공학석사학위논문

Design and Implementation of a Flexible and Extensible Data Processing Runtime

유연하고 확장성 있는 데이터 처리 런타임의 디자인과 구현

2018 년 2 월

서울대학교 대학원

컴퓨터공학부

김 주 연

Abstract

Design and Implementation of a Flexible and Extensible Data Processing Runtime

Joo Yeon Kim
Department of Computer Science and Engineering
The Graduate School
Seoul National University

Today's data analytics applications take a wide variety of characteristics. They are also executed in various resource environments, with many distinct requirements. To face these requirements, many systems have been developed with optimization techniques that are suitable for each system's needs. However, the field of data processing is continuously growing with diverse requirements for job characteristics and resource environments. With current system designs which demonstrate pre-defined runtime behaviors, it is extremely difficult to apply new optimization techniques to them. Onyx is a system that approaches to solve this problem by designing and implementing a flexible and extensible execution runtime. The Onyx execution runtime is designed and implemented around the execution properties that must be flexibly controllable and extensible in order for jobs to be executed under the desired runtime behaviors. It uses a user configurable job representation, Onyx IR, annotated with execution properties which control the underlying runtime behaviors for each job to flexibly execute jobs according to users' requirements. Examples and evaluations show that new optimization techniques are easily applicable to Onyx, which otherwise require a significant amount of engineering effort using current data processing systems.

Keywords: Data Processing, Data Processing Framework, Data Analytics, Data Analytics Framework, Data Processing Engine, Data Analytics Engine

Student Number: 2016-21194

Contents

List of Tables

List of Figures

List of Code

Chapter 1

Introduction

Extracting information and gaining insights from big data has become crucial in diverse domains over the past years. To meet this demand, many data processing systems have emerged with different designs to implement optimizations of their own. The implementations not only limit themselves to well-known common optimizations, but also involve specific techniques the system developers have devised. Such specific optimization techniques are usually tailored towards certain job characteristic and resource environment requirements. As a result, they have distinct runtime behaviors optimized for those requirements.

For example, researches to overcome the challenges of processing jobs on cheap, unreliable transient resources have been conducted on system implementations of their own [1–3]. Along the lines of improving datacenter efficiency, systems that implement techniques for disaggregating different types of resources like memory, CPU and GPU, are being actively developed [4–7]. In this global era, cross datacenter problems must also be considered [8–10]. The space of resource environments is only becoming richer, with many potential research questions.

On another axis, we have diverse job characteristics. We have jobs with large scales of data, and systems that have succeeded in achieving reasonably good performance, each by implementing optimizations of its own. However, they often overlook to optimize the performance of processing smaller workloads, which are much more common workloads in practice [11]. Solving the common data skew problem can be different for different jobs, but each system focuses on its own optimization that may be not as effective for another job.

Although each of the systems performs well with the jobs and in the environments they target, the systems do not necessarily perform well for other unconsidered cases, and often do not consider supporting a diversity of environments and jobs in their designs. Thus, it is extremely challenging to quickly adapt such systems to new job characteristics and resource environments without substantial effort.

The fundamental cause of such challenges lies in the design of the system execution runtimes. Many of the systems often design their runtimes with various parts of job execution tightly integrated in the system core according to the optimizations they aim to implement. This is often useful in abstracting away the complexity of distributed computing, but pre-determines the runtime behaviors of job execution. For instance, computational parallelism, whether to push or pull input data from previous computations, and on which type of resources to run the submitted computations are solely determined by the implementation in the system core, without being configurable. Furthermore, since the system core implementations are monolithic, modifying or extending them for different job characteristics and resource environments is challenging. For an application writer to optimize an application to perform well on a certain system engraved with its underlying behaviors, it requires an understanding of the resource environments and job characteristics the system was designed for, an overhead that often requires a lot of time and effort. For a system developer to modify or extend such a system's runtime behaviors, the monolithic system cores must be modified, which requires an even deeper understanding of the system.

To overcome these limitations, we propose Onyx, a system that enables flexible and extensible runtime behaviors to support diverse job characteristics and resource environments. Onyx consists of a compiler and an execution runtime. In achieving our goal, we clearly identify and organize execution properties of data processing jobs and design our compiler and execution runtime around these properties. The compiler receives user applications and compiles them in the form of a directed acyclic graph (DAG), annotated with various configurable values for execution properties. The runtime executes the job DAG according to the execution properties for different job executions.

This paper focuses on the runtime of Onyx. The Onyx execution runtime has been designed to support the basics of data processing in its backbone, while exposing extensible components that can be flexibly configured to adapt to diverse job characteristics and resource environments.

Chapter 2

Background

This chapter explains the current state of the field of data processing, motivating the need for a more flexible and extensible system like Onyx. In Section 2.1, we clarify some of the basic concepts and terminology used for Onyx, and data processing systems in general. Section 2.2 lists out a few examples of job characteristics and resource environments that the data processing community is trying to optimize. In Section 2.3, we show a few examples that illustrate how many current systems often exhibit pre-defined runtime behaviors and how the system runtime core must be modified to achieve particular optimizations. Section 2.4 identifies *execution properties* from observing current data processing systems, a set of key properties that we use to build Onyx.

2.1 Data Processing Concepts

A typical data processing system consists of two major components, a compiler and an execution runtime. First, the compiler accepts an application from users, converts it into a dataflow job expressed as a directed acyclic graph (*DAG*), where the vertices represent operators for the application connected by edges that represent the dataflow between the operators. When the job DAG is submitted to the execution runtime, the runtime partitions the DAG into multiple sub-DAGs called *stages*. A *stage* is a unit of execution the runtime uses for scheduling a job. It often becomes useful when dealing with faults because the runtime can keep track of the job progress, knowing exactly where to restart when faults occur. After stage partitioning, each vertex(operator) is replicated in the form of multiple *tasks*, for the computation to be executed simultaneously as multiple parallel tasks. Each task is run on an *executor*, launched on a container allocated by a resource manager of the resource environment the job is executing on. Dataflow between operators occur from a task to another, in *blocks* and/or a *partitions*. The relationships between the sending and the receiving tasks are defined by *communication patterns* of the corresponding operators. Common patterns are many-to-many (a.k.a shuffle), one-to-many (a.k.a broadcast), many-to-one and one-to-one.

2.2 Optimizations for Data Processing

Researches have been conducted to optimize data processing jobs over the past years. Many have succeeded in identifying common problems in data processing and have come up with well working optimizations. Many others have succeeded in devising optimizations for specific problems. However, as the domain continues to evolve, new challenges continue to arise with the necessity for new optimizations. From observing numerous factors that bring about challenges in data processing jobs, we can categorize them into two big categories - job characteristics and resource environments. Here are examples in each category:

1. Job characteristics

    - Data skew is a popular problem which can vary depending on the workload. In many cases, there are high possibilities of input data being skewed, with a small number of unevenly popular keys such as internet traces [12]. Works like Optimus [13] solve this problem by applying optimizations during runtime, to detect data skew and evenly re-distribute data among tasks.
    - Workload scale varies from one job to another. For example, due to the prevalence of larger workloads, we mainly focus on optimizing the systems

for large scales as challenges mainly arise when processing them. Many general processing systems have implemented optimization techniques for relatively large workload scales starting from a few hundred GB upto PB scales by implementing optimizations of their own [14, 15]. But different optimization techniques may be applicable to different workload scales.

