Neptune: Scheduling Suspendable Tasks for Unified Stream/Batch Applications

Panagiotis Garefalakis
Imperial College London

Konstantinos Karanasos
Microsoft

Peter Pietzuch
Imperial College London

Abstract

Distributed dataflow systems allow users to express a wide range of computations, including batch, streaming, and machine learning. A recent trend is to unify different computation types as part of a single *stream/batch* application that combines latency-sensitive ("stream") and latency-tolerant ("batch") jobs. This sharing of state and logic across jobs simplifies application development. Examples include machine learning applications that perform batch training and low-latency inference, and data analytics applications that include batch data transformations and low-latency querying. Existing execution engines, however, were not designed for unified *stream/batch* applications. As we show, they fail to schedule and execute them efficiently while respecting their diverse requirements.

We present Neptune, an execution framework for stream/batch applications that dynamically prioritizes tasks to achieve low latency for stream jobs. Neptune employs *coroutines* as a lightweight mechanism for suspending tasks without losing task progress. It couples this fine-grained control over CPU resources with a locality- and memory-aware (LMA) *scheduling policy* to determine which tasks to suspend and when, thereby sharing executors among heterogeneous jobs. We evaluate our open-source Spark-based implementation of Neptune on a 75-node Azure cluster. Neptune achieves up to 3× lower end-to-end processing latencies for latency-sensitive jobs of a stream/batch application, while minimally impacting the throughput of batch jobs.

CCS Concepts

• **Information systems** → **Data analytics**; • **Computer systems organization** → *Cloud computing*; • **Software and its engineering** → Scheduling.

Keywords

Unified Analytics, Performance, Scheduling

ACM Reference Format:
Panagiotis Garefalakis, Konstantinos Karanasos, and Peter Pietzuch. 2019. Neptune: Scheduling Suspendable Tasks for Unified Stream/Batch Applications. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357223.3362724

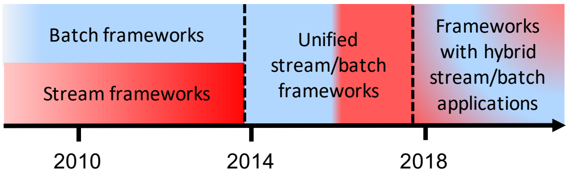

Batch frameworks
Stream frameworks
Unified stream/batch frameworks
Frameworks with hybrid stream/batch applications
2010
2014
2018

Figure 1: Evolution of stream and batch frameworks

1 Introduction

Over the past decade, distributed dataflow frameworks have enabled users to process vast amounts of data on clusters of machines [22, 58]—we track their evolution in Figure 1. Early implementations were dedicated to *either* batch [4, 8, 46, 61] *or* stream processing [7, 11, 16]. The former focused mainly on high processing throughput, whereas the latter had also to achieve low processing latency. Acknowledging the limitation, both for users and cluster operators, of having to manage multiple systems and copy data across them, unified distributed dataflow frameworks evolved to support both batch and stream processing [1, 2, 16, 42, 61]. This evolution is in line with the current momentum towards unified analytics in general, combining large-scale data processing with state-of-the-art machine learning and AI [54]. Towards this direction, these systems have also exposed unified programming interfaces to seamlessly express stream and batch applications [2, 3, 12].

As the next step in this unification, users have begun to combine latency-sensitive stream jobs with latency-tolerant batch jobs *in a single application* [19, 34]. We term these applications "*stream/batch*" where "stream" refers to the latency-sensitive jobs of the application and "batch" to the remaining ones. Examples of such applications can be found in the domains of online machine learning, real-time data transformation and serving, and low-latency event monitoring and reporting.

Consider the stream/batch application in Figure 2, which implements a real-time detection service for malicious behavior at an e-commerce company [51]. A batch job trains a machine learning model using historical data, and a stream job performs inference to detect malicious behavior. The stream/batch application therefore allows for the jobs to share application logic and state (e.g., using the same machine learning model between training and inference). This sharing facilitates application development and management. In addition, combined cluster resources can be used more efficiently by the different job types.

In this work, we show that supporting stream/batch applications presents new opportunities in terms of job scheduling. The stream jobs of these applications must be executed with minimum delay to achieve low latency, which means that batch jobs may have to be

SoCC '19, November 20–23, 2019, Santa Cruz, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6973-2/19/11…$15.00
https://doi.org/10.1145/3357223.3362724

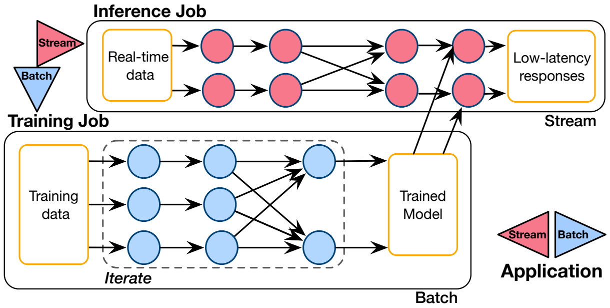


Figure 2: Stream/batch application for online malicious behavior detection (A batch job trains a model using historical data; a stream job performs inference to detect malicious behavior.)

preempted. Prior to unified stream/batch applications, stream and batch jobs had to be deployed separately in a cluster and cooperate with each other through general-purpose resource managers such as YARN [56] or Mesos [30]. This may result in low utilization of resources when dedicating parts of the cluster to each system, loss of progress when having to kill batch jobs to prioritize stream ones, or high preemption times [17, 21, 24, 30, 56, 58].

Given that batch and stream jobs share the same runtime, our key idea is to employ application-specific mechanisms and policies that dynamically prioritize stream jobs without unduly penalizing batch jobs. As we show in §2, existing schedulers for stream/batch applications, such as Spark's Fair scheduler [10], do not take advantage of this opportunity and only prioritize jobs based on static priorities, which introduces queuing delays for latency-sensitive tasks [32, 37, 39].

We describe **Neptune**, a new execution framework for stream/-batch applications that dynamically prioritizes latency-sensitive jobs, while achieving high resource utilization. Our paper makes the following contributions:

(i) Lightweight suspendable tasks. To prioritize latency-sensitive tasks, Neptune *suspends* tasks that belong to batch jobs on-demand. As a lightweight task preemption mechanism, Neptune uses *coroutines*, which avoid the overhead of thread synchronization. Coroutines can suspend batch tasks within milliseconds, thus reducing head-of-line blocking for latency-sensitive tasks. When tasks are resumed, they preserve their execution progress. To the best of our knowledge, Neptune is the first distributed dataflow system to use coroutines for task implementation and scheduling (§4).

(ii) Locality/memory-aware scheduling policy. To satisfy the requirements of stream/batch applications when they share the same executors, Neptune uses a *locality- and memory-aware (LMA) scheduling policy* for task suspension. It load-balances by equalizing the number of tasks per node, thus reducing the number of preempted tasks; it is locality-aware by respecting a task's preferences; and minimizes memory pressure due to suspended tasks by considering real-time executor metrics (§5).

(iii) Spark-based execution framework. Neptune is designed to natively support stream/batch applications in Apache Spark [61], one of the most widely-adopted distributed dataflow frameworks. Users express stream/batch applications using the existing unified Spark API and add only one configuration line to indicate jobs as "stream" or "batch". The implementation of Neptune is available as open-source[1] (§6).

Our experimental evaluation demonstrates the benefits of Neptune for stream/batch applications (§7). On a 75-node Azure cluster using machine learning and streaming benchmarks, Neptune reduces end-to-end latencies by up to 3× compared to Spark and increases the throughput by up to 1.5×. Using the locality- and memory-aware (LMA) policy, Neptune achieves close to optimal performance for both latency-critical and latency-tolerant tasks with a modest memory impact. Through micro-benchmarks, we confirm that suspendable tasks can efficiently pause and resume with sub-millisecond latencies and that, in scenarios with memory pressure, LMA scheduling policy can better utilize resources.

2 Supporting Unified Applications

We begin with background on distributed dataflow platforms in §2.1. We then describe the requirements for unified stream/batch applications and the associated opportunities in §2.2. Finally, we explore the design space when scheduling such applications in §2.3.

2.1 Distributed dataflow platforms

Distributed dataflow platforms, such as Hadoop [4], Spark [61], Flink [16], and Dryad [33], have enabled users to analyze large datasets using clusters of machines. In these platforms, user computation is expressed as a graph of operators that processes dataflows.

