frame, we further defined a control group of proportional size in a different Facebook data center. In addition, we compared the performance of our proposed solution against a simple, greedy procrastinating heuristic from previous work [5]. The greedy heuristic schedules a task as close to its deadline as possible, and its relationship to Clockwork was touched on in sections 2 and 5. Both algorithms were run in different weeks within the same month.

Since we opted to run our evaluation on live production clusters, and each experiment needed to be run sequentially on the same workload for a minimum of 1 week to draw meaningful comparisons of the task-level standard deviations for landing times, this naturally constrained the number of experiments we could run outside of a simulated environment. In effect we traded breadth for scale and practical relevance. The central feature of the present scheduling problem also contributed to the complexity in interpreting observed effect sizes in our experiments, namely the tightly interconnected structure of the workload. Because the landing times in our experiments are tightly correlated, it is difficult to make any conclusive claims with regards to statistical significance.

	Average queueing time			
	SPARK		PRESTO	
Pre-Experiment	36.3s	–	74.9s	–
Greedy Heuristic	39.5s	+8.8%	86.3s	+15.2%
Clockwork Algorithm	24.3s	-30.8%	51.7s	-31.0%

Table 1: Average queueing time for the Spark and Presto queries in the test group. The Clockwork scheduling algorithm reduces queueing-based latency by about 30% based on reduced contention for cluster resources.

7.1 Landing times

The landing time of a task run was normalized relative to its period ID, a timestamp which specifies the task generation time and uniquely identifies a single task instance. We measured landing time variance for each task using a trimmed standard deviation, removing the min and max value over each 7 day window, as this metric proved to be more robust to outliers. The distribution of landing time variance over all the tasks in the test and control groups are shown in figure 3. In the test group, we observed progressively improved landing time variances in aggregate over the pre-experiment period using the greedy heuristic and Clockwork, respectively. Clockwork reduced the trimmed standard deviation in landing times by 72 minutes on average with an interquartile range of 100s and 5496s compared against the pre-experiment period, and 653s on average with an interquartile range of -65s and 88s compared to the greedy heuristic. The control group, on the other hand, showed no discerneable changes to landing time variances as expected.

As previously alluded to, variances in landing times are attributable to two sources: competition for resources in the cluster as well as uncertainty in the landing times of a task's upstream dependencies. The greedy heuristic by design exclusively attempts to attenuate the latter source of uncertainty by edging the task as close to its deadline as possible. Consequently, we observe Clockwork lands tasks earlier than the greedy heuristic in aggregate, by a median and mean of 3 and 56 minutes, respectively. Clockwork also landed tasks 40 minutes later on average than during the pre-experiment baseline as expected, though interestingly the median task landed 5 minutes *earlier* (56% of tasks had an earlier average landing time). The discrepancy suggests that while a fraction of the workload was postponed to later in the day, the capacity that was subsequently left free was used to move up the start times of tasks higher up in the DAG hierarchy. The daily SLO attainment rate for tasks in our workload largely remained at the baseline rate of about 97% throughout our experiment time frame.

7.2 Cluster utilization and contention

The case for global scheduling lies in its ability to better leverage cluster resources, in such a way to reduce contention between tasks. Queueing time—defined as the time a task spends waiting in queue for resources after its upstream dependencies have been satisfied—is used here as a proxy for resource contention. Clockwork reduced queuing delays significantly in both Spark and Presto, by approximately 30% compared to the pre-experiment window on average. Conversely, the procrastinating heuristic increased average queueing delays on the order of 10%. These results are summarized in table 1.

Furthermore, the cluster exhibited smoother resource utilization with both Clockwork and the greedy heuristic compared to our pre-experiment baseline. This effect is shown for Spark in figure 5. We measured the roughness of the hourly time series for allocated memory for Spark by taking the variance of the first differences, in other words $\mathrm{Var}(m_{t+1} - m_t)$. By this metric, Clockwork reduced roughness by 16.7% compared to the pre-experiment phase, while the greedy heurstic achieved a 13.1% reduction against the same baseline.