2. Resource environments:

    - Enhancing datacenter efficiency has many approaches to it. Borrowing and using cheap transient resources from over-provisioned resources for latency-critical jobs is one of many important approaches. Works including Pado [1], TR-Spark [2], and Flint [3] study optimizations to effectively recover from evictions in order to make decent progress even in frequent evictions of such transient resources.
    - Recent datacenters have started to disaggregate different types of resources like CPU, memory, GPU, and disk from one another. This approach not only increases datacenter efficiency, but also facilitates datacenter management. Facebook has announced and publicized a lot of their work related to resource disaggregation at several talks [5]. Along the resource disaggregation approach, the use of I-Files in Sailfish [4] is an optimization applicable to reduce disk I/O.
    - Many of today's online applications are serviced globally with multiple geo-distributed datacenters. Data processing across datacenters is becoming more widespread with many researches for optimizations in such wide area data analytics being conducted [8–10].

Specific data processing systems are optimized for the specific optimizations they aim to achieve. General data processing systems are often optimized for common analytics on common datasets running on common computing resources. Whether it be specific or general, they have limitations in covering unconsidered cases. We have only listed a few examples here. The diversity of job characteristics and resource environments only continues to grow. In order to face the upcoming diversity, one could simply say that we could implement new optimization techniques which define new job execution behaviors on an existing, general data processing system if necessary. Then two questions arise. 1. ”How easy is it to add a new method of job execution?” - a question of extensibility and 2. ”How easy is it to configure parts of a job to control how a job is executed?” - a question of flexibility, to get the system to use the new method of execution for new cases, or even to change small, existing parts of a job execution. Nevertheless, this turns out very challenging as it requires modifications to pre-defined runtime behaviors, which are often designed strongly tied to the system core, as we will see in the following section.

2.3 Pre-Defined Runtime Behaviors

This section illustrates how today’s state-of-the-art systems have pre-defined runtime behaviors for optimization techniques they choose to implement.

- **Spark** [15] is by far the most popular data processing system widely used for many applications. Its contributions include enhancing performance using in-memory computations based on resilient distributed datasets (RDDs) [16], and fault tolerance using lineages. In Spark, applications are translated into graphs of user-defined RDD transformations. These applications determine the computation parallelism and data partitioning inside a job, and remain unchanged during job execution. In the Spark runtime, the order in which stages are executed is fixed. With the graph of RDD transformations, stage partitioning occurs before the actual execution of a job. Stages are fixed to be partitioned at shuffle boundaries, and the Spark scheduler schedules these stages according to a topological ordered lineage of produced RDDs. Factors related to dataflow are fixed in the system runtime as well. Dataflow channel is fixed to memory backed by disk for shuffle transformations and other transformations are pipelined in memory. The dataflow model is fixed to the pull based model where the sender operator is complete and all its output blocks are ready before the data is transferred to the receiver operator. Spark can support machines with different number of cores and memory when placing executors. However, this turns out to take a lot of effort. Moreover, in order to make ”labeling” different executors more meaningful, scheduling specific tasks to specific executors must come together, which is not configurable in the Spark runtime.

- **Flint** [3] is an optimization technique that implements automated checkpointing and server selection policies to handle evictions of transient resources. The Flint prototype has been implemented by modifying the system core of Spark. As mentioned above, Spark does not allow flexible scheduling for various resources, so Flint modifies Spark to implement its server selection policy. Flint marks the inter-stage edges of the job DAG to be checkpointed for automated checkpointing. This requires an implementation of Flint's checkpointing policy into Spark's system core. Note that after the modifications, Flint no longer behaves like Spark during job execution. It has its own pre-defined runtime behaviors.

- **Pado** [1] is another optimization technique that places tasks appropriately to executors on transient or reserved resources. In order to make the placement more effective, Pado has a different stage partitioning strategy from Spark. Moreover, Pado uses the push dataflow model - the sender operator begins pushing its produced data to the receiving operator in smaller chunks (could be as small as pipelining the data) - in order to evacuate data out of transient resources as soon as possible. As Pado has many different approaches to job execution from Spark and because Spark runtime is not easy to modify, Pado implements its own execution runtime, with its own pre-defined behaviors.

- **Optimus** [13] is a framework that optimizes jobs dynamically by rewriting job execution graphs, built on top of an execution engine, Dryad [17]. It collects runtime statistics to rewrite the execution graph, but such graph rewrites are fixed in the system core as the controllable factors of the underlying system, Dryad, are pre-defined. For example, one of Optimus' major contributions is dynamic data skew handling, but this is only supported for the conventional data partitioning mechanisms because it is an extension to Dryad which keeps the scope of partitioning methods bounded. Optimus supports user-defined rewrites integrated with a high-level data-parallel language, but this is still restricted to choosing a sub-query to execute from a set of alternative sub-queries.

The above systems all exhibit pre-defined runtime behaviors tightly built into the system core. If a modification needs to be made, a deep understanding of the system core is necessary. Even if a modification is successfully made, the system can no longer behave in the old manner as the modification changes the pre-defined behaviors. Moreover, runtime level options like caching data in memory or checkpointing data in persistent storage is sometimes exposed to application writers. This requires them to not only write data processing applications, but also to understand the system core and the underlying runtime behaviors to optimize and fine-tune their applications. System designs with pre-defined runtime behaviors are successful in achieving optimized performance for given job characteristics and resource environments. However, the systems' core components must be thoroughly understood and rewritten in order to adapt to new requirements. This can be extremely laborious and challenging work.

2.4 Execution Properties

To facilitate implementing optimizations for new requirements, we need a system design that can flexibly change its execution when configured. Moreover, we need such configurable parts to be exposed for further extensions. From observing many data processing systems and optimization techniques, we have identified and organized several common *execution properties* that such systems have altered and configured flexibly for the execution runtime to behave as intended. *Execution properties* are parts of a job execution that determine runtime behaviors. Our idea is that when such execution properties are made flexible and extensible in the execution runtime, the runtime behaviors will no longer be pre-defined and the system would be able to easily support many optimization techniques for diverse job characteristics and resource environments. Table 2.1 lists the execution properties we have identified and defined to build Onyx.

Regarding operator computation, these properties include choosing the type of executor to run each task on (*executor placement*), determining the number of simultaneous parallel tasks (*parallelism*), and grouping operators together and ordering them for execution (*stages* and *schedule groups*). Regarding intermediate data transfer, we expose the option to choose where to store the data produced for each operator (*data store*), in which pattern the communication should occur (*data communication pattern*), how the output intermediate data should be partitioned (*partitioner*), how the input intermediate data should be assigned (*key assignment*), whether the data should be pulled or pushed (*dataflow model*), whether or not the data should be cached (*caching*), and how the used intermediate data should be handled (*used data handling*).

For example, *executor placement* can be applied to scheduling specific tasks to executors on resources like transient containers which are prone to evictions, or more stable reserved containers. While transferring data, intermediate data can be stored in memory or even on remotely distributed file systems, depending on the requirements.

Execution Property	**Description**
Executor Placement	Executor type to run a task on.
Parallelism	Number of simultaneous parallel tasks for an operation.
Stage	Stage ID given to each operator after stage partitioning.
Schedule Group	Priority number for scheduling a stage. Stages that belong to the same schedule group can be scheduled simultaneously.
Data Store	Where to store the intermediate data.
Data Communication Pattern	The pattern to write and read the intermediate data.
Partitioner	How to partition the output intermediate data.
Key Assignment	How to assign the partition keys to the receiving tasks.
Dataflow Model	How to fetch data from previous computations.
Caching	Whether to cache the intermediate data or not.
Used Data Handling	How the used intermediate data should be handled. e.g., delete, and save a backup in disk, if in memory.

Table 2.1: Onyx Execution Properties which Determine Runtime Behaviors

Chapter 3

Onyx Overview

In section 2.4, we identify the execution properties which should be exposed so that we can configure them to adjust runtime behaviors. This chapter gives an overview of our system, Onyx as a whole, in achieving this goal. Onyx consists of two main components, the compiler and the execution runtime. The compiler takes a user application as input and outputs an optimized physical execution plan. The Onyx system itself is implemented in Java 8, but its goal is to understand dataflow applications written in several high-level programming models, like applications written for Spark or written in Apache Beam [18]. When receiving applications, the compiler translates this logical layer into expressive, general-purpose intermediate representations (IR), called *Onyx IR*. Onyx IR involves a series of *optimization passes* which enable users to control optimizations for their jobs with execution properties made configurable by the runtime. Once Onyx IR completes going through the optimization passes, it is converted to an execution plan to be executed by the execution runtime.