Application programming interface. To develop user programs, or *applications*, distributed dataflow platforms permit the mixture of SQL-style relational queries with functional or imperative code. The platform then distributes the computation on the entire cluster. Lately, we witness a shift towards a unification of APIs. This allows users to express both stream and batch computation through a single API, such as Spark's Structured Streaming API [12], Flink's Table API [3], or the external API layer of Apache Beam [2]. Listing 1 shows an example of the use of a unified API to implement a stream/batch application, as described below.

Execution model. An application consists of multiple *jobs*, and each job is converted to a *directed acyclic graph* (DAG) of operators. Operators in the DAG are grouped into *stages*. Each stage corresponds to a collection of *tasks* that perform computation over different partitions of data. At each stage boundary, data is written to a local cache residing in memory or disk by tasks in the upstream stages and then transferred over the network to tasks in the downstream stages.

For efficiency reasons, most platforms follow an *executor model* in which tasks are dispatched to long-running executors (as opposed to instantiating a container for each task). Executors are deployed on worker nodes and typically run for the entire lifetime of an application. Each executor provides *slots* that represent the compute capabilities of the node. Each slot can execute a single task. An *application scheduler* assigns tasks to executors, respecting data dependencies, resource constraints and data locality [28, 29, 59].

[1] Neptune is publicly available at: https://github.com/lsds/Neptune

Listing 1: Application for real-time detection

```
1  val trainData = context.read("malicious-train-data")              ⎫
2  val pipeline = new Pipeline().setStages(Array(                     ⎪
3    new OneHotEncoderEstimator(/* range column to vector */),        ⎬ Batch
4    new VectorAssembler(/* merge column vectors */),                 ⎪
5    new Classifier(/* select estimator */)))                         ⎪
6  val pipelineModel = pipeline.fit(trainData)                        ⎭
7  val streamingData = context                                        ⎫
8    .readStream("kafkaTopic")                                        ⎪
9    .groupBy("userId") /* some aggregations omitted */               ⎪
10   .schema() /* input schema */                                     ⎬ Stream
11 val streamRates = pipelineModel                                    ⎪
12   .transform(streamingData) /* apply model to input */             ⎪
13 streamRates.start() /* start streaming */                          ⎭
```

Evolution of execution model. As discussed in §1, early dataflow systems were dedicated to either stream or batch computation. Thus, an application comprising both stream and batch jobs would have to be executed by separate engines, e.g., following a "lambda" architecture [38]. This dichotomy poses a burden to users, who must deploy multiple frameworks, and to cluster operators, who must split cluster resources among these engines. To this end, dataflow platforms evolved to support both stream and batch applications. Using a unified stream/batch platform, such as Spark [61] or Flink [16], the application in Listing 1 can now be executed by a single engine.

2.2 Hybrid stream/batch applications

Along with the evolution of dataflow platforms towards unified stream/batch engines, users started combining latency-sensitive stream and latency-tolerant batch jobs as part of the same application. Such unified designs simplify development and operationalization: users can reuse code with consistent semantics, avoid the complexity of interacting with external systems, and have a single application to monitor and tune.

Use case: Real-time malicious behavior detection. Listing 1 shows the implementation of the real-time malicious behavior detection service [51] from Figure 2. The service is realized as a hybrid stream/batch application with multiple jobs: a batch job performs model training by iterating over the historical training data to reflect the latest changes and saves a `model` (lines 1–6), which, in turn, is shared in memory with a real-time streaming job that uses the model for real-time prediction (lines 7–13). The streaming job computes a series of aggregations on the real-time Kafka input and then queries the model to detect malicious behavior (line 12).

Requirements. To support stream/batch applications, an execution engine must *combine the different requirements* of stream and batch jobs regarding latency[2] and throughput, while achieving high resource utilization. In Listing 1, the output of the latency-sensitive streaming job must be computed with low latency to ensure fast reaction times; the latency-tolerant batch job must run with high throughput to efficiently utilize the remaining resources and reflect the most recent state in the model, as the training data is continuously updated.

When all resources are occupied, a streaming job may have to wait for other tasks to finish before starting processing, which leads to increased latency due to the queuing delay. Therefore, a platform for stream/batch applications must *minimize queuing delays for latency-sensitive jobs*, even under high resource utilization. At the same time, the throughput of batch jobs should not be compromised. Therefore, resources must be shared across jobs depending on their needs, while avoiding starvation and maintaining high utilization. A unified execution engine must ensure *high throughput and resource utilization* despite the heterogeneity of job types.

The fact that jobs in stream/batch applications are part of a single application executed by one execution engine allows us to employ application-aware techniques to meet the above requirements.

2.3 Scheduling alternatives

Next we discuss how the jobs of a stream/batch application (see Figure 2) are handled by existing distributed dataflow platforms, such as Spark, showcasing their well-known limitations [32, 37, 39].

Figure 3 illustrates the scheduling problem that a distributed dataflow system faces. Here the stream/batch application consists of a real-time inference job and a historical data training job. Rectangles at the top represent job stages, and each task in a stage requires one resource unit. Execution time is normalized in time units (T). Light blue rectangles represent tasks of the (stream) inference job; dark blue ones represent (batch) training tasks. Note that training tasks require up to 3 times more time and 2 times more resources than the inference tasks and can thus introduce significant queuing delays to the latter.

In an ideal scheduling scenario, stream tasks would start execution immediately to achieve the lowest latency, which is $2T$. The remaining resources would be used by batch tasks for high resource utilization and throughput.

A basic approach that avoids queuing and achieves low latency employs *static resource allocation* in which each job uses a dedicated set of resources (DIFF-EXEC in the figure). This corresponds to approaches that either use separate engines for stream and batch, or use a single engine but submit separate applications for each job. It is a common solution in existing production environments that use general-purpose resource managers and have dedicated resource queues for latency-critical jobs [17, 21, 25].

In our example, we assign 25% of resources to the stream job executor and the rest to the batch job executor. Although the latency for the stream job is low, it is still 2× higher than the optimal, as separate executors cannot guarantee low latency. As resources cannot be shared across jobs, the application completes in $7T$, which is the worst of all scenarios. Jobs could be allowed to go over capacity, but that would result in either higher queuing delays for stream jobs or wasted work for batch jobs, as we discuss below. Moreover, this approach does not permit the sharing of application state or logic across jobs.

By default, unified stream/batch engines such as Spark [13] and Flink [16] schedule applications with multiple jobs in a FIFO fashion, based on job priorities and job submission times. If the training job gets triggered first, it will consume all cluster resources and when the inference job arrives, it will get queued behind it, leading to an increased latency of $6T$.

[2] In a dataflow system, latency is the time between receiving a new record and producing output for this record.

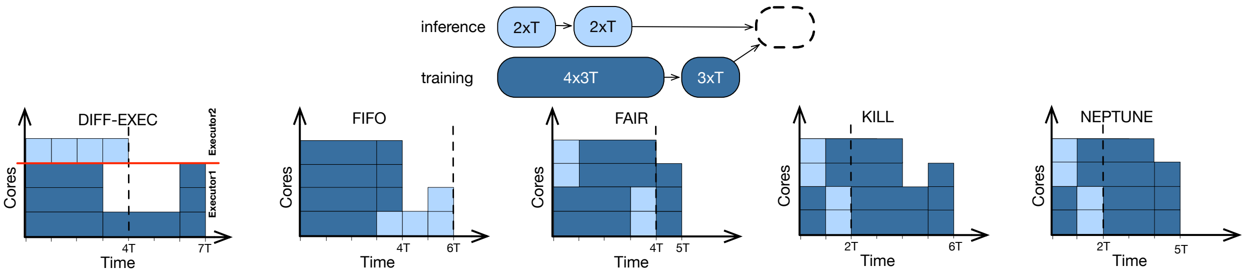


Figure 3: Resource usage over time for the detection application under different scheduling policies

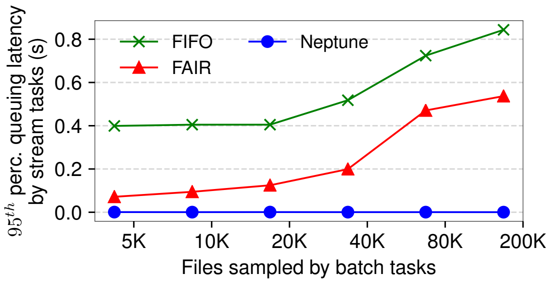


Figure 4: Stream task queuing latency

FAIR is an alternative scheduling policy that runs tasks in a round-robin fashion. All jobs receive an equal, possibly weighted, share of resources. Stream jobs submitted while a batch job is running can obtain resources right away and experience good response times. In our example, FAIR achieves better utilization (the application completes in $5T$) and reduces the stream job's response time by $2T$, compared to FIFO. FAIR, however, cannot guarantee low queuing delays. The achieved latency is 2× higher than the optimal, because FAIR treats all jobs as equal without respecting latency requirements.

