IWFMS: An Internal Workflow Management System/Optimizer for Hadoop

Lian Liu, Yao Shen
Department of Computer Science and Engineering
Shanghai JiaoTong University
Shanghai, China
liustrive@gmail.com, yshen@cs.sjtu.edu.cn

ABSTRACT

The scale of jobs running on parallel computation platform such as Hadoop is increasing quickly, thus workflow engines that manage data processing jobs have become increasingly important. However, traditional workflow management systems are mostly outsiders to Hadoop and cannot fulfill many important requirements, such as user constraints and scheduling optimizations. In this paper, we present IWFMS, an internal workflow scheduling system/optimizer that 1) Manage the resource allocation and execution of jobs in workflows to achieve higher efficiency, 2) Schedule workflows to meet a much richer set of user constraints such as deadlines, priorities and workflow trigger events. We discuss the architecture of its key components and evaluate its features and performance.

KEYWORDS

IWFMS; Hadoop; Workflow Management; Optimization; User Constraints.

1 INTRODUCTION

Hadoop [1] is a massively scalable parallel computation platform capable of running many thousands of jobs per day. With the user group growing, big data in its raw form rarely satisfies the common user and Hadoop developer's data requirements for performing data processing tasks. Frameworks that help automate this increasingly complex process and codify work into repeatable units or workflows that can be reused over time without the need to write any new code or steps are essential layer of Hadoop for common user and more complicated series of work, such as Oozie[2], Azkaban[3]. These systems, known as workflow engines, provide a multi-tenant service that effectively manages diverse jobs written in a variety of tools and languages, and in the same time, has an interface which is user-friendly and quite convenient. These systems also allow users to specify constraints. For example, model workflow execution triggers in the form of the data, time or event predicates, the workflow job is started after those predicates are satisfied. However, these workflow engines are external workflow submitting systems which submit workflow as independent, unrelated jobs and provide very few execution optimizations. The cluster then schedule these jobs without considering the relationship between them, this design structure is a great restrict to the potential performance improvement of workflow execution, furthermore, the user constraints are also limited, supported trigger events are usually time intervals or data availability but cannot be complicated circumstances that involve cluster conditions or scheduling information. These factors have restricted the scenarios in which workflow engines could have been more widely used.

In this paper, we introduce a new management system and optimizer for workflow applications, the Internal Workflow Management System (IWFMS [4]). It allows users to build complex dependency structure like data transformations or decision branches out of multiple component jobs cost effectively, which inherit the superiorities of existed workflow engines, while at the same time, it provides a richer set of user
constraint options, increased system utilization and a significant reduction in the workflow completion time.

IWFMS is architected to work as a plug-in for Hadoop, unlike other existed workflow implementations, IWFMS provides service for both users and the clusters. Its core module, the WFScheduler, maintains the cluster information and helps optimize workflow jobs’ execution, clusters with IWFMS installed can recognize workflow jobs from ordinary jobs and schedule them according to their workflow structures and configurations.

IWFMS can also be used as a single optimizer which will collaborate smoothly with main workflow engines like Oozie. Organizations that have been using traditional workflow management systems may not want to change the user interface for interactive workflow management and APIs for integration, under these circumstances, they can abandon the user service part of IWFMS (although not recommended) and just plug WF Scheduler into Hadoop, this will provide most of the workflow execution optimizations from IWFMS and, with some manual modifications to the configuration file, full package of new user constraints.

This rest of this paper is organized as follows: in section 2 we explore related workflow systems and identify their shortcomings with respect to Hadoop requirements. Section 3 provides a brief introduction to workflow applications. Section 4 describes the design and implementation of IWFMS. In section 5 we present the experimental setup for measurement and discuss the results for IWFMS performance. Finally we summarize our current work and discuss our future work in section 6.

2 RELATED WORKS

Scientific workflow on clusters and grids have been studied extensively, many workflow management systems exist for various usage needs such as [5-8]. In this section, we discuss a few existing workflow management systems along with their benefits and shortcomings.