8 CONCLUDING REMARKS

In this paper, we present Clockwork, a dependency-aware scheduling framework for data pipelines that improves landing time stability with delayed dispatch times. Our solution is designed to address a novel formulation of the stochastic graph scheduling problem, and we demonstrate its effectiveness in large online experiments against a greedy procrastinating heuristic from previous work as well as the prior baseline performance with the default scheduling logic in Dataswarm. Clockwork is deployed as a standalone service and currently is coordinating tens of thousands of daily tasks as part of Facebook's data warehouse infrastructure. The Clockwork planner code has been released open-source at https://github.com/facebookresearch/Clockwork.

The algorithm presented in section 5 amounts to an offline scheduling algorithm. An interesting extension would be to consider the online variant. Especially under the objective of minimizing landing time variance, this direction becomes highly nontrivial, as it explores a tension between the imperative to stick to a somewhat static schedule while also adapting to unforeseen events which are commonplace in any data warehouse environment running at scale. This framing hints at a potentially rich adverserial learning problem.

Clockwork takes an agnostic view of task priorities, effectively assuming priority information is encoded in the task SLOs. An open question is whether and to what extent explicit user-defined priorities can improve both landing time stability and SLO attainment by themselves, and the implications for designing a complementary scheduling algorithm. Lastly, we observe that Clockwork produces more even resource utilization, which in itself is a desireable outcome from a cluster management perspective. Augmenting Clockwork to explicitly pursue higher average cluster utilization presents an interesting exercise. The main challenge derives from simultaneously weighing multiple objectives, namely balancing between competing priorities at the cluster and individual task levels.

ACKNOWLEDGMENTS

The authors gratefully acknowledge Sriram Rao for his guidance in the early phase of this project, and Nicolas Stier and our anonymous reviewers for their feedback on this submission. Joost de Nijs, Yudong Sun, and Vivek Vaidya provided analytical as well as project support for our experiments. We thank various members of Facebook's data warehouse who helped shape our thinking, including Sital Kedia, Mayank Garg, Wei Li, Claus Søgaard,

Yu Quan, Shashank Gupta, Delia David, Mahesh Somani, Richard Wareing, Daniel Peek, the Dataswarm team, and many others. We further thank our leadership team of Jigar Desai, Mat Oldham, David Clausen, Waqar Malik, Oytun Eskiyenenturk, Marc Light, Xin Fu, Maurizio Ferconi, Dario Benavides, and Ashish Kelkar for their feedback and encouragement.

REFERENCES

[1] Mainak Adhikari, Tarachand Amgoth, and Satish Narayana Srirama. 2019. A Survey on Scheduling Strategies for Workflows in Cloud Environment and Emerging Trends. *ACM Comput. Surv.* 52, 4, Article 68 (Aug. 2019), 36 pages. https://doi.org/10.1145/3325097

[2] Apache Software Foundation. 2020. *Airflow.* https://airflow.apache.org/

[3] Anne Benoit, Ümit V. Çatalyürek, Yves Robert, and Erik Saule. 2013. A Survey of Pipelined Workflow Scheduling: Models and Algorithms. *ACM Comput. Surv.* 45, 4, Article 50 (Aug. 2013), 36 pages. https://doi.org/10.1145/2501654.2501664

[4] Peter Brucker. 2010. *Scheduling Algorithms* (5th ed.). Springer Publishing Company, Incorporated.

[5] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. 2014. Reservation-Based Scheduling: If You're Late Don't Blame Us!. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2670979.2670981

[6] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-Resource Packing for Cluster Schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 455–466. https://doi.org/10.1145/2619239.2626334

[7] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. Graphene: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 81–97.

[8] David Hand. 2006. Classifier Technology and the Illusion of Progress. *Statist. Sci.* 21 (02 2006), 1–14. https://doi.org/10.1214/088342306000000060

[9] Alexey Ilyushkin and Dick Epema. 2018. The Impact of Task Runtime Estimate Accuracy on Scheduling Workloads of Workflows. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '18)*. IEEE Press, 331–341. https://doi.org/10.1109/CCGRID.2018.00048

[10] Alexey Ilyushkin, Bogdan Ghit, and Dick Epema. 2015. Scheduling Workloads of Workflows with Unknown Task Runtimes. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGRID '15)*. IEEE Press, 606–616. https://doi.org/10.1109/CCGrid.2015.27

[11] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 117–134.