To avoid queuing delays for stream tasks, it is possible to employ non work-preserving *preemption* by killing tasks of batch jobs. This can be combined with any of the above strategies. In Figure 3, KILL together with FAIR shows minimal queuing, but requires restarting killed tasks and losing their progress. This leads to the second worst effective resource utilization and thus throughput—the application completes in $6T$. Even if jobs supported work-preserving preemption, one would have to account for the extra overhead of checkpointing task state.

NEPTUNE, our solution shown in the rightmost part of Figure 3, prioritizes stream tasks by *suspending* running batch tasks, minimizing queuing delays ($2T$ runtime for the stream job). When resources become available, batch tasks are resumed without losing progress, thus achieving high resource utilization ($5T$ for application completion) without any checkpointing overhead.

Measuring queuing delay. To assess the effect of different scheduling policies on the queuing delay in practice, we run the stream/-batch application from Listing 1 in Spark for varying sizes of training jobs (by increasing the number of files they consume). Figure 4 shows the 95th percentile of queuing latency for the stream tasks. The results indicate that Spark's default policies, FIFO and FAIR, cannot achieve low queuing delays for the latency-sensitive stream tasks as we increase the training load (and, thus, resource utilization). In contrast, Neptune results in minimal queuing delays, even under high resource utilization by efficiently suspending batch tasks. As we show in §7, this leads to significant gains both in terms of end-to-end latencies and task throughput.

3 Neptune Design

The design of Neptune, our solution to support the execution of stream/batch applications, is shown in Figure 5. To reduce the queuing of latency-sensitive tasks during execution, Neptune uses *suspendable tasks*, which employ efficient coroutines to yield and resume execution. Neptune uses a *centralized scheduler* to assign application tasks to *executors*, which periodically send metrics back to the scheduler in the form of heartbeats. The scheduler enacts policies that trigger task suspension and resumption, which are executed locally on each executor. The scheduling policies guarantee job requirements (see §2.2) and satisfy higher-level objectives such as balancing cluster load.

3.1 Defining stream/batch applications

In Neptune, stream/batch applications can be expressed with Spark's lower-level Resilient Distributed Dataset (RDD) API or its higher-level DataFrame API. All jobs in Neptune share a common execution environment, also known as *context*. A context maintains the connection to application executors and is used to create jobs and broadcast variables. Each executor can only have a single active application context.

A user specifies the requirements for a stream/batch application by tagging jobs using a local job property. At submission time, a stream or a batch job is configured with a `neptune.priority` property set to `high` or `low`, respectively. This and other local properties (e.g., to select a job resource pool/queue under FAIR scheduling), are inheritable variables, maintained at the job level and automatically transmitted to each child stage and task. The propagation is achieved through the application context when submitting a job, which passes the properties to the job scheduler responsible for stage execution.

3.2 Scheduling stream/batch applications

In traditional dataflow frameworks, a scheduler builds an execution plan for an application and executes its jobs. The scheduler in Neptune also enacts policies that suspend and resume tasks according to their requirements.

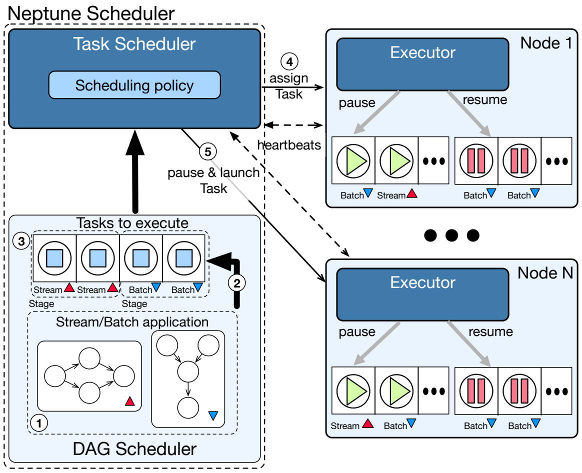


Figure 5: NEPTUNE architecture (Executors maintain separate queues for running and suspended tasks, and periodically send heartbeats to the central scheduler. The scheduler assigns tasks to executors and decides to suspend already-running ones according to its scheduling policy.)

Figure 5 shows how the NEPTUNE scheduler is comprised of two main components: the DAG Scheduler and the Task Scheduler. The DAG Scheduler computes the execution plan of each job in the form of a directed acyclic graph (DAG) (step 1).[3] Each DAG vertex represents a stage, which groups tasks that can execute the same computation in parallel. During the execution of the application, the DAG Scheduler keeps track of the tasks that are already completed. When all data dependencies of a stage are met, it adds the tasks of that stage to a priority queue of tasks ready to be executed next (step 2). Within the task queue, tasks are ordered based on whether they are stream or batch, dictated by their user-defined priority, as discussed in §3.1 (step 3). For example, a stage of a latency-sensitive stream job is always placed before a stage of a latency-tolerant batch job in the queue. For stages that are not tagged by the user, NEPTUNE uses FIFO ordering based on submission time.

Tasks that are ready-to-execute are then passed to the Task Scheduler. To execute tasks of stream stages without delay, NEPTUNE introduces *suspendable* tasks. Any task implemented as suspendable can be interrupted and later resumed without loss of progress (see §4). When a new task arrives, it is immediately executed if there are sufficient free resources available (step 4). Otherwise, running batch tasks are suspended, as dictated by the scheduling policy, to free up resources for stream tasks to start their execution immediately (step 5). To avoid losing the progress made by a preempted task, the oldest suspended task is scheduled for execution when a stream task terminates.

Suspended tasks are maintained locally on the same worker because using them on different workers would require task migration. Migration would incur the extra costs of transferring input data or intermediate outputs and would require a policy to decide when to migrate. Since NEPTUNE preempts batch tasks (which are of arbitrary length) to prioritize the execution of stream tasks (which are typically short) and reacts in milliseconds to maintain high resource utilization, it does not support task migration by design.

3.3 Executing stream/batch tasks

Task execution in NEPTUNE is performed by a set of executors, which are also responsible for task suspension and resumption, as dictated by the job scheduler. Each executor has c cores and can run c concurrent tasks. Periodically, executors also send heartbeats to the scheduler with information about task state and performance metrics such as memory usage, CPU utilization, and garbage collection and disk spilling activity.

Executors maintain queues of running and suspended tasks. When a task is assigned to an executor, it is added to the running task queue and is executed immediately when there are available cores in the executor; otherwise, when a stream task preempts a running task, the executor first marks the batch task as paused, moves it from the running to the suspended queue, and prepares the new task for execution with low delay. As tasks complete and resources are freed, the scheduler resumes suspended tasks (the oldest suspended task is resumed first). The executor then shifts the tasks from the suspended to the running queue and continues execution.

Discussion. Although NEPTUNE is implemented on top of Apache Spark, our design, as shown in Figure 5, is applicable to other dataflow frameworks for stream/batch applications, such as Apache Flink [16] and Storm [11], that follow a similar architecture. We opted to have NEPTUNE's first implementation on Spark, hence we cannot claim concrete gains on other systems. We expect Flink, which uses a continuous-operator execution model, to benefit even more from task suspension, as operators typically run longer than Spark's scheduled tasks.

4 Suspendable Tasks

To minimize the queuing delay for stream jobs with a low impact on batch jobs, NEPTUNE requires an efficient preemption mechanism for suspending tasks and resuming them without loss of progress. This mechanism should be transparent to users and not require a bespoke programming model, e.g., that requires users to trigger periodic checkpoints within tasks [26].

A classical approach to suspend tasks would rely on thread synchronization. Worker threads that execute tasks would use the *wait* and *notify* primitives to suspend and resume. Thread synchronization primitives typically require system calls, and the preemption is performed by the OS scheduler, which moves threads between a wait and a running queue. As the number of threads increases, thread synchronization with OS involvement therefore can become a bottleneck and introduce extra delays (see §7.4).