Oozie[2] is a Java Web application that combines multiple Hadoop jobs sequentially into one logical unit of work. There are two types of Oozie jobs: Oozie Workflow jobs and Oozie Coordinator jobs, the former are a sequence of actions that form a Directed Acyclic Graphs (DAGs), and the latter are recurrent workflow jobs that are triggered by time and data availability. Oozie makes it easy to control over complex jobs and repeat them at predetermined intervals. However, due to its complete separation from clusters, it’s more a workflow builder and autosubmitter than a full-function workflow management system. It considers little about Hadoop clusters conditions and provides none information about the workflow applications to Hadoop, what Oozie does is submitting any jobs as soon as they are ready to be executed, and then leave them to Hadoop without execution optimization towards their workflow structures and data dependency.

The Pegasus [9] system provides a comprehensive solution for constructing and enacting scientific workflows on the NCSA TeraGrid [10]. Pegasus enables the workflows to be executed locally in a simultaneous manner. It monitors the execution performance and provides mechanisms to implement optimization techniques. However, Pegasus seems to be not quite suitable for Hadoop clusters, and cannot fulfill the requirements under many scenarios.

Azkaban [3] is a batch workflow job scheduler created by LinkedIn to run their Hadoop jobs. It also provides an easy to use web user interface to maintain and track their workflow applications. Azkaban executes all workflow jobs as part of a single server process, which does not support authorization for job submission and control. It shares the same advantages of most existing management systems that are easy to use and user-friendly, but it provides even fewer options for user constraints than Oozie, and lacks significant features required for a workflow management system.
There are several other e-Science tools that help construct and execute workflows using local and/or remote data[11-14], in comparison, IWFMS is different from existing workflow systems in that part of it works inside Hadoop, this constitution breaks the obstacle between management system and Hadoop, which leaves far more space for optimization techniques and user customizations.

3 BACKGROUND

3.1 Workflow Application
A workflow application is a collection of actions (i.e. Hadoop Map/Reduce jobs, Pig jobs, scripts or executable files) arranged in a control dependency Direct Acyclic Graph (DAG), specifying a sequence of actions execution. A launched Hadoop workflow application consists of a series of submitted or uncommitted Hadoop jobs due to whether their dependencies are satisfied. A workflow engine usually in charge of construct the dependency DAG and set its triggers, when the triggers are satisfied, the workflow engine will start submitting jobs in the workflow application according to their DAG position. The DAG graph is specified in some kinds of descriptive language, in our implementation, we use hPDL, a XML Process Definition Language, which is almost the same as what Oozie uses, except that we add some new constraint nodes to the grammar.

3.2 hPDL Language
hPDL is a fairly compact language, it uses a limited amount of flow control and action nodes. Control nodes define the flow of execution, which includes mechanisms to control the workflow execution path like decision, fork or join operation, and beginning/end of a workflow such as start, end and fail nodes. Action nodes are the mechanism by which a workflow triggers the execution of a computation/processing job, the job can be written in a variety of tools and languages, e.g. Hadoop map-reduce jobs, executable files, shell or Python scripts and sub-workflows.

3.3 DAG Path
In this paper, DAG path concept is very important to our optimization techniques. A DAG path of the workflow application contains all the action nodes (jobs) along a directed path from the start node to the end node. A workflow job may belong to multiple paths, and a DAG path is called critical path in this paper if its jobs get the heaviest sum total workload. A workflow application’s completion time is decided by the duration of critical path.

4 FEATURES and IMPLEMENTATION
In this section, we first describe the overview of IWFMS's superiority and functions. Then we discuss how WFScheduler addresses its key functions in subsequent sub-sections.