[12] Georgia Kougka, Anastasios Gounaris, and Alkis Simitsis. 2018. The many faces of data-centric workflow optimization: a survey. *Int. J. Data Sci. Anal.* 6, 2 (2018), 81–107. https://doi.org/10.1007/s41060-018-0107-0

[13] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*. Association for Computing Machinery, New York, NY, USA, 50–56. https://doi.org/10.1145/3005745.3005750

[14] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 270–288. https://doi.org/10.1145/3341302.3342080

[15] Bilkisu L. Muhammad-Bello and Masayoshi Aritsugi. 2017. Robust Deadline-Constrained Resource Provisioning and Workflow Scheduling Algorithm for Handling Performance Uncertainty in IaaS Clouds. In *Companion Proceedings of The10th International Conference on Utility and Cloud Computing (UCC '17 Companion)*. Association for Computing Machinery, New York, NY, USA, 29–34. https://doi.org/10.1145/3147234.3148110

[16] Michael L. Pinedo. 2016. *Scheduling: Theory, Algorithms, and Systems* (5th ed.). Springer Publishing Company, Incorporated.

[17] Cynthia Rudin and Joanna Radin. 2019. Why Are We Using Black Box Models in AI When We Don't Need To? A Lesson From An Explainable AI Competition. *Harvard Data Science Review* 1, 2 (2019). https://doi.org/10.1162/99608f92.5a8a3a3d

[18] Subhash Sarin, Hanif Sherali, and Lingrui Liao. 2014. Minimizing conditional-value-at-risk for stochastic scheduling problems. *Journal of Scheduling* 17 (02 2014). https://doi.org/10.1007/s10951-013-0349-6

[19] Subhash C. Sarin, Balaji Nagarajan, and Lingrui Liao. 2010. *Stochastic Scheduling: Expectation-Variance Analysis of a Schedule.* Cambridge University Press. https://doi.org/10.1017/CBO9780511778032

[20] Martin Skutella and Marc Uetz. 2005. Stochastic machine scheduling with precedence constraints. *SIAM J. Comput.* 34, 4 (2005), 788–802.

[21] Mike Starr. 2014. Dataswarm. PyData Silicon Valley 2014. https://pyvideo.org/pydata-silicon-valley-2014/mike-starr-dataswarm.html.

[22] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. 2010. Data Warehousing and Analytics Infrastructure at Facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 1013—-1020. https://doi.org/10.1145/1807167.1807278

[23] Shivaram Venkataraman, Erik Bodzsar, Indrajit Roy, Alvin AuYoung, and Robert S. Schreiber. 2013. Presto: Distributed Machine Learning and Graph Processing with Sparse Matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 197–210. https://doi.org/10.1145/2465351.2465371

[24] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664

A PROOF OF CLAIMS IN SECTION 5.1

A.1 Proof for claim 1

The proof for claim 1 proceeds in 3 steps: we derive $\frac{\partial}{\partial a_i}\frac{\beta}{2}\text{Var}(\tau_i+\Delta_i)$ and $\frac{\partial}{\partial a_i}\mathbb{E}(\tau_i+\Delta_i-d_i)^+$ separately and combine these results. For notational simplicity, we introduce $X_i = \max_{k\in\mathcal{U}_i}(\tau_k+\Delta_k)$, in other words X_i is the latest completion time across all the upstreams for task i.