To obtain a scalable and efficient mechanism for task suspension, NEPTUNE instead uses stackful *coroutines* [47]. Coroutines are resumable functions that can suspend execution at *yield points* inside the computation. A regular function can be considered a special case of a coroutine that does not suspend execution and returns to the caller after completion. When a coroutine suspends execution, it returns a *status handle* to the caller, which can be later used to resume execution.

Unlike checkpointing mechanisms [17, 24], coroutines suspend tasks quickly because the state of local variables is saved in a call

[3] The compilation step that translates a user-submitted application to a set of DAGs is omitted from the figure.

Listing 2: Collect coroutine task

```
1 val collect : (TaskContext, Iterator [T]) -> (Int, Array[T]) =
2   coroutine {(context: TaskContext, itr : Iterator [T]) =>
3     val result = new mutable.ArrayBuffer[T]
4     while (itr.hasNext) /* iterate records */
5     | result.append(itr.next) /* append record to dataset */
6     | if (context.isPaused()) /* check task context */
7     ⌊ ⌊ yieldval (0) /* yield value to caller */
8   result.toArray /* return result dataset */
```

stack represented as arrays. Stackful coroutines can also call each other and return to the same resume site. This is important for implementing dataflow tasks composed of functions that call each other (e.g., shuffle tasks, as explained below). Coroutines can directly replace existing subroutine tasks by maintaining the same API, offering task preemption transparently to users.

Neptune uses stackful coroutines to implement *suspendable tasks*, which have a yield point after the processing of each record. As an example, Listing 2 shows the implementation of the collect task that returns all dataset elements. Notice the yield point in line 7. The coroutine task receives the taskContext and a record iterator as arguments (line 2). In the iteration loop (lines 4–7), the new record is first appended to the result dataset (line 5). Every task is associated with an individual task context, which determines whether the task will be suspended, as dictated by the scheduling policy. If the task context is marked as paused, it returns a value to the caller to express whether the task was suspended gracefully (line 7); otherwise, it continues execution. In a similar manner, coroutines can implement simple task logic, such as aggregations, or more complex logic, such as nested coroutine calls.

The executor component runs the suspendable tasks. Figure 6 shows how an executor creates a suspendable ShuffleMapTask instance by invoking a *call* function (step 1). The suspendable task uses a coroutine stack to store its context, local variables, and state, including the partition and jobId. Depending on the number of output partitions, type of aggregation and serializer, the ShuffleMapTask internally calls a Writer method either from a serialized sort (i.e., BypassMergeSortShuffleWriter) or deserialized sort (i.e., SortShuffleWriter) class (step 2). The Writer is another coroutine with a separate stack storing its local variables and state, including the ShuffleId, mapId, and the record iterator. The Writer coroutine can return to the same caller as the ShuffleMapTask. Once launched, the Writer executes up to a point and then receives a *pause* call from the executor (omitted from the figure). The Writer then *yields* a value to the executor, indicating the reason for its suspension (step 3). The executor can then *resume* the same task instance (up to the next yield point or to completion) or invoke another function (step 4).

The above is a typical example of function composition that can be expressed with stackful coroutines. Coroutine composition enables both the calling and called coroutines to yield, adding multiple suspension points, which is necessary for suspending tasks with more complex logic.

Discussion. Although suspendable tasks are both efficient to suspend/resume and transparent to users, they keep state in memory, which may increase memory pressure on executors. We observe, however, that typical production clusters tend not to be constrained by memory.

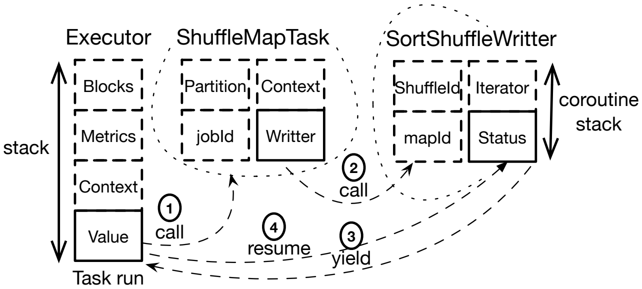


Figure 6: Neptune coroutine composition (The executor runs a coroutine ShuffleMapTask, which instantiates a coroutine SortShuffleWriter. Both can yield to the executor when the TaskContext is paused.)

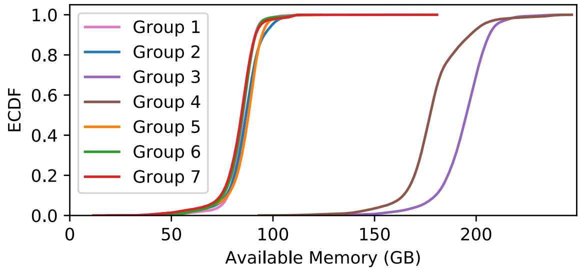


Figure 7: Available machine memory in a production cluster within Microsoft (The cluster is comprised of tens of thousands of machines with each group corresponding to different hardware configurations. Data is over four days.)

Figure 7 shows the available machine memory in a Microsoft production cluster comprised of tens of thousands of machines. We took measurements every few seconds over the course of four days, and plot the CDF of the available machine memory, grouped by machines with the same hardware configuration. The available memory in 90% of the machines is at least 75 GB and in some configurations over 150 GB. Tasks in this cluster typically require less than 10 GB of memory each, which means that several more tasks can be started at each machine in terms of memory. Similar trends hold at other companies. For example, in production Spark clusters at Facebook, 95% of the machines utilize no more than 70% of the machine's memory [52].

In cloud deployments, users often overprovision their memory resources [20] (and, if not, they can always allocate more memory to their VMs)—Neptune takes advantage of this to reduce the latency of higher-priority stream tasks. In cases when memory does become the bottleneck, Neptune's LMA scheduling policy, described in §5, ensures that machines with the least memory pressure are picked to suspend tasks.

Another noteworthy point is how fast coroutines can yield, which is determined by the amount of processing per data record. Although optimizations such as function pipelining can increase yield times, we find that the difference is typically negligible in practice. In our experiments (§7.4), we only observe increased yield

Algorithm 1: LMA scheduling policy

```
   Input: List Executors,          // In descending preference order
1  List ExecutorsMetricsWindow                    // Executor metrics
2  Upon event HeartBeat hb from Executor e do
3      ExecutorsMetricsWindow[e].push(hb.metrics)
4      Executors.updateOrdering
5  return
6  Upon event Submit Task t do
       // Cache local executor
7      Executor $t_{exec} \leftarrow$ hostToExecutor.get(t.cacheLocation)
8      if $t_{exec}$ is None or $t_{exec}$.freeMemory $<$ threshold then
           // Executor with the least pressure
9          $t_{exec} \leftarrow$ Executors.head
10     if $t_{exec}$ has availableCores then
           // Launch task $t$ on free cores
11         $t_{exec}$.Launch($t$)
12     else
           // Suspend task $t_p$ and launch $t$
13         Task $t_p \leftarrow t_{exec}$.tasks.filter(LowPriority)
14         $t_{exec}$.PauseAndLaunch($t$, $t_p$)
15 return
```

times for the handling of large file partitions when queries are non-selective. As we explain, this is caused by the file reader mechanism in Spark and can be mitigated.

5 LMA scheduling policy

Suspendable tasks (§4) provide the mechanism for efficient task preemption. The scheduler, however, must decide *which* tasks to preempt and *when*. These scheduling decisions are crucial to achieve low latency for stream tasks with minimum disruption to batch tasks, while accounting for task locality and cluster load balance.

As described in §3.2, when a new job stage is submitted, it is first added to a queue of tasks pending for execution. Once the stage dependencies are met, a list of tasks is submitted to run. Batch tasks run immediately as long as there are enough free resources, otherwise they wait for resources to become available. For stream tasks, Neptune's scheduler uses the LMA (*locality- and memory-aware*) scheduling policy, outlined in Algorithm 1.

LMA considers the stream task's locality preferences[4] and the executors' memory conditions. To avoid executors with high memory pressure, it uses metrics such as memory usage, disk spilling, and garbage collection activity. These metrics are collected periodically through the heartbeats between the executors and the scheduler (line 2) and are smoothened over configurable windows (line 3) to update a preference order of executors (line 4).

When deciding where to place a task, LMA first checks whether the task has a preference to a specific executor. If so, and the executor's free memory is above a threshold (we use twice the task's memory demand to make sure there is sufficient memory in the executor), this executor is picked; otherwise, LMA selects the executor with the least memory pressure (lines 7–9). If the selected executor has available CPU cores, the algorithm sends a *launch* message to the executor (line 11); otherwise, it selects a batch task to preempt (line 13), and sends a *pause & launch* message (line 14). This message suspends the batch task and starts the new one in a single network round trip.