4.1 Design Goals
Both the workflow jobs and ordinary jobs arrive randomly to the cluster, the former may be driven by WFEngine or external workflow engines like Oozie or Azkaban et al. When a job is submitted, by checking the distributed cache, we first decide whether it is a workflow job. If the job belongs to a workflow application, we read and analysis the configuration file along with the job submission file to get to know which workflow it belongs to and the position of the DAG structure it lies in, besides, if the job is the first job submitted of a workflow application, we also load and maintain the information of the workflow and the user-specified configuration, according to which we manage the workflow's lifecycle (such as sleep, invoke and launch) and optimize its execution. During the workflow execution, we observe the process rate and dynamically estimate the execution time of each DAG path to identify the critical path, then we adjust resource allocation and data distribution for jobs on each path to achieve a shortest overall execution time, which is determined by the critical path. As the completion time tends to be indicated by monitoring the processing cost of jobs on critical path, the resource needed for
each jobs of the workflow application can be computed to meet the deadline (if specified), one of the design goals is to maximize the number of workflow applications while satisfying the deadlines.

The design goals for IWFMS were: (1) Help Hadoop to recognize jobs submitted that belong to workflow applications, be aware of the workflow applications' DAG structures and adjust scheduling strategy accordingly. (2) Dynamically allocate resources in cluster to meet the given deadline of workflow applications based on the observed progress rate achieved by whose jobs. (3) Dynamically allocate resources for jobs on different DAG paths of a workflow application to achieve higher resource utilization and shortest total completion time. (4) Provide more trigger mechanisms by maintaining cluster status information and monitoring jobs and workflow applications.

4.2 IWFMS Architecture

IWFMS consists of two major parts, WFEngine and WFScheduler. WFEngine provides user interface for constructing and submitting workflow applications. WFScheduler is the kernel component for IWFMS to implement workflow execution optimizing mechanisms. WFScheduler is developed as a contrib module for Hadoop, and we have implemented various versions that support different job scheduling strategies for Hadoop such as Fair, Capacity etc. Figure 1 shows the basic architecture of the WFScheduler, which is divided into several modules. WFScheduler works as a middle layer between Hadoop JobTracker and Hadoop scheduler (FIFO, Fair, Capacity etc.), it detects jobs of workflow applications and manage them while does not influence the scheduling of ordinary jobs. We have separated the original Hadoop scheduling part from WFScheduler and make it a pluggable module, however due to the architecture of Hadoop itself, the WFScheduler, along with other scheduler contained, works together as a pluggable module for Hadoop.

**Figure 1 WFScheduler Internal Architecture**

The core modules of WFScheduler are:

- **Workflow Manager**: This module keeps track of submitted workflow applications, in charge of managing their lifecycles and resource allocation, it also responds to workflow trigger events by adding jobs of the workflow that was triggered to the scheduler initialize queue. The workflow queue maintains the information about all running and waiting workflow application, including node distribution and runtime condition.

- **Workflow ProgressRate Watcher**: This module monitor the progress rate of each running workflow applications, more specifically, it compute each path's progress rate of the workflow applications by observing the running cost of jobs on different paths, based on which it hereby estimate the completion time of critical path and resource needed for other paths.

- **Cluster Watcher**: This module keeps track of the cluster information. It watches cluster nodes conditions and job running stats, gathers important (or registered) events then send them to the core center, Workflow Manager, in which certain reaction will be determined.

- **hPDL parser**: The hPDL parser exists in both WFEngine and WFScheduler. It serves as a xml format hPDL language file parser which extract information needed
about a certain workflow application, such as, invoke conditions, dependency between jobs, workflow jobs information, user constraints etc.

- **Hadoop Scheduler:** The Hadoop scheduler works almost the same as traditional Hadoop scheduler, for instance, Fair Scheduler [15] and Capacity Scheduler [16]. The other parts of WFScheduler work as a middle layer between Hadoop JobTracker and Hadoop scheduler. In fact, most task schedulers for Hadoop, which serves as pluggable modules, can easily be modified into IWFMS’s Hadoop scheduler, thus, as mentioned above, IWFMS can support different job scheduling strategies for Hadoop.