3.1 Onyx IR

In this section, we briefly explain how the compiler uses Onyx IR to facilitate users to flexibly control execution properties made configurable and extensible in the execution runtime.

Onyx IR is a DAG of vertices which represent operators connected by dataflow edges between them. Each of the IR vertices and IR edges can be annotated to be able to express the different execution properties mentioned earlier (§2.4). Out of the 11 execution properties, *executor placement*, *parallelism*, *stage* and *schedule group* are related to the computational operators; thus we annotate vertices with values for each of these properties. The remaining execution properties (except for *caching* which we leave for future work) are related to the intermediate dataflow; thus we annotate edges with values for each of these properties. For example, edges that a user wants to store the intermediate data as local file can be annotated to use the ”local file” implementation of the execution runtime for the *data store* execution property.

3.2 Optimization Passes

When annotating each of the execution properties, the compiler goes through a series of *optimization passes*. Onyx IR can be flexibly modified, both in its logical structure and annotations, through the optimization pass interfaces. The modification during compile-time occurs in two ways: through *annotating passes* and *reshaping passes*. First, *annotating passes* annotate IR vertices and edges with specific execution properties to execute jobs with desirably configured runtime behaviors. For simplicity, an annotating pass requires a set of prerequisite execution properties to be annotated, and annotates the value for a single execution property. Second, *reshaping passes* modify the shape of IR DAGs by inserting, regrouping, or deleting IR vertices and edges, such as inserting vertices to collect runtime metric for dynamic optimizations. Furthermore, a *runtime pass* can also be performed during runtime for dynamic optimizations, such as data skew, using runtime statistics. This action occurs dynamically after the submitted job begins execution. It modifies the execution plan using the received runtime metric, so that the execution is dynamically optimized during runtime using the provided optimization logic specified by the user.

By utilizing a series of passes, users can optimize an application to exploit specific optimization techniques depending on job characteristics and resource environments, by choosing or adding appropriate configurations for the execution runtime. For example, in order to optimize an application to run on evictable transient containers, we can use a specialized *executor placement pass*, to place each operator appropriately on different types of resources, and a *dataflow model* pass to control the fashion in which each computation should fetch its input data, with a number of other passes for further optimizations.

This greatly simplifies the work by eliminating the burden of exploring and rewriting system internals for modifying pre-defined runtime behaviors. If an option for an execution property is already supported by the Onyx execution runtime, users can optimize their jobs with a simple process of using a new combination of optimization passes. If an implementation for a specific execution property is not supported, (e.g., a key-value store for a type of *data store*) a developer can easily add such implementation (explained in detail in Chapter 4) and configure applications to use the implementation by adding passes for IR DAGs.

3.3 Submitting to the Execution Runtime

Onyx IR is a logical representation of the job to be executed. It must be converted into a physical form for the execution runtime to execute. This involves translations like expanding an operator annotated with *parallelis*m in Onyx IR to the desired number of tasks and connecting the tasks according to the *data communication patterns* annotated on the IR edges. Physical execution plan is also in the form of a DAG, with the same values annotated for execution properties as the given IR DAG if necessary. Onyx IR DAG and physical execution plan can be translated from one another by sharing the identifiers. After a series of Onyx IR transformations through optimization passes, the compiler translates Onyx IR to a physical execution plan, submitting it to the runtime for execution.

Chapter 4

The Execution Runtime

4.1 Design

The main goal of the Onyx execution runtime is to support the diversity of job characteristics and resource environments with the least amount of implementation effort. To achieve this, the runtime must a) be flexible to easily control different execution properties and provide the options to users and b) exhibit a design that allows an easy extension to the execution properties. But at the same time, it should achieve such a design with the least amount of performance overhead.

First, we identify components that are inherent and essential to any data processing job and do not change. We implement these components as the backbone of our runtime, so that the code that needs no modifications is tightly implemented for better performance. Then, in order to make it a flexible and extensible system, we categorize the execution properties into 3 categories and take the necessary designs to handle each category:

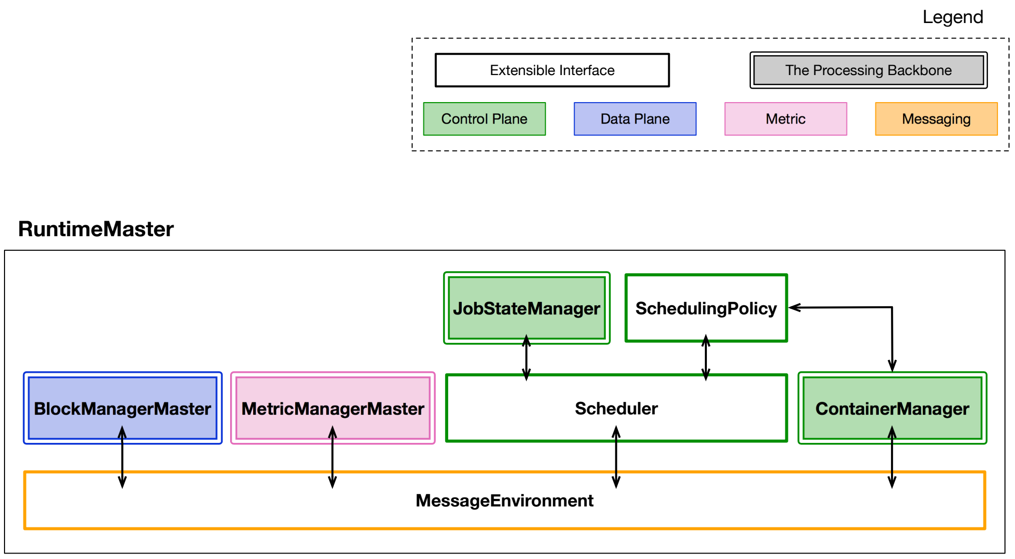
1. We identify execution properties that are subject to extensions - *executor placement*, *parallelism*(any integer), *data store*, *partitioner*, and *key assignment*. For example, a new executor type may be added. We may want to customize *partitioning* our data and use different *key assignment* methods accordingly. We may always want to add support for new files systems as our *data stores*. *Data store*,

partitioner, and *key assignment* are big and independent enough execution properties that can each be separated as modules. However, the rest of the execution properties are simple values or patterns that are only parts of processing a job in the backbone. We carry the values for each of these properties in an appropriate, extensible data structure, and use it in the backbone which has been implemented to execute jobs flexibly with values given in the data structure.