To prevent low priority jobs from being suspended indefinitely, Neptune also has an *anti-starvation* mechanism (omitted from the pseudocode). For each task, Neptune tracks the number of times that it has been suspended. If the task has been paused more than a given number of times, it is run uninterrupted for a period of time, ensuring progress of every stage. Note that the same mechanism, along with application-specific knowledge, can be used to bound the delay incurred by important batch tasks, such as tasks that update shared state.

6 Implementation

We implemented Neptune by extending Apache Spark version 2.4.0. Our changes (approximately 20,000 lines of Scala code) are in the Spark scheduler and execution engine. Neptune's code is publicly available at: https://github.com/lsds/Neptune. Figure 8 depicts Neptune's integration with Spark. The modified Spark components and the new Neptune ones are in blue.

Spark includes a `Driver` that runs on a master node and an `Executor` on each worker node, all running as separate processes. The `Executor` uses a thread pool for running tasks. An application consists of stages of tasks, scheduled to run on executors. Every application maintains a unique `SparkContext`, the entrypoint for job execution.

DAG scheduler. On application submission, the application's `SparkContext` hands over a logical plan to the `DAGScheduler`, which translates it to a directed acyclic graph (DAG). Each DAG is split into stages at shuffle boundaries. The `DAGScheduler` maintains which RDDs and stage outputs are materialized, determines the locations for running each task, and decides on an execution schedule. Neptune extends the `DAGScheduler` to allow stages with different requirements, expressed as priority levels (§3.1). Neptune currently supports two priority levels, high and low, for stream and batch jobs, respectively.

Task scheduler. Each DAG stage consists of a set of tasks and tasks are submitted for execution by the `TaskScheduler`. Neptune extends the `TaskScheduler` to support the suspension of running tasks and the launch of new ones in a single round trip (through a *pause & launch task* message).

Scheduling policy. Neptune uses Spark's existing policy for scheduling batch tasks and the LMA policy, described in §5, to determine where to place stream tasks and when to suspend running batch tasks. As tasks terminate and the paused task queue in a worker node is not empty, the oldest suspended task is resumed, as long as there are no stream tasks pending.

Executor. On new task launch, an executor deserializes the task content, creates a `TaskMemoryManager`, initializes a runner, and finally executes the task on a thread pool. Neptune extends the executor with an extra suspended task queue and the transition logic between task states. A pause & launch call suspends a running task and marks the `TaskContext` as paused, yielding in the next record iteration. The executor then adds the task in the paused queue and frees the unused resources from the `TaskMemoryManager`.

[4] These are locality preferences to nodes that already hold locally the data that will be used for computation.

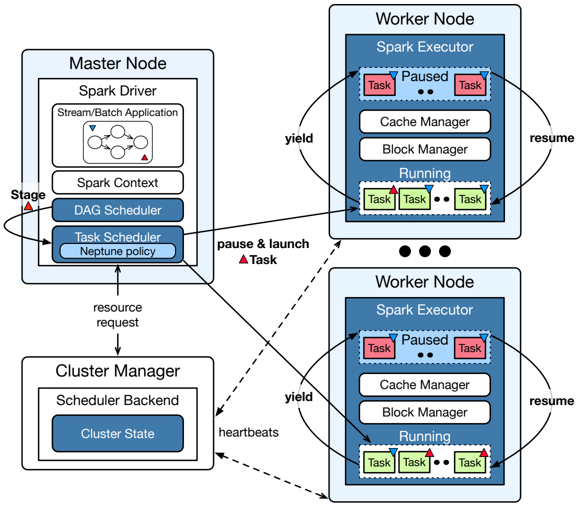


Figure 8: Neptune integration in Spark (As part of the master node, the Neptune scheduler assigns tasks to worker nodes. On each worker node, Neptune maintains a queue of running and suspended tasks.)

For the cached data partitions and intermediate task shuffle outputs, executors provide a local cache via the `CacheManager` and the `BlockManager`, respectively.

Cluster state. Our LMA scheduling policy (§5) requires additional information from each executor (e.g., disk spilling activity) to take memory pressure into account. Neptune augments Spark's cluster state by having executors piggyback such metrics onto the heartbeat messages. The cluster manager maintains a sliding-window moving average per metric.

Suspendable tasks. A task is the smallest unit of execution associated with an RDD partition running on the executors. Every task in Spark, and consequently Neptune, has the notion of locality, inherited by the implementation of the underlying RDD. For example, when using HadoopRDDs to read data from HDFS, the locality preferences are based on the nodes where the HDFS blocks reside. A task can be either: a `ResultTask` that executes a function on the RDD partition records and sends the output back to the driver; or a `ShuffleMapTask` that computes records on the RDD partition and writes the results to the `BlockManager` for use by later stage tasks. To support suspendable tasks in a way that is completely transparent to the user, Neptune reimplements all basic `ResultTask` and `ShuffleMapTask` functionality in Spark across programming interfaces using coroutines (§4).

7 Evaluation

Next we evaluate the performance of Neptune. After describing our setup (§7.1), we present the following experiments: (i) we compare the execution of stream/batch applications with Neptune and with existing solutions using typical application benchmarks [18, 31, 55] (§7.2); (ii) we measure the performance impact in terms of throughput and latency with varying resource demands (§7.3); and (iii) we explore Neptune's task suspension mechanism through micro-benchmarks (§7.4).

7.1 Experimental setup

Cluster setup. We run our experiments on 75 Azure E4s_v3 virtual machine (VM) instances. Each VM has 4 CPU cores, 32 GB of memory, and 60 GB of SSD storage. We use 4 slots on each VM for executing tasks. We deploy Apache Spark version 2.4.0 as the baseline for our experiments. All experiments use the same JVM, heap size, and garbage collection flags. We warm up the JVM before taking measurements.

Workloads. We employ the following workloads:

(1) The *Yahoo Streaming Benchmark* (YSB) models analytics on a stream of ad impressions [18]. A producer inserts records in a stream and the benchmark groups the events into 10-second windows per ad campaign. It then measures how long it takes for all events in the window to be processed. Our stream/batch application combines two concurrent instances of YSB, a latency-sensitive and a latency-tolerant one.

(2) The *machine learning (ML) training/inference* application uses online Latent Dirichlet Allocation (LDA) to perform topic modeling and inference, similar to our example of Figure 2. LDA is an unsupervised machine learning method that uncovers hidden semantics ("topics") from a group of documents, each represented as a group of tokens. We use the NYTimes dataset for training and a small subset for inference [31]. The dataset has 300k documents, 100k words, 1k topics and 99.5 million tokens. We use the online variational Bayes LDA algorithm with parameter values miniBatchFraction=0.05 and maxIterations=50. Gibbs sampling is used to infer document topic assignments, as in Spark MLlib [53].

(3) *TPC-H* [55] is a decision support benchmark with 22 analytical relational queries, which include computation logic such as aggregates, large joins, and arithmetic computations [14]. We use a scale factor of 10 with data stored in Parquet format [9].

Comparisons. We compare the following systems:

Neptune (or NEP): This is our system using the *locality and memory-aware* (LMA) scheduling policy (see §5) that respects task locality preferences and load-balances preempted tasks across nodes.

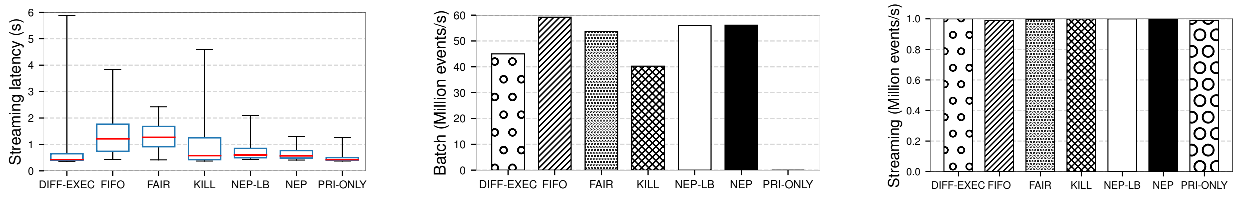
NEP-LB: This is Neptune with a simpler policy that load balances task preemption based on the nodes' memory condition but ignoring task locality preferences.