Claim 4. *Assuming task durations are pairwise independent, and the duration of a task is independent of its start time,*

$$\frac{\partial}{\partial a_i}\frac{\beta}{2}Var(\tau_i+\Delta_i) = \beta\left(\int_{a_i}^{\infty}\mathbb{P}(X_i>t)dt\right)\left(\mathbb{P}(X_i>a_i)-1\right)$$

Proof.

$$\frac{\partial}{\partial a_i}\frac{\beta}{2}\text{Var}(\tau_i+\Delta_i) = \frac{\beta}{2}\left(\frac{\partial}{\partial a_i}\text{Var}(\tau_i)+\frac{\partial}{\partial a_i}\text{Var}(\Delta_i)\right) = \frac{\beta}{2}\left(\frac{\partial}{\partial a_i}\text{Var}(\max(X_i,a_i))\right)$$

$$\frac{\partial}{\partial a_i}\text{Var}(\max(X_i,a_i)) = \frac{\partial}{\partial a_i}\left(\mathbb{E}\big(\max(X_i,a_i)^2\big)-\mathbb{E}\big(\max(X_i,a_i)\big)^2\right)$$

Since $a_i \geq 0$, the expectation $\mathbb{E}\big(\max(X_i,a_i)\big)$ can be written as

$$\begin{aligned}\mathbb{E}\big(\max(X_i,a_i)\big) &= \int_0^{\infty}\mathbb{P}\big(\max(X_i,a_i)>t\big)dt\\ &= \int_0^{a_i}\mathbb{P}\big(\max(X_i,a_i)>t\big)dt + \int_{a_i}^{\infty}\mathbb{P}\big(\max(X_i,a_i)>t\big)dt\\ &= a_i + \int_{a_i}^{\infty}\mathbb{P}(X_i>t)dt\end{aligned}$$

By similar logic,

$$\mathbb{E}\big(\max(X_i,a_i)^2\big) = a_i^2 + \int_{a_i^2}^{\infty}\mathbb{P}(X_i^2>t)dt$$

Combining the above terms,

$$\begin{aligned}\text{Var}(\max(X_i,a_i)) &= \int_{a_i}^{\infty}\mathbb{P}(X_i^2>t)dt - 2a_i\int_{a_i}^{\infty}\mathbb{P}(X_i>t)dt - \left(\int_{a_i}^{\infty}\mathbb{P}(X_i>t)dt\right)^2\\ \frac{\partial}{\partial a_i}\text{Var}(\max(X_i,a_i)) &= 2\left(\int_{a_i}^{\infty}\mathbb{P}(X_i>t)dt\right)\left(\mathbb{P}(X_i>a_i)-1\right)\end{aligned}$$

□

Claim 5.

$$\frac{\partial}{\partial a_i}\mathbb{E}(\tau_i+\Delta_i-d_i)^+ = \mathbb{P}(\Delta_i>d_i-a_i)\mathbb{P}(X_i<a_i)$$

Proof. Re-writing $\frac{\partial}{\partial a_i}\mathbb{E}(\tau_i+\Delta_i-d_i)^+$ using the law of total expectation,

$$\begin{aligned}&\frac{\partial}{\partial a_i}\mathbb{E}_{\Delta_i}\big(\mathbb{E}_{X_i}\big(\max(X_i,a_i)+\phi-d_i|\Delta_i=\phi\big)\big)\\ &\quad= \mathbb{E}_{\Delta_i}\big(\frac{\partial}{\partial a_i}\mathbb{E}_{X_i}\big(\max(X_i,a_i)+\phi-d_i|\Delta_i=\phi\big)\big)\end{aligned}$$

Expanding the inner term, and introducing $p_{X_i}(\cdot)$ as notation for the probability measure of random variable X_i,

$$\begin{aligned}&\frac{\partial}{\partial a_i}\mathbb{E}_{X_i}\big(\max(X_i,a_i)+\phi-d_i|\Delta_i=\phi\big)\\ &\quad= \frac{\partial}{\partial a_i}\left(\int_0^{a_i}(a_i+\phi-d_i)^+p_{X_i}(\omega)d\omega + \int_{a_i}^{\infty}(X_i(\omega)+\phi-d_i)^+p_{X_i}(\omega)d\omega\right)\\ &\quad= \mathbf{1}(a_i>d_i-\phi)\mathbb{P}\big(X_i<a_i\big)\end{aligned}$$

where the last step follows after applying Leibniz's rule and canceling out terms. Lastly, we return to our application of law of total expectation to obtain

$$\begin{aligned}\frac{\partial}{\partial a_i}\mathbb{E}(\tau_i+\Delta_i-d_i)^+ &= \mathbb{E}_{\Delta_i}\big(\mathbf{1}(a_i>d_i-\Delta_i)\mathbb{P}\big(X_i<a_i\big)\big)\\ &= \mathbb{P}\big(\Delta_i>d_i-a_i\big)\mathbb{P}\big(X_i<a_i\big)\end{aligned}$$