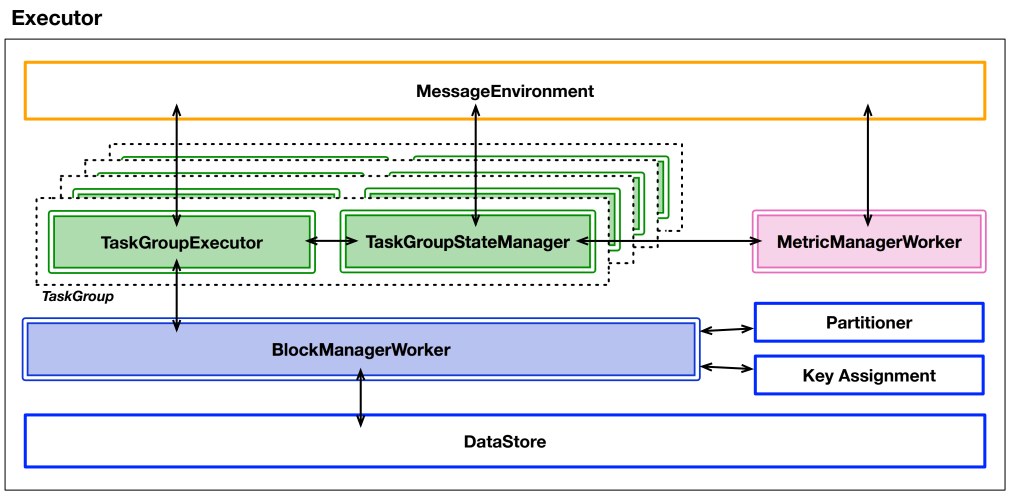
2. We identify execution properties that can be handled in many different ways during job execution - *executor placement*, *stage*, and *schedule group*. These properties can always be handled using new mechanisms, which we should be able to easily add to our runtime. For example, we may want to change the *executor placement* policy, or we may want to implement a new technique of scheduling using our *stage* and *schedule group* definitions. We expose the mechanisms to control these properties as a set of interfaces. New implementations can be added and integrated transparently to support other components of the job through these well-defined interfaces.

3. By this point, the remaining execution properties are *data communication pattern*, *dataflow model*, and *used data handling*. *Data communication pattern* has 4 options, many-to-many(shuffle), one-to-many(broadcast), many-to-one and one-to-one. *Dataflow model* only has 2 options, pull and push, and remains fixed. *Used data handling* has 2 options, discard and keep. These properties simply need to be flexibly configured. We tightly implement these properties in the backbone of the runtime. Even if implemented in the backbone, their runtime behaviors follow the physical execution plan translated from Onyx IR, which users can flexibly control (§3).

In the sections to follow, we use Figure 4.1 to describe the execution runtime's architecture which reflects the design choices made.



a) Runtime Master



b) Executor

Figure 4.1: The Execution Runtime Architecture. The double stroked, shaded boxes belong to the processing backbone. The single stroked, unshaded boxes are extensible modules integrated with other components through interfaces. Refer to the legend for more information.

4.2 The Processing Backbone

At the heart of the Onyx execution runtime is the processing backbone, denoted by the double stroked, shaded boxes in Figure 4.1. The backbone includes code that must be executed for all Onyx jobs, with references to the extensible data structures that carry user configured values for some of the execution properties as previously mentioned (§4.1).

Executor Management: Onyx is implemented on top of Apache REEF [19] to handle container management. `ContainerManager` uses REEF's APIs to launch an executor per container. When requesting for containers, Onyx uses user configurations to label appropriate containers with corresponding executor types. The labeled executor information maintained in `ContainerManager` reacts to the set of events like `onExecutorLaunched()` or `onExecutorRemoved()` propagated through REEF's APIs.

Computation: All Onyx jobs are composed of a set of stages. Each stage consists of one or more *task groups*. A *task group* is a computation unit composed of one or more tasks that can be computed in a single executor, distributed across executors to be conducted in parallel according to the `Scheduler` implementation of user's choice. Each executor runs a user defined number of task groups in parallel, each assigned to a separate thread. A task group is executed in `TaskGroupExecutor` essentially executing its list of tasks. A task executes the given user function by reading inputs and writing outputs through `BlockManagerWorker`.

Dataflow: `BlockManagerMaster` manages the locations of blocks and coordinates data transfers by communicating with `BlockManagerWorker` in each executor. `BlockManagerWorker` manages reading inputs and writing outputs of tasks executed on each executor. A task produces a block, which consists of one or more partitions as determined by the `Partitioner` implementation. A task consumes data from multiple producer tasks according to the *data communication pattern* annotated to the corresponding edge in the physical execution plan. From each producer task, the consumer task consumes one or more partitions as determined by the `KeyAssignment` implementation. Once the locations of the data to be written/read are determined, the actual data is written/read depending on the `DataStore` implementation.

Physical Data Transfer: Data transfers in data processing jobs can occur in various granularities of batches (e.g., a block all at once, element by element), and can flexibly be pulled by the consumers or pushed by the producers depending on the chosen *data flow model*. The two *data flow models* at multiple granularities have been implemented as a part of our lower data plane.

Execution States: During the entire job execution, the execution runtime maintains the computation states of tasks, task groups, and stages. `JobStateManager` manages the stage states by communicating with `Scheduler`. TaskGroup and task states are managed by receiving state change messages from `TaskGroupStateManager` created for each TaskGroup running on an executor. This facilitates the work of flexible job execution in a distributed environment as all executions must conform to the defined state transitions. Extending Onyx with new implementations for optimizations is also easier because all implementations as well as the backbone, must comply with these states, showing signs of inconsistent integration when state violations occur.

Metric Management: The processing backbone includes metric management with `MetricManagerWorker` on each executor and `MetricManagerMaster` in the master. The execution runtime collects system metric for each computation unit. Moreover, we leave application metric for customization and our processing backbone takes care of the propagation and the management of these metrics.

Dynamic Optimization: The processing backbone includes a dynamic optimization mechanism. It uses user customizable metrics that can be collected during runtime by inserting a metric collection operator to the job DAG (first, the IR DAG the translated to the physical execution plan) using a reshaping pass(§3.2). The collected metric is published back to the compiler where the job DAG is optimized according to the user defined runtime pass. When the optimized execution plan is received, the execution runtime continues executing the job, restarting from where it left off.

4.3 The Flexible and Extensible Execution Properties

- ***Executor Placement*** Users can label different containers with different specifications using a JSON format file. Onyx uses the JSON format file to build an extensible data structure named `ResourceSpecification` and uses this when maintaining executors for each type of `ResourceSpecification` in `ContainerManager`. The labels used for this data structure is shared with the compiler so that users can flexibly control *executor placements* using Onyx IR and optimization passes. In addition to simply having different types of executors, we want to be able to control and add new mechanisms to schedule specific operators to different types of executors. We achieve this by leaving the mechanism up to the `SchedulingPolicy` implementation. `SchedulingPolicy` (Listing 4.1) should communicate with `ContainerManager` to use labeled executors for different `ResourceSpecification` when scheduling task groups. The implemented methods `onExecutorAdded()` and `onExecutorRemoved()` would include the communication.

- ***Parallelism*** Controlling the *parallelism* of each operation simply occurs when the annotated IR DAG is translated to the physical execution plan, where an operator vertex is expanded to the number of tasks as specified by the annotated *parallelism*.

```java
/**
* Defines the policy by which {@link Scheduler} assigns task
    groups to executors.
*/
public interface SchedulingPolicy {

/**
* Returns this scheduling policy's timeout before an executor
    assignment.
* @return the timeout in milliseconds.
*/
long getScheduleTimeoutMs();

/**
* Attempts to schedule the given taskGroup to an executor
    according to this policy.
* If there is no executor available for the taskGroup, it
    waits for an executor to be assigned before it times out.
* (Depending on the executor's resource type)
*
* @param scheduledTaskGroup to schedule
* @return the ID of the executor on which the taskGroup is
    scheduled if successful, an empty Optional otherwise.
*/
Optional<String> attemptSchedule(final ScheduledTaskGroup
    scheduledTaskGroup);

/**
* Adds the executorId to the pool of available executors.
* Unlocks this policy to schedule a next taskGroup if locked.
* (Depending on the executor's resource type)
*
* @param executorId for the executor that has been added.
*/
void onExecutorAdded(String executorId);

/**
```

```
* Deletes the executorId from the pool of available executors.
* Locks this policy from scheduling if there is no more
    executor currently available for the next taskGroup.
* (Depending on the executor's resource type)
*
* @param executorId for the executor that has been deleted.
* @return the ids of the set of task groups that were running
    on the executor.
*/
Set<String> onExecutorRemoved(String executorId);

/**
* Marks the executorId scheduled for the taskGroup.
* Locks this policy from scheduling if there is no more
    executor currently available for the next taskGroup.
* (Depending on the executor's resource type)
*
* @param executorId of the executor assigned for the taskGroup.
* @param scheduledTaskGroup scheduled to the executorId.
*/
void onTaskGroupScheduled(final String executorId, final
    ScheduledTaskGroup scheduledTaskGroup);

/**
* Marks the taskGroup's completion in the executor.
* Unlocks this policy to schedule a next taskGroup if locked.
* (Depending on the executor's resource type)
*
* @param executorId of the executor where the taskGroup's
    execution has completed.
* @param taskGroupId whose execution has completed.
*/
void onTaskGroupExecutionComplete(String executorId, String
    taskGroupId);

/**
* Marks the taskGroup's failure in the executor.
* Unlocks this policy to reschedule this taskGroup if locked.
```

```
   * (Depending on the executor's resource type)
   *
   * @param executorId of the executor where the taskGroup's
       execution has failed.
   * @param taskGroupId whose execution has completed.
   */
  void onTaskGroupExecutionFailed(String executorId, String
      taskGroupId);
}
```