FIFO and FAIR: These are the two non-preemptive scheduling policies available in Apache Spark today. FIFO policy prioritizes tasks based on job submission time; FAIR policy assigns resources to jobs proportionally to their weight. For FAIR, we configure the weight for stream jobs to be equal to the job parallelism and for batch jobs to be equal to one.

KILL: This is a preemptive scheduling policy combined with the FAIR policy. It allows batch tasks to be killed to minimize queuing delays for stream tasks.

DIFF-EXEC: We deploy two executors per machine, each using half the CPU cores. One executor is used for high-priority tasks and the other for low-priority ones.

PRI-ONLY: We execute only the stream tasks in complete isolation to achieve the ideal latency scenario.

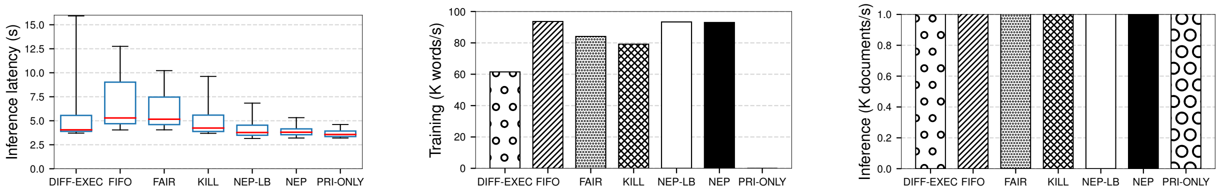


(a) Streaming query latency

(b) Batch query throughput

(c) Streaming query throughput

Figure 9: Yahoo Streaming benchmark (streaming + batch)



(a) Inference latency

(b) Training query throughput

(c) Inference query throughput

Figure 10: LDA NYTimes dataset (training + inference)

7.2 Application performance

First, we study the effect of Neptune on end-to-end application performance for different applications.

Yahoo Streaming Benchmark (YSB). We measure the end-to-end latency for each window as the difference between the last event processed and the window end times; we measure throughput as the total number of records processed over time. Our application consists of two YSB jobs. The latency-sensitive stream job has parallelism to occupy 15% of cluster resources and generates thousands of events/second. The latency-tolerant batch job can occupy all cluster resources and generates millions of events/second.

Figure 9a shows the end-to-end latency for the stream job. We use box plots in which the lower/upper parts of the box represent the 25th/75th percentiles, respectively; the middle line is the median; and the whiskers are the 5th and 99th percentiles, respectively.

FAIR scheduling compared to FIFO reduces the 75th and 99th percentile latencies for the stream job by 5% and 37%, respectively. It still remains more than 2× higher than for PRI-ONLY with the stream job in isolation. By preempting batch tasks, KILL reduces the median latency by 54% compared to the non-preemptive FAIR, but the 99th percentile is almost 2× higher than FAIR. The reason is twofold: (i) KILL cannot preempt more than a weighted share of resources, which is ineffective when too many stream tasks wait for execution; and (ii) the housekeeping process of killing tasks in a Spark executor involves releasing locks and cleaning up allocated memory and pages in the block manager, which is time-consuming.

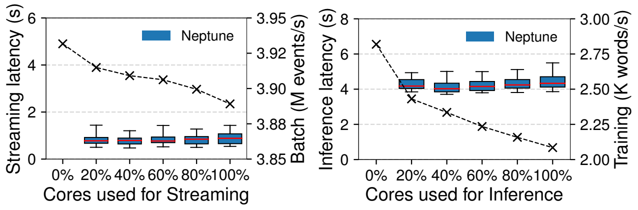
Neptune with the LMA policy (NEP) achieves latencies comparable to PRI-ONLY. NEP-LB that does not consider cache preferences achieves 61% worse latency compared to NEP for the 99th percentile—cache preferences affect task latencies, especially at the higher percentiles. Finally, DIFF-EXEC reduces the median latency but increases the tail latency. We observe high tail latencies when executors on the same machine interfere with each other—Neptune avoids this by having a single executor per machine and using an effective task scheduling policy.

The scheduling policies also affect the batch query throughput, as shown in Figure 9b. As FIFO is unaware of priorities and fairness, it achieves the highest throughput, as batch jobs consume all cluster resources when scheduled. FAIR's improved latency compared to FIFO comes at the cost of decreased batch throughput by 10% due to the prioritization of stream over batch tasks. As expected, due to the termination of batch tasks, KILL achieves the worst throughput, namely 32% lower than FIFO. Although DIFF-EXEC runs the batch jobs on separate executors, it experiences a 24% lower throughput than FIFO (but 12% higher than KILL). This is because DIFF-EXEC relies on a static allocation of resources with up to 85% of executors used for batch jobs. Finally, both NEP and NEP-LB achieve a throughput comparable to the best performing FIFO policy (within 5%).

Figure 9c shows identical streaming job throughput for all configurations. Spark uses a micro-batching model that accumulates streaming events and periodically triggers processing for all available data. Over time, the non resource-demanding stream job computation is amortized, resulting in fixed throughput but varying latency. Thus, for a share of resources, Spark achieves similar throughput, whether tasks are scheduled immediately or queued.

ML training/inference application. Next we evaluate the topic modeling application that has a training and an inference job. The training job generates a new model after a number of iterations and stores it in HDFS using all available resources. The inference job is loading the latest trained model in cache and infers topics for the received documents, consuming up to 15% of cluster resources.

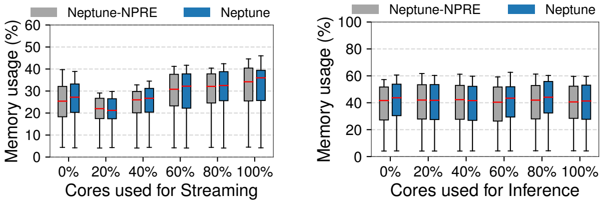
Figure 10a shows the latency achieved for the inference job. As LDA training tasks are computationally intensive, queuing inference tasks behind training tasks has a significant impact on task latency. Although FAIR scheduling reduces the 99th percentile latency by 13% compared to FIFO, it is still 3× higher than the latency



(a) Varying YSB streaming

(b) Varying LDA inference

Figure 11: Performance impact of varying demands



(a) Varying YSB streaming

(b) Varying LDA inference

Figure 12: Memory impact of varying job demands

of running only the inference job (PRI-ONLY). Note that the preemptive KILL policy reduces the median latency by 18% compared to FAIR, but its 99th percentile latency is 55% higher than running the job in isolation (PRI-ONLY), as explained above for the YSB stream/batch application.

NEP achieves a latency that is just 6% and 12% higher than PRI-ONLY for the median and the 99th percentiles, respectively. Neptune without cache awareness (NEP-LB) has a 29% worse latency for the 99th percentile compared to NEP. Although running the jobs in different executors (DIFF-EXEC) reduces median latency, it incurs high tail latencies due to executor interference.

Figure 10b reports the achieved throughput for the training job. Similar to the YSB application, FIFO yields the highest throughput, entirely ignoring job priorities. FAIR and KILL achieve 10% and 15% lower throughputs, respectively, compared to FIFO by prioritizing inference tasks over training. DIFF-EXEC, which uses only up to 85% of available executors, achieves a 34% lower throughput compared to FIFO. Unlike YSB, in this application, DIFF-EXEC performs worse than KILL because reduced resources have more impact on computationally intensive training jobs. Finally, both NEP and NEP-LB achieve comparable throughput to the best performing FIFO policy, with an overhead of 1%.

Figure 10c shows identical inference throughput across configurations, as explained for YSB above.

In summary, Neptune achieves latencies comparable to the ideal PRI-ONLY policy for latency-sensitive jobs, and reduces the throughput of latency-tolerant jobs by less than 5% compared to the best performing FIFO policy.

7.3 Varying resource demands

Next we evaluate the performance in terms of throughput and latency under varying resource demands.

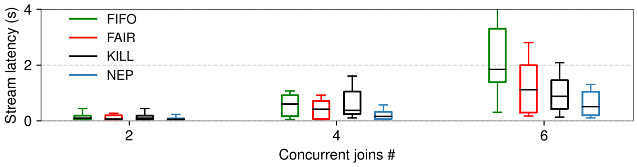


Figure 13: Latency impact of increasing memory use

Different job resources. The applications above only use part of the cluster resources for the stream jobs. In this experiment, we use a cluster of 4 Azure VMs, and we increase the stream job resource demands from 0% to 100%; batch jobs still use all available resources. We use Neptune to execute jobs and measure latency, throughput, and memory consumption.