□

Proof for claim 1. Collecting the results from claim 4 and claim 5, and setting this equal to 0 we obtain

$$\mathbb{P}\big(\Delta_i>d_i-a_i\big)\mathbb{P}\big(X_i<a_i\big)+\beta\left(\int_{a_i}^{\infty}\mathbb{P}(X_i>t)dt\right)\left(\mathbb{P}(X_i>a_i)-1\right)=0$$

$$\mathbb{P}\big(\Delta_i>d_i-a_i\big) = \beta\left(\int_{a_i}^{\infty}\mathbb{P}\big(X_i>t\big)dt\right)$$

Introducting $F_{\Delta_i}(\cdot)$ as the cumulative distribution function for the duration of task i, and recalling the fact that $\mathbb{P}\big(X_i>t\big) = \mathbb{P}\big(\max_{k\in\mathcal{U}_i}(\tau_k+\Delta_k)>t\big)$, we can further expand the result as

$$1-F_{\Delta_i}(d_i-a_i) = \beta\int_{a_i}^{\infty}\left(1-\prod_{k\in\mathcal{U}_i}F_{\Delta_k}(t-\tau_k)\right)dt$$

□

A.2 Proof for claim 2

Proof. Let $\mathcal{D}_i$ and $cl(\mathcal{D}_i)$ represent the immediate downstreams and all downstream ancestors of task i, respectively. $\frac{\partial}{\partial a_i}\big(\sum_i\mathbb{E}(\tau_i+\Delta_i-d_i)^+ + \frac{\beta}{2}\sum_i\text{Var}(\tau_i+\Delta_i)\big)$ can be decomposed into the following terms:

$$\begin{aligned}&\frac{\partial}{\partial a_i}\big(\mathbb{E}(\tau_i+\Delta_i-d_i)^+ + \frac{\beta}{2}\text{Var}(\tau_i+\Delta_i)\big)\\ &\quad+\sum_{j\in cl(\mathcal{D}_i)}\frac{\partial}{\partial a_i}\big(\mathbb{E}(\tau_j+\Delta_j-d_j)^+ + \frac{\beta}{2}\text{Var}(\tau_j+\Delta_j)\big)\\ &\quad+\sum_{j\notin cl(\mathcal{D}_i),j\neq i}\frac{\partial}{\partial a_i}\big(\mathbb{E}(\tau_j+\Delta_j-d_j)^+ + \frac{\beta}{2}\text{Var}(\tau_j+\Delta_j)\big)\end{aligned}$$

The third term evaluates to zero, since τ_j is independent of the placement for a_i when $j\notin cl(\mathcal{D}_i)$. Inside the second summation, if $j\in\mathcal{D}_i$ we can write $\tau_j(a_i)$ as

$$\tau_j(a_i) = \max\left(a_j, \max(a_i,X_i)+\Delta_i, \max_{k\in\mathcal{U}_j,k\neq i}(\tau_k+\Delta_k)\right)$$

where X_i corresponds to the notation introduced in claim 4. Since $\tau_j(a_i)$ is monotonically nondecreasing in a_i—and by extension it can be seen that $\tau_j(a_i)$ for its downstreams are similarly monotonic in a_i—it follows that $\frac{\partial}{\partial a_i}\mathbb{E}(\tau_j(a_i)+\Delta_j-d_j)^+$ will be nonnegative. For