Listing 4.1: SchedulingPolicy.java

- ***Stage and Schedule Group*** Though a typical scheduler would execute a job stage by stage, `Scheduler` as shown in Listing 4.2, has no restrictions about how to schedule sub-parts of a job. Moreover, the given interface suggests that we can also define how scheduler should behave upon faults. Implementations of `Scheduler` can utilize task group states in `onTaskGroupStateChanged()` to define the fault tolerance mechanism for input read and output write failures, while container failures can be handled using `onExecutorRemoved()`. While leaving a lot of room for flexibility, we understand that this may lead to too much flexibility without much guide. This is the reason why we make *stage* partitioning and assigning *schedule group* flexible. Both *stages* and *schedule groups* can be flexibly adjusted in the compiler's optimization pass when annotating each vertex with these execution properties, allowing users control over scheduling even in typical situations like stage by stage scheduling. `updateJob()` is used for dynamic optimization of executing jobs. When actually selecting an executor to schedule a task group, `Scheduler` should coordinate with `SchedulingPolicy`. `SchedulingPolicy` (Listing 4.1) should look at *executor placements* and other factors like executor capacity when scheduling a task group in the `attemptSchedule()` method. A common implementation of a policy would be round-robin selection or locality-aware selection. Task group states should also be updated to `SchedulingPolicy` to update the task group assignment information through `onTaskGroupScheduled()`, `onTaskGroupExecutionComplete()`, and `onTaskGroupExecutionFailed()`.

- ***Data Store*** The execution runtime can store blocks in various types of stores. We define interfaces for `DataStore` and allow various implementations to be plugged in as modules. We provide three default options for `DataStore`: `MemoryStore`, `LocalFileStore` and `RemoteFileStore`. `MemoryStore` resides in the memory of the executor the producing task runs on. `LocalFileStore` saves data in the local file system of the executor the producing task runs on, in a serialized form. `RemoteFileStore` saves data elsewhere, in a remote file system. Such a remote file system may of a single data node, or may be a distributed file system. `DataStore` involves multiple interface and abstract classes including the simple and general `DataStore` APIs (e.g., `putBlock()`, `getBlock()`) with `FileStore` and `RemoteFileStore` APIs (e.g., `readFile()`, `writeBlockToFile()`), so we omit the specific classes for the scope of this paper.

```
public interface Scheduler {

/**
* Schedules the given job.
* @param physicalPlan of the job being submitted.
* @param jobStateManager to manage the states of the submitted
    job.
*/
void scheduleJob(PhysicalPlan physicalPlan,
JobStateManager jobStateManager);

/**
* Receives and updates the scheduler with a new physical plan
    for a job.
* @param jobId the ID of the job to change the physical plan.
* @param newPhysicalPlan new physical plan for the job.
* @param taskInfo pair containing the information of the
    executor id and task group to mark as complete after the
*             update.
*/
void updateJob(String jobId, PhysicalPlan newPhysicalPlan,
    Pair<String, TaskGroup> taskInfo);

/**
* Called when an executor is added to Runtime, so that the
    extra resource can be used to execute the job.
* @param executorId of the executor that has been added.
*/
void onExecutorAdded(String executorId);

/**
* Called when an executor is removed from Runtime, so that
    faults related to the removal can be handled.
* @param executorId of the executor that has been removed.
*/
void onExecutorRemoved(String executorId);

/**
* Called when a TaskGroup's execution state changes.
* @param executorId of the executor in which the TaskGroup is
    executing.
* @param taskGroupId of the TaskGroup whose state must be
    updated.
* @param newState for the TaskGroup.
* @param attemptIdx the number of times this TaskGroup has
    executed.
*************** the below parameters are only valid for
    failures *****************
* @param tasksPutOnHold the IDs of tasks that are put on hold.
    It is null otherwise.
* @param failureCause for which the TaskGroup failed in the
    case of a recoverable failure.
*/
void onTaskGroupStateChanged(String executorId,
String taskGroupId,
TaskGroupState.State newState,
int attemptIdx,
List<String> tasksPutOnHold,
TaskGroupState.RecoverableFailureCause failureCause);

/**
* To be called when a job should be terminated.
* Any clean up code should be implemented in this method.
*/
void terminate();
}
```

Listing 4.2: Scheduler.java

- ***Partitioner, Key Assignment and Data Communication Pattern***: All tasks read input data partitions and write output data blocks through `BlockManagerWorker`. For each edge between operators, `BlockManagerWorker` uses *data communication pattern* to understand the number of source and destination tasks. When resolving the partitions to write for the produced block, it refers to `Partitioner` as shown in Listing 4.3. `Partitioner` can use keys for the elements to control the number of partitions and the partitioning mechanism flexibly. A typical way of partitioning data would be to find the hash of the keys, modulo the number of destination tasks. When resolving the partitions to read given the source task index, `BlockManagerWorker` refers to the `KeyAssignment` implementation, which determines which *key assignments* (a range of partition keys) to read from.

```java
/**
* This interface represents the way of partitioning output
    data from a producer task.
* It takes an iterable of elements and divides the data into
    multiple {@link Partition}s.
*
* It can use the number of destination tasks as a guide when
    determining the partitioning mechanism.
*/
public interface Partitioner {

/**
* Divides the output data from a task into multiple partitions.
*
* @param elements the output data from a producer task.
* @param dstParallelism the number of destination tasks.
* @param keyExtractor extracts keys from elements.
* @return the list of partitions.
*/
List<Partition> partition(Iterable elements, int
    dstParallelism, KeyExtractor keyExtractor);
}
```

Listing 4.3: Partitioner.java

- ***Dataflow Model and Used Data Handling***: When `BlockManagerWorker` reads inputs or writes outputs through `DataStore`, it refers to the *dataflow model* to handle data appropriately in our lower data plane. `BlockManagerWorker` also takes care of used data according to the annotated configuration when transferring intermediate data is complete.

- **`MessageEnvironment`**: Although not related to flexibly running a job, we also open the `MessageEnvironment` interface for control message exchanges. Any underlying implementation that suits the user/system developers' needs can be used or added.

For all of the extensible modules with interfaces, the Onyx execution runtime includes one or more practical and common implementations. Those who only need flexibility requirements can configure their jobs to use the provided implementations. Others who need more and have extensibility requirements can use these implementations as guides when extending the system with new implementations.

Chapter 5

Examples

In this chapter, we demonstrate running real-world applications on Onyx using examples from Chapter 2. For each of these examples, we show how a job is configured flexibly with the job DAG (but in the form of Onyx IR which gets translated directly to the physical execution plan for simplicity). We then demonstrate how each job DAG is executed in the Onyx execution runtime.