Stream latency. As we run YSB with different resource demands, we observe in Figure 11a that Neptune maintains low latency across all percentages. Since the tasks that must be suspended increase, this shows the effectiveness of our preemption mechanism. The higher number of preempted tasks, however, incurs a penalty in throughput: when stream tasks ask for 40% and 100% of the cluster, throughput drops by 0.8% and 1.1%, respectively, compared to optimal (0%). Similarly, increasing inference resources in the LDA application from 0% to 100% does not affect latency in Neptune (Figure 11b). When inference tasks demand 40% and 100% of the cluster, throughput drops by 17% and 26%, respectively.

Memory overhead. We also explore the effect of task preemption in Neptune by measuring memory usage. In Figure 12, we compare with plain Neptune without preemption (Neptune-NPRE). As shown in Figure 12a, when YSB streaming tasks are first introduced in the system, the task scheduler schedules fewer batch tasks. Memory usage drops by 25% for the 99th percentile for Neptune, both with and without preemption. It then steadily increases in line with the stream job demands. Across the full range, however, Neptune’s memory usage is no more than 2% for the 99th percentile compared to Neptune-NPRE. Similarly for the ML training/inference application, even though there is no drop in memory usage, the increase is no more than 1.5% for the 99th percentile compared to Neptune-NPRE (Figure 12b). We conclude that Neptune’s preemption mechanism only introduces a modest memory overhead.

Memory pressure. We investigate the effectiveness of Neptune under memory pressure. For this, we synthetically create a memory-constrained environment. We use the same 4 Azure VMs, and we reduce each worker’s JVM size to 1 GB. We use a single YSB stream instance with 30% of the cluster resources and an increasing number of joins, joining millions of rows as the batch workload. The worker’s average memory utilization increases from 20% to 40% and 80% with 2, 4, and 6 joins, respectively.

Figure 13 shows the end-to-end latency for the streaming job across different scheduling policies. With 2 concurrent joins and low memory pressure, all policies achieve similar mean latencies of around 100 ms. As we increase the number of joins to 6 and 8, Neptune achieves 99th latencies that are 1.9× and 3× lower than FIFO (2.8× and 1.7× lower than KILL), respectively. NEP-LB, ignoring individual task locality preferences, performs similar to Neptune in

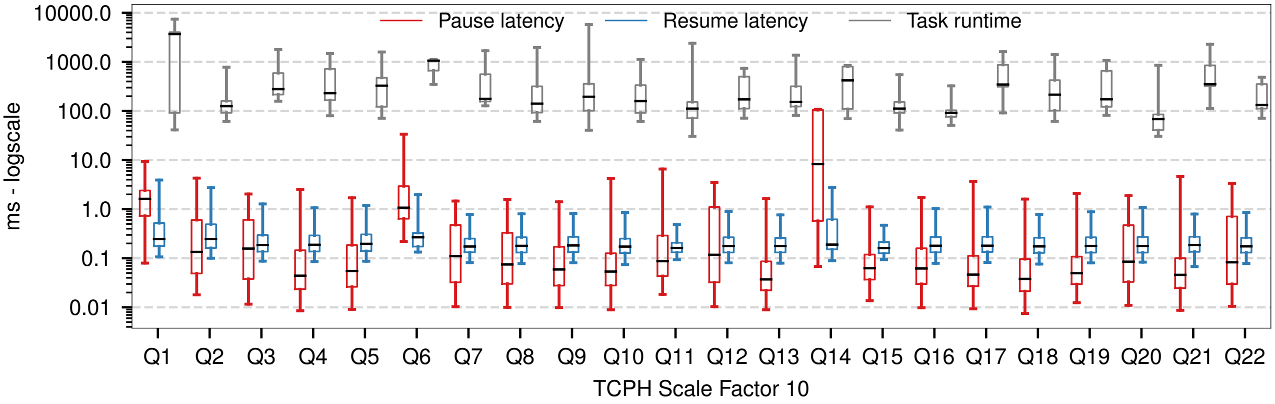


Figure 14: Neptune task suspension mechanism latency breakdown using TPC-H queries

this context. Using real-time metrics, Neptune manages to predict accurately the memory pressure of each individual executor, thus reducing memory bottlenecks to obtain lower tail latencies.

7.4 Task suspension

Next we evaluate Neptune's coroutine-based task suspension mechanism using the TPC-H workload using solely batch tasks [55].

Suspension latency. We first run the TPC-H benchmark on a cluster of 4 Azure VMs and measure the task duration distribution for each query. Using a custom `TaskEventListener`, we rerun the benchmark and continuously transition tasks from `PAUSED` to `RESUMED` states until completion, while measuring the latency for each transition. As coroutine tasks can yield after each record iteration, a task that does heavy per-record computations takes longer to get suspended. By continuously triggering yield points, we measure the worst case scenario in terms of transition latency for each query task.

Figure 14 shows the task runtime and pause/resume latency distributions for all 22 queries, and whiskers represent the 5th and 99th percentiles, respectively. We observe that, although queries have different task runtimes ranging from 100s of milliseconds to 10s of seconds, Neptune pauses and resumes tasks with sub-millisecond latencies.

An exception is Q14 with a 75th percentile pause latency of 100 ms. This query has no filters other than on the date, which is the attribute by which data is partitioned. This means that the Parquet reader has to consume full partitions before the corresponding Spark tasks can yield. The less selective the query, the longer it takes to consume each partition (less/no filters are pushed down to the reader), which results in increased suspension times. This problem can be mitigated by repartitioning the table, increasing the task parallelism, or improving readers to be more fine-grained which is orthogonal to our approach.

Coroutines vs. thread synchronization. Finally, we compare Neptune's coroutine-based task preemption with an alternative approach based on traditional thread synchronization (ThreadSync). Thread synchronization relies on the preemption performed by the OS scheduler and is implemented in Neptune using thread *wait* and *notify* calls. We compare Coroutines with ThreadSync running TPC-H queries on a 32-core Azure VM with the custom `TaskEventListener` alternating tasks from `PAUSED` to `RESUMED` states, while increasing the parallelism of the executor from 2 to 64.

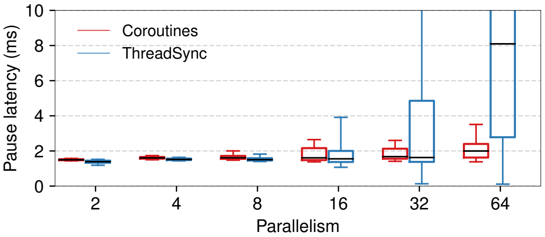


Figure 15: Latency of task suspension mechanisms

Figure 15 shows the results for the first TPC-H query (with similar behavior for the rest). With up to 8 parallel tasks, both mechanisms have low latency. As the parallelism increases, however, the overhead of ThreadSync rises: the 99th percentile of yield latency increases by 2.6× for 16 threads compared to Coroutines. As the OS scheduler must continuously arbitrate between wait/run queues, ThreadSync exhibits worse scaling compared to coroutines, which bypass the OS.

Note that each stream task typically has a runtime of a few 10s of milliseconds. At the same time, the gains from coroutine task suspension compared to thread synchronization are in the range of a few milliseconds and can be as high as 600 ms for the 99th percentile with 64 threads (omitted from Figure 15). Therefore, the overhead of suspending a task with thread synchronization can be a significant part of a stream task's runtime (up to orders of magnitude higher in the worst case), considerably affecting its latency. As stream jobs consist of multiple tasks, potentially triggering 100s of task suspensions, such overheads accumulate.

8 Related Work

We compare Neptune to existing systems, focusing on distributed dataflow execution engines and scheduling policies.

Dataflow execution engines. There are numerous dataflow frameworks for batch [41, 57, 61] or streaming [7, 11, 16] computation, which expose a variety of programming models and provide scalability and fault-tolerance [23, 33, 60]. The recent demand for hybrid stream/batch applications resulted in several unified APIs on top of those frameworks [1, 16, 61], such as Spark's Structured Streaming [12], Flink's Table API [3], or external layers such as Apache Beam [2]. Unlike Neptune, these systems only provide logical abstractions for stream/batch jobs without a unified execution engine that is aware of job requirements.

SnappyData [40] is a unified engine that supports streaming, analytics and transactions by integrating Spark with a transactional main-memory database. To realize low-latency transactions, it bypasses the Spark scheduler. Instead, Neptune builds on Spark's execution engine to achieve low latency for arbitrary jobs.