the variance term inside the second summation, if task durations are assumed to be independent from their start times this term reduces to $\frac{\partial}{\partial a_i}\frac{\beta}{2}\text{Var}(\tau_j(a_i))$. Since $\text{Var}(\max(X'_j, a_j)) \geq \text{Var}(\max(X_j, a_j))$ when a_j is a constant and X'_j stochastically dominates X_j (proof of this fairly intuitive property is omitted), and X_j is montonically nondecreasing with a_i, $\frac{\partial}{\partial a_i}\frac{\beta}{2}\text{Var}(\tau_j(a_i))$ can also be guaranteed to be nonnegative. Combining the above logic with the results of claims 4 and 5 renders

$$\frac{\partial}{\partial a_i}\Big(\sum_i \mathbb{E}(\tau_i + \Delta_i - d_i)^+ + \frac{\beta}{2}\sum_i \text{Var}(\tau_i + \Delta_i)\Big)$$
$$= \mathbb{P}\big(X_i < a_i\big)\bigg(\mathbb{P}\big(\Delta_i > d_i - a_i\big) - \beta\int_{a_i}^{\infty}\mathbb{P}(X_i > t)dt\bigg) + \psi(a_i)$$

for some nonnegative function $\psi(\cdot)$. From this it follows that plugging into the above relation $\hat{a}_i$ suggested in claim 1 yields $\psi(\hat{a}_i) \geq 0$, which means the objective function can only be further reduced by decreasing $\hat{a}_i$. Since the above relation also establishes a necessary condition for optimality in the resource-unconstrained problem (assumed in this claim), it further follows that $a_i^* \leq \hat{a}_i$. □

A.3 Proof for claim 3

Proof. Assume the alternate case holds, in other words that a task k which satisfies the conditions in claim 3 has an optimal dispatch time $a_k^* > 0$, with corresponding optimal dispatch times a_i^* for its immediate downstream tasks. Consider a new dispatch time a'_k satisfying $0 \leq a'_k < a_k^*$ and $F_{\Delta_i}(t - a'_k) > F_{\Delta_k}(t - a_k^*)$ for any $t \in [a_i^*, \infty)$. Note that $F_{\Delta_i}(t - a'_k) \geq F_{\Delta_k}(t - a_k^*)$ for all t and all $a'_k \leq a_k^*$. Recall from the proof of claim 2 that a necessary condition for optimality in the resource-unconstrained problem is

$$\mathbb{P}\big(X_i < a_i^*\big)\bigg(\mathbb{P}\big(\Delta_i > d_i - a_i^*\big) - \beta\int_{a_i^*}^{\infty}\mathbb{P}(X_i > t)dt\bigg) + \psi(a_i^*) = 0$$

where, as cited in the proof of claim 1, $\mathbb{P}\big(\Delta_i > d_i - a_i^*\big) - \beta\int_{a_i^*}^{\infty}\mathbb{P}(X_i > t)dt$ can be rewritten as

$$1 - F_{\Delta_i}(d_i - a_i^*) - \beta\int_{a_i^*}^{\infty}\bigg(1 - \prod_{k\in\mathcal{U}_i} F_{\Delta_k}(t - \tau_k)\bigg)dt$$

Substituting the definition $\tau_k = \max(X_k, a_k)$ and noting $X_k = 0$ by definition for a task k with no upstreams, then $\beta\int_{a_i^*}^{\infty}(1 - F_{\Delta_k}(t - a'_k) * \prod_{j\in\mathcal{U}_i, j\neq k} F_{\Delta_j}(t - \tau_j))dt < 1 - F_{\Delta_i}(d_i - \hat{a}_i)$. This implies the partial derivative in shown in claim 2 is positive, which is a contradiction. The contradiction further holds for all a'_k unless $\mathbb{P}\big(a'_k + \Delta_k > a_i^*\big) = 0$ for all i such that $k \in \mathcal{U}_i$. If $\mathbb{P}\big(\Delta_k > \hat{a}_i\big) > 0$ as stated in the claim, then necessarily $\mathbb{P}\big(\Delta_k > a_i^*\big) > 0$ since $\hat{a}_i \geq a_i^*$ as per claim 2, which completes the proof. □