5.1 Push Optimization for Small Scale Workloads

When we talk about batch data processing systems, we mainly focus on optimizing the systems for large scales as challenges mainly arise when processing them. But are large scale workloads the usual case? What if small scale workloads are actually more common in practice? Chen et al. analyze common industry MapReduce workloads in their work [11]. The workload traces are from various Hadoop deployments for real-life, business critical clusters (Cloudera and Facebook). One of their key contributions invalidates a major assumption of batch data processing - the prevalence of large scale workloads. Despite the expectation that most jobs will be at large scales, most jobs actually turn out to have input, shuffle and output sizes from a few MB to a GB. If workloads are small and the existing systems mainly target processing large scales, there must be numerous factors that can be optimized to process small scales of data. For example, a simple optimization would be to process the entire workload in memory instead of using any form of disk. This would require changing the *data store*. Another optimization technique would be to push the intermediate data to the consumer tasks before all the producer tasks complete execution, saving on data transfer time. Changing the *dataflow model* enables the consumer tasks to begin processing the arrived data while the producer tasks continue with their execution simultaneously.

Figure 5.1 shows the expression of a MapReduce job with and without push optimization in Onyx. a) shows the job DAG without push optimization. *Dataflow model* is "pull" and *data store* is "memory". This is how a job would be executed on Spark according to its pre-defined runtime behaviors. On the other hand, simply by changing the *dataflow model* execution property to "push" using an annotating pass (§3.2), users can execute a MapReduce job for small scale workloads with the IR DAG after push optimization shown in b). Users can assign *schedule group* to each operator, to enable map tasks and reduce tasks to be scheduled simultaneously for the intermediate data to be pushed.

c) shows the job execution of a) where the reduce tasks must wait until the last map task completes. They must wait even if the blocks of 2 complete map tasks are ready to be transferred. d) shows the job execution of b). Grouped by *schedule group* 0, all tasks can be scheduled simultaneously, and the reduce tasks can receive the blocks of 2 complete map tasks even if there is a map task that has not yet completed.

With Onyx, both of the job executions are easily configurable.

5.2 Harnessing Transient Resources: Pado

Pado (§2.3) proposes an optimization technique for harnessing transient resources in datacenters. Transient resources are prone to evictions. This implies that computations and data on these resources are lost and need to be recomputed upon evictions. In contrast, reserved resources are limited, but stable and thus the computations and data

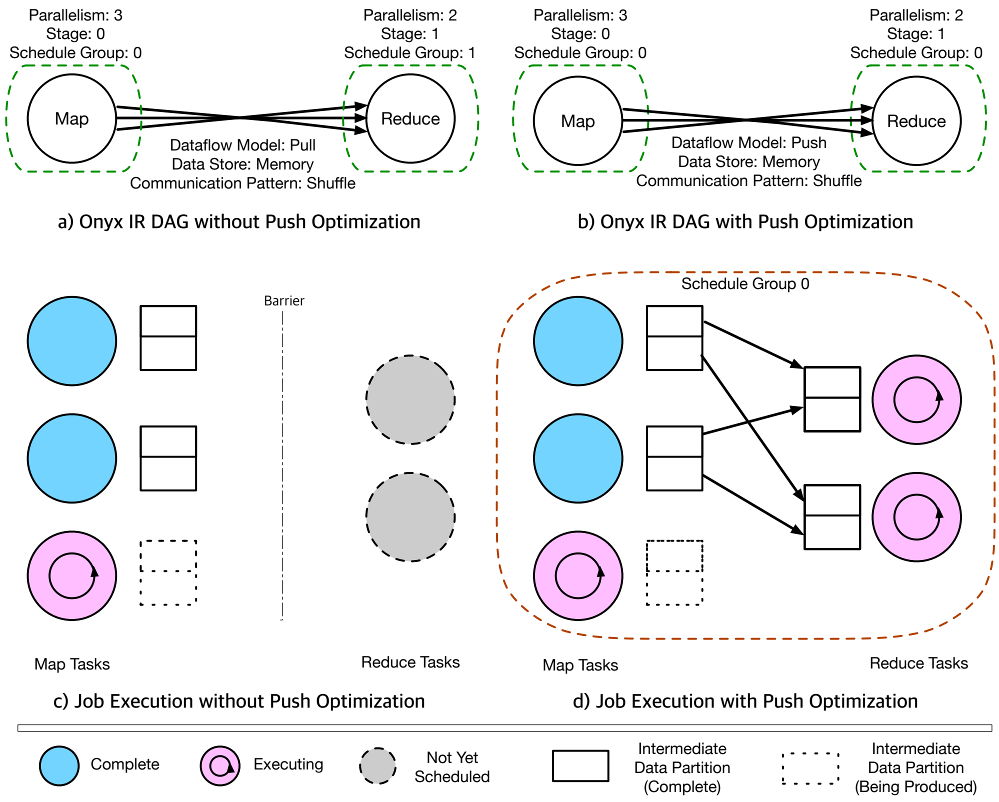


Figure 5.1: With and Without Push Optimization

residing on these resources are safe even during evictions. Pado's technique schedules the consumer tasks of ”expensive” dependencies (e.g., shuffle, where an eviction can cause many producer tasks to be recomputed) on reserved containers, and pushes data to the consumer tasks to evacuate data out of unsafe transient containers.

Figure 6.2 shows the expression of a MapReduce job with and without Pado optimization in Onyx. a) shows the job DAG without Pado optimization. Such a job execution is shown in c). Since the map stage takes *schedule group* 0 and the reduce stage takes *schedule group* 1, the reduce tasks are not scheduled until all map tasks complete. The map tasks can take any *executor placement*. As a result, 2 map tasks are executed in a transient container, while a map task is executed in a reserved container. When an eviction occurs, the output partitions of those run on the transient container are all lost, and need to be recomputed.

b) shows the job DAG with Pado optimization. Such a job execution is shown in d). Since both the map and reduce stages take *schedule group* 0, they can be scheduled simultaneously. The map tasks are scheduled to a transient container according to the *executor placement*. The reduce tasks are scheduled to a reserved container according to the *executor placement*. In order to evacuate data as soon as possible, we choose to push the intermediate data. As a result, even when an eviction occurs, the output partitions of those run on the transient container have already been transferred to the reserved container, and do not need to be recomputed.

Implementing the Pado optimization technique requires adding *executor placement* values, implementing a `SchedulingPolicy` that schedules different tasks to different *executor placements*. `SchedulingPolicy` should coordinate with `ContainerManager` which has already been implemented in an extensible manner to use the user configured *executor placement* values. Although not reflected in the figure, applications that are much more complex than simple MapReduce have complex job DAGs. In order to exploit the stability of reserved containers, Pado proposes