Several recent systems in the context of databases [35, 48] and machine learning [49] have used coroutines to reduce the latency of operations, but Neptune is the first system to do so for a distributed dataflow framework.

Scheduling Policies. A significant amount of research work has investigated the problem of scheduling jobs in large clusters. Mesos [30], YARN [56], and Borg [58] are general-purpose resource managers that schedule jobs from different frameworks and implement resource sharing policies [5, 6, 28]. They lack, however, an understanding of the application structure and task dependencies. Jobs with shorter task durations [43] led to the development of distributed job schedulers such as Sparrow [44] and Apollo [15]. These schedulers assume independent jobs and thus perform distributed scheduling across jobs. In Neptune, we use a centralized scheduler to target stream/batch applications with jobs that share state and dependencies. Their requirements cannot be handled efficiently by application-agnostic schedulers.

More critical jobs can be prioritized with static policies such as fair scheduling [6], but this does not avoid queuing delays. Schedulers such as Mercury [36], Yaq [50], and Hawk [25] prioritize tasks based on runtime estimates. Neptune could also exploit estimates to derive task priorities. Even if such estimates are available and accurate [45], they rely on the assumption that shorter tasks are of a higher priority, which is not always the case.

Existing preemptive schedulers can minimize queuing. Non-work-preserving schedulers require task resubmission, which results in loss of progress; work-preserving ones [17, 24] require coarse-grained checkpointing of state, which leads to increased reaction times (in the order of seconds). Therefore, existing preemptive schedulers either waste cluster resources or fail to provide the millisecond reaction times needed in unified execution engines.

Finally, schedulers such as MEDEA [27] optimize the placement of long-running executors in a cluster given a set of constraints. In Neptune, we dynamically prioritize tasks within executors once they are already placed. In fact, Neptune could utilize placement constraints to ensure high-quality container placement in shared compute clusters.

9 Conclusion

Motivated by emerging trends in dataflow systems, we presented Neptune, an execution framework to support stream/batch applications that share executors. Neptune dynamically prioritizes latency-critical jobs, while effectively utilizing application resources. With its suspendable tasks based on coroutines, the design of Neptune paves the way for truly unified stream/batch applications.

Acknowledgments

We would like to thank our shepherd, Rebecca Taft, and the anonymous reviewers for their valuable comments. We also thank Alexandros Koliousis, Jana Giceva, Carlo Curino, Holger Pirk, and George Theodorakis for their insightful feedback throughout this work. This work was supported by the EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS) (EP/L016796/1).

References

[1] Apache Apex. http://apex.apache.org. 2019.

[2] Apache Beam. http://beam.apache.org. 2019.

[3] Apache Flink's Table API. https://ci.apache.org/projects/flink/flink-docs-stable/dev/table/tableApi.html. 2019.

[4] Apache Hadoop. http://hadoop.apache.org. 2019.

[5] Apache Hadoop Capacity Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html. 2019.

[6] Apache Hadoop Fair Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html. 2019.

[7] Apache Heron. http://heronstreaming.io. 2019.

[8] Apache Hive. https://hive.apache.org. 2019.

[9] Apache Parquet. https://parquet.apache.org. 2019.

[10] Apache Spark Fair Scheduler Pools. https://spark.apache.org/docs/latest/job-scheduling.html#fair-scheduler-pools. 2019.

[11] Apache Storm. http://storm.apache.org. 2019.

[12] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. *SIGMOD*, 2018.

[13] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark SQL: Relational Data Processing in Spark. *SIGMOD*, 2015.

[14] Peter Boncz, Thomas Neumann, and Orri Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. *Technology Conference on Performance Evaluation and Benchmarking*, 2013.

[15] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. *OSDI*, 2014.

[16] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.

[17] Wei Chen, Jia Rao, and Xiaobo Zhou. Preemptive, Low Latency Datacenter Scheduling via Lightweight Virtualization. *USENIX ATC*, 2017.

[18] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Tom Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking Streaming Computation Engines at Yahoo! *Yahoo! Engineering Blog*, 2015.

[19] Continuous Applications: Evolving Streaming in Apache Spark 2.0. https://databricks.com/blog/2016/07/28/continuous-applications-evolving-streaming-in-apache-spark-2-0.html. 2016.

[20] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. *SOSP*, 2017.

[21] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based Scheduling: If You're Late Don't Blame Us! *SoCC*, 2014.

[22] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. *NSDI*, 2019.

[23] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 2004.

[24] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive Data Center Scheduling Without Runtime Estimates. *SoCC*, 2018.

[25] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid Datacenter Scheduling. *USENIX ATC*, 2015.

[26] Lu Fang, Khanh Nguyen, Guoqing Xu, Brian Demsky, and Shan Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. *SOSP*, 2015.

[27] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. MEDEA: Scheduling of Long Running Applications in Shared Production Clusters. *EuroSys*, 2018.

[28] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. *NSDI*, 2011.

[29] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. *OSDI*, 2016.

[30] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. *NSDI*, 2011.

[31] Matthew Hoffman, Francis R Bach, and David M Blei. Online Learning for Latent Dirichlet Allocation. *NIPS*, 2010.

[32] How to avoid long running jobs blocking short running jobs. http://apache-spark-user-list.1001560.n3.nabble.com/How-to-avoid-long-running-jobs-blocking-short-running-jobs-td33882.html. 2018.

[33] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *EuroSys*, 2007.

[34] Xiaowei Jiang. A Year of Blink at Alibaba: Apache Flink in Large Scale Production. https://www.dataversity.net/year-blink-alibaba. 2017.

[35] Christopher Jonathan, Umar F. Minhas, James Hunter, Justin J. Levandoski, and Gor V. Nishanov. Exploiting Coroutines to Attack the "Killer Nanoseconds". *PVLDB*, 2018.

[36] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. *USENIX ATC*, 2015.

[37] Lightweight pipeline execution. http://apache-spark-user-list.1001560.n3.nabble.com/Lightweight-pipeline-execution-for-single-eow-td33506.html. 2018.

[38] Nathan Marz and James Warren. Big Data: Principles and best practices of scalable real-time data systems. *New York; Manning Publications Co.*, 2015.

[39] Mixing Long Run Periodic Update Jobs With Streaming Scoring. http://apache-spark-user-list.1001560.n3.nabble.com/Mixing-Long-Run-Periodic-Update-Jobs-With-Streaming-Scoring-td25705.html. 2018.

[40] Barzan Mozafari, Jags Ramnarayan, Sudhir Menon, Yogesh Mahajan, Soubhik Chakraborty, Hemant Bhanawat, and Kishor Bachhav. SnappyData: A Unified Cluster for Streaming, Transactions and Interactice Analytics. *CIDR*, 2017.

[41] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. *SOSP*, 2013.

[42] Open Sourcing Delta Lake. https://databricks.com/blog/2019/04/24/open-sourcing-delta-lake.html. 2019.

[43] Kay Ousterhout, Aurojit Panda, Josh Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The Case for Tiny Tasks in Compute Clusters. *HotOS*, 2013.

[44] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. *SOSP*, 2013.

[45] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A Kozuch, and Gregory R Ganger. 3Sigma: Distribution-based cluster scheduling for runtime uncertainty. *EuroSys*, 2018.

[46] Presto. http://prestodb.io. 2019.

[47] Aleksandar Prokopec and Fengyun Liu. Theory and Practice of Coroutines with Snapshots. *ECOOP*, 2018.

[48] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. *PVLDB*, 2017.

[49] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A. Gibson, and Eric P. Xing. Litz: Elastic Framework for High-Performance Distributed Machine Learning. *USENIX ATC*, 2018.

[50] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient Queue Management for Cluster Scheduling. *EuroSys*, 2016.

[51] Robert Xue. Real-time fraud detection at Groupon. https://bit.ly/2Chmnv7. 2018.

[52] Sameer and Ankit Agarwal. Scaling Apache Spark at Facebook. https://databricks.com/session/scaling-apache-spark-at-facebook. Spark and AI Summit, 2019.

[53] Spark MLLib. http://spark.apache.org/docs/latest/ml-guide.html. 2019.

[54] Spark+AI Summit. https://databricks.com/sparkaisummit/north-america. 2019.

[55] TPC-H Benchmark. http://www.tpc.org/tpch. 2019.

[56] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. *SoCC*, 2013.

[57] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. *SOSP*, 2017.

[58] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. *EuroSys*, 2015.

[59] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. *EuroSys*, 2010.

[60] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. *NSDI*, 2012.

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