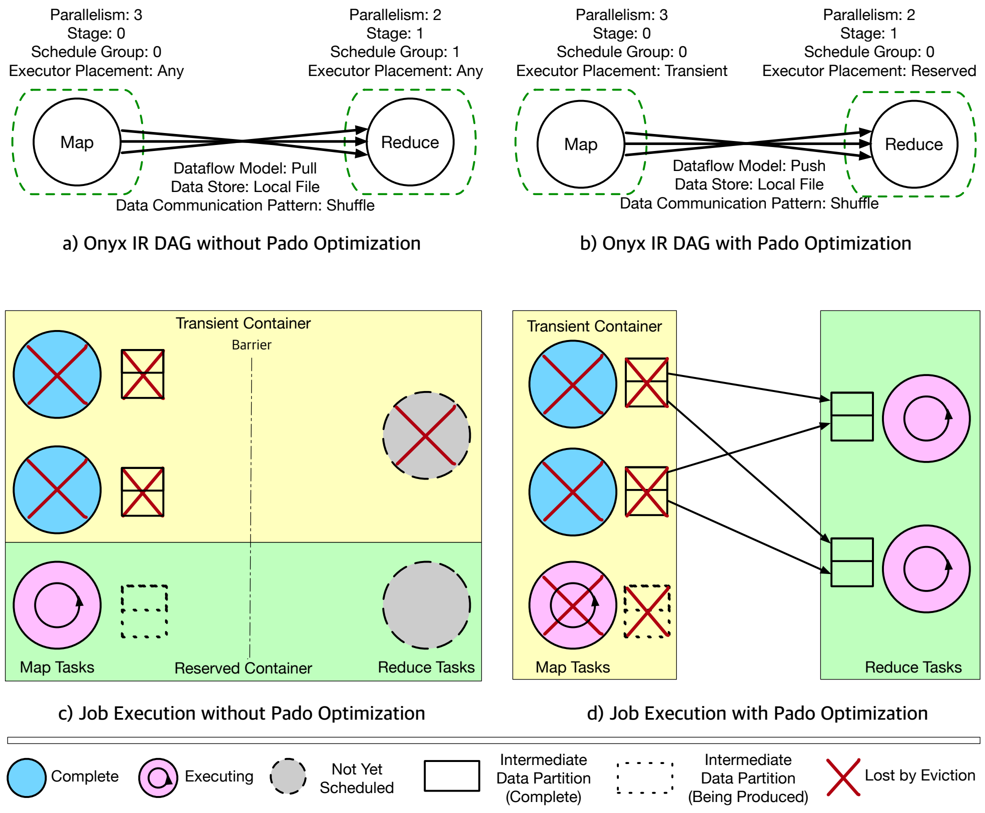


Figure 5.2: With and Without Pado Optimization

a stage partitioning mechanism which differs from traditional stage partitioning. A different stage partitioning mechanism can be simply added as an optimization pass, and the common `Scheduler` implementation in the execution runtime needs no modification.

Chapter 6

Evaluation

In this chapter, we evaluate Onyx by implementing the optimization techniques shown as examples in Section 5.1 and Section 5.2.

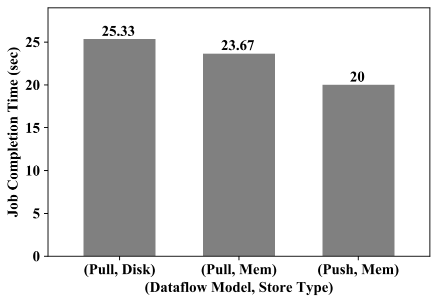
6.1 Small Scale Workloads

6.1.1 Experimental Setup

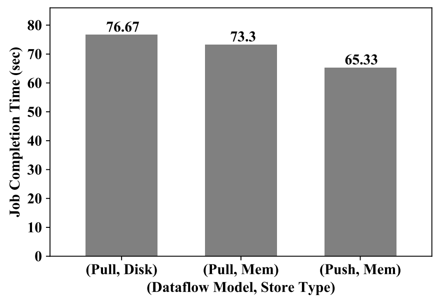
We ran MapReduce applications shown in Figure 5.1 on a YARN [20] cluster of 3 AWS EC2 instances. The 3 instances were of identical type, m3.xlarge, with 4 cores and 15GB of memory, running on a network of 1Gbps of measured bandwidth using the Iperf [21] tool. One of the instances was used as the master while the other two instances were each used to run a single executor with 4 threads. Input data of 4 different sizes, 10MB, 100MB, 300MB, and 1GB were used from Wikipedia's 2016 pagecounts dataset [22] for simple word count MapReduce jobs. For each input, 3 experiments were conducted: the "pull" *dataflow model* with 2 varying *data store* types - "local file" and "memory" - and the "push" *dataflow model* in "memory" *data store.*

6.1.2 Results

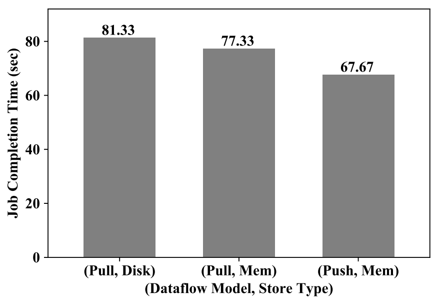
The experiment results in Figure 6.1 show that the push model performs better than the pull model for all of the small scale workloads chosen.



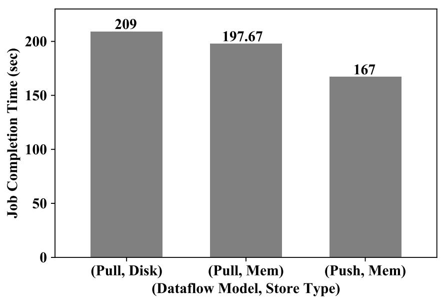
(a) 10MB



(b) 100MB



(c) 300MB



(d) 1GB

Figure 6.1: Job Completion Times for 4 Different Workload Scales. 3 runs for each workload are conducted: 1. *Dataflow model*: pull, *Data store*: local file, 2. *Dataflow model*: pull, *Data store*: memory, and 3. *Dataflow model*: push, *Data store*: memory. Job completions times take the least for the ”push” *dataflow model* for all workloads.

Table 6.1 shows the performance gains of using the push model in memory as compared to pulling the intermediate data from disk or memory. The push model using memory is upto approximately 21% faster than pulling intermediate from disk and 15% faster than pulling intermediate data from memory. One thing important to note here is that the push model can achieve an even better performance because the workload is small enough to entirely fit in memory, without incurring any disk overhead.

Table 6.1: Performance Gains of Using Push, Memory

Workload Size	over Pull, Local File	over Pull, Memory
10MB	21.05%	15.49%
100MB	14.78%	10.50%
300MB	16.80%	13.67%
1GB	20.00%	15.51%

6.1.3 Discussion

We investigated the performance of using the ”push” based *dataflow model* on common workload scales. Although the benefit is rather clear, there is still some room for further optimization. The granularities at which the intermediate data is transferred to the reduce tasks may vary. For simplicity, our implementation transfers intermediate data produced by each map task upon completion, but the performance may be enhanced if the transfer occurs at finer granularities, for example, by pipelining the intermediate data records between the map and reduce tasks. By pipelining intermediate data record by record, the combiner functions on the reduce tasks can begin their computation earlier, contributing to a larger performance gain in the push model. We are currently working on pipelining intermediate data, especially for the stream processing domain.

6.2 Harnessing Transient resources

6.2.1 Experimental setup

We ran an alternating least squares (ALS) application, a machine learning algorithm commonly used for recommendation systems. We used the 10GB Yahoo! music dataset on a cluster of 40 AWS EC2 m4.2xlarge instances with 8 cores and 32GB of memory. For this experiment, we ran the same application with the same dataset on the same cluster with Spark 2.2.0 for performance comparisons. Onyx used 35 transient containers and 5 reserved containers, a setting not easily configurable in Spark.

We ran ALS for two scenarios. The first scenario was without any eviction. For this scenario, we ran ALS once on Spark and once on Onyx without Pado optimization. The other scenario was with evictions. Executors running on transient containers were evicted at a 5-minute eviction arrival rate. The system was allowed to reclaim executors right after an eviction. For this scenario, we ran ALS once on Spark and once on Onyx with Pado optimization.

6.2.2 Results

Figure 6.2 shows the experiment results. When there is no eviction, Spark performs better than Onyx. Spark's pre-defined runtime behavior computes as much of intermediate data as possible in memory and flushes to disk before the executors run out of memory. Onyx currently does not have a `DataStore` that behaves this way, so we used the local file implementation for *data store*. We are currently working on this "as much as possible in memory, backed by disk" `DataStore` implementation. Once implemented, running the same experiments would be extremely easy, by using the different implementation instead of the local file implementation and changing the optimization pass to annotate the job DAG with the new value for *data store*. Despite the differences, Onyx yields a comparable performance. When evictions occur, Spark

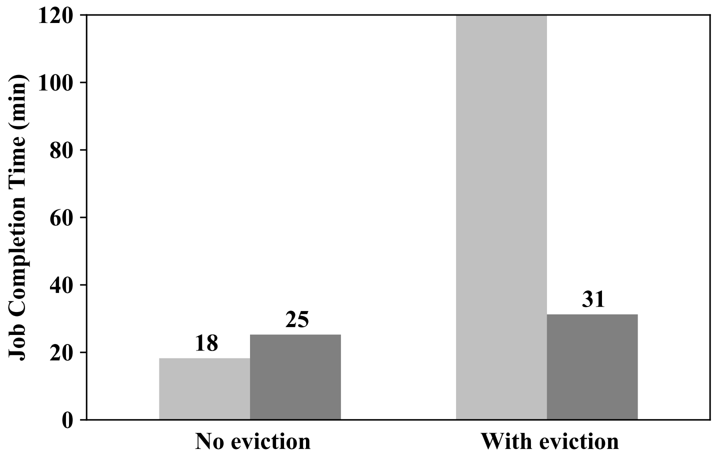


Figure 6.2: Comparison of Spark (light grey) and Onyx (dark grey) Job Completion Times for ALS.

stops making progress because it needs frequent recomputations of lost intermediate data. On the other hand, Onyx successfully implements the Pado optimization technique and completes the job in not much longer time than that of without evictions. We have cross-verified that the results are similar to those reported by the authors of Pado.

6.2.3 Discussion

Pado on Onyx has room for further improvements. Pado implements several other system optimizations such as task input caching and task output partial aggregation. *Caching* is an execution property we have identified (§2.4) but not yet implemented in the execution runtime. Once *caching* is supported, we expect to have additional `SchedulingPolicy` implementations that exploit cached data locality when scheduling tasks. It is important here again to note that other existing systems would require the monolithic system core to be modified even when adding a new feature, while it means simply plugging in the new implementation in the Onyx execution runtime.

Chapter 7

Conclusion

Each data processing system has its unique optimization techniques and benefits suitable for its main requirements. However, we have a diversity of job characteristics and resource environments with many distinct requirements. Optimization techniques for some requirements have been researched while those for other requirements remain as open questions. As gaining insights from data processing is becoming more and more popular, the diversity only continues to prosper. Systems with pre-defined runtime behaviors are not flexible enough to manage the growing diversity.

Onyx addresses this challenge by proposing and implementing a system design whose runtime behaviors can be flexibly determined. Moreover, Onyx's design enables the execution runtime to extended should new requirements arise. We identify the *execution properties* of data processing jobs that need to be provided as configurable knobs in order to flexibly control runtime behaviors.

Onyx IR plays an important role as a simple configuration tool to control the runtime behaviors by annotating computation vertices and dataflow edges with choices for execution properties. The Onyx execution runtime uses execution properties in its system design. It appropriately implements the core components in the processing backbone with options to control the behavior for the execution properties that only require flexibility. For other execution properties that can be extended, the execution runtime provides interfaces to be implemented for new requirements and integrates the extensible components transparently using the interfaces.

Onyx facilitates many users and system developers in applying desired optimization techniques to meet their requirements. We hope that Onyx's ability to flexibly leverage job executions would open up more opportunities for many more optimization techniques to be researched for diverse job characteristics and resource environments in the field of data processing.

Bibliography

[1] Y. Yang, G.-W. Kim, W. W. Song, Y. Lee, A. Chung, Z. Qian, B. Cho, and B.-G. Chun, "Pado: A data processing engine for harnessing transient resources in datacenters," in *EuroSys*, 2017.

[2] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda, "Tr-spark: Transient computing for big data analytics," in *SOCC*, 2016.

[3] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy, "Flint: Batch-interactive data-intensive processing on transient servers," in *EuroSys*, 2016.

[4] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves, "Sailfish: A framework for large scale data processing," in *SOCC*, 2012.

[5] "Facebook's disaggregated storage and compute for Map/Reduce." `https://atscaleconference.com/videos/facebooks-disaggregated-storage-and-compute-for-mapreduce/`.

[6] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker, "Network support for resource disaggregation in next-generation datacenters," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, p. 10, ACM, 2013.

[7] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation.," in *OSDI*, pp. 249–264, 2016.

[8] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, “Low latency geo-distributed data analytics,” *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 421–434, 2015.

[9] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, “Global analytics in the face of bandwidth and regulatory constraints.,” in *NSDI*, pp. 323–336, 2015.

[10] R. Viswanathan, G. Ananthanarayanan, and A. Akella, “Clarinet: Wan-aware optimization for analytics queries,” in *OSDI*, 2016.

[11] Y. Chen, S. Alspaugh, and R. Katz, “Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads,” *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1802–1813, 2012.

[12] “The CAIDA Anonymized Internet Traces.” `https://www.caida.org/data/`.

[13] Q. Ke, M. Isard, and Y. Yu, “Optimus: A dynamic rewriting framework for data-parallel execution plans,” in *EuroSys*, 2013.

[14] “Apache Hadoop.” `http://hadoop.apache.org`.

[15] “Spark.” `http://spark.apache.org`.

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *NSDI*, 2012.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: Distributed data-parallel programs from sequential building blocks,” in *EuroSys*, 2007.

[18] “Apache Beam.” `https://beam.apache.org/`.

[19] M. Weimer, Y. Chen, B.-G. Chun, T. Condie, C. Curino, C. Douglas, Y. Lee, T. Majestro, D. Malkhi, S. Matusevych, *et al.*, "Reef: Retainable evaluator execution framework," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1343–1355, ACM, 2015.

[20] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, p. 5, ACM, 2013.

[21] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs, "Iperf: The tcp/udp bandwidth measurement tool," *http://dast. nlanr. net/Projects*, 2005.

[22] "Wikipedia Pagecounts." `https://wikitech.wikimedia.org/wiki/Analytics/Archive/Data/Pagecounts-raw`.

국문초록

오늘날의 데이터 분석 애플리케이션들은 다양한 특성들을 가진다. 또한, 이 애플리케이션들은 다양한 자원 환경에서 수행됨으로써 각각이 뚜렷하게 구별되는 요구 사항들을 가지게 된다. 이런 요구 사항들을 만족시키기 위하여, 수많은 데이터 처리 시스템들이 개발되었고, 각 시스템들은 자신들이 목표로 하는 요구 사항에 최적화된 기법을 적용하였다. 하지만, 데이터 처리 분야는 다양한 작업 특성과 자원 환경에 맞춰 급격히 발전하고 있다. 현존하는 시스템들은 런타임의 수행 방식을 사전에 정의하도록 디자인되어 새로운 최적화 기법을 적용하기가 굉장히 어렵다. 이 문제를 해결하기 위해 개발된 Onyx 는 유연하고 확장성 있는 런타임을 디자인하고 구현한 시스템이다. Onyx 의 런타임은 유연하게 조정되고 확장 되어야 하는 수행 요소들을 파악하고, 해당 요소들을 중점으로 디자인 및 구현되어 사용자가 설정한 방식으로 작업이 수행되도록 하였다. 사용자는 작업의 표현 방식인 Onyx IR에 런타임 수행 방식을 제어하는 수행 요소들을 명시하여, 자신의 요구 사항에 맞게 런타임의 수행 방식을 설정할 수 있다. 이 논문안에 주어진 예제들과 실험 결과들을 통해 다른 기존의 시스템에서는 구현하기 힘든 새로운 최적화 기법들이 Onyx 에는 쉽게 적용될 수 있음을 확인할 수 있다.

주요어: 데이터 처리, 데이터 처리 엔진, 데이터 분석, 분석 엔진, 데이터 분석 엔진, 데이터 분석 프레임워크

학번: 2016-21194