### 4.3 Workflow Job Scheduling

The scheduling strategy of workflow jobs and traditional jobs differs in several aspects. There are various factors that may influence the resource allocation and running priority of a workflow job, for example, the use defined deadline of workflow application the job belongs to, the total resource needed for the workflow DAG path the job is in, distribution of input data and the dependency property. We take these factors into consideration in order to achieve higher execution efficiency.

#### 4.3.1 Workflow Deadline Estimation

To estimation the total duration of a workflow application we need to assess the completion time of each workflow paths. The critical path, whose jobs have the heaviest workload and longest total completion time, decides when the workflow application will finally complete.

In our implementation, to balance the progress rate of whole workflow, we need not only the duration of the critical path but also ones of each workflow path with certain amount of resources allocated. Accordingly, we can dynamically decide how many resources the currently running job on each DAG path need.

We develop an initial estimation model based a set of assumptions:

1) The cluster consists of homogeneous nodes, so that the time cost of processing for each map or reduce task is equal;

2) The time cost of uninitialized jobs’ task can be approximated by the average time cost of all map or reduce tasks on the DAG path;

3) Input data is either already available in HDFS or will successfully outputted by dependent job.

We extend the estimation model used by [17-18], and combine techniques proposed in [19] to it for workflow schedule environment. In our workflow estimation model, we introduce Hadoop specific notations $T, P, J, I, n_m, n_r, C_m, C_r, f_m, f_r, S$, as described below:

- $T = \{ T_{p1},...,T_{pK} \}$: The estimated complete time of path.

- $P = (J_1, J_2, ..., J_N)$: Workflow DAG paths, which consist of a series of jobs and connect the start node and the end node.

- $J = (t_{m1}, t_{m2}, ..., t_{mu}, t_{r1}, ..., t_{rv})$: A Hadoop workflow job that corresponds to an action node of the workflow DAG.

- $I = \{ I_{p1}, I_{p2}, ..., I_{pk} \}$: $I_{pi}$ represents input data of the first job of path $P_i$.

- $n_m, n_r$: Number of map/reduce slots assigned to job $i$.

- $C_m = \{ C_{mi} | i \in J \}$: Cost of processing a unit data in map task.

- $C_r = \{ C_{ri} | i \in J \}$: Cost of processing a unit data in reduce task.

- $f_m$: Map filter ratio. The fraction of input that the map process produces as output of job $i$.

- $f_r$: Reduce filter ratio. The fraction of input that the reduce process produces as output of job $i$.

- $S = \{ S_{i} | i \in J \}$: Start time of first map task for path $i$.

Our scheduling strategy is based on following expressions, in which $j$ means currently the cluster is scheduling the $j$-th job of path $k$:
\[ T_{P_k} = S^k + \sum_{i=0}^{j} I_i \prod_{l=1}^{i-1} f_i \prod_{j=1}^{i} C_m \prod_{l=1}^{i} C_r \prod_{m=1}^{i} \prod_{n=1}^{i} n_m \]

\[ + \sum_{i=N}^{j} I_i \prod_{l=1}^{N} f_i \prod_{j=1}^{N} C_m \prod_{l=1}^{N} C_r \prod_{m=1}^{N} \prod_{n=1}^{N} n_m \]

\[ + \sum_{i=+1}^{j} I_i \prod_{l=1}^{+1} f_i \prod_{j=1}^{+1} C_m \prod_{l=1}^{+1} C_r \prod_{m=1}^{+1} \prod_{n=1}^{+1} n_m \]

(1)

\[ T_{P_i} \approx T_{P_j} \quad \text{(where } P_i, P_j \in P) \]

(2)

\[ \text{availableSlots}_m = \sum_{k=0}^{K} n_m^k \]

(3)

\[ \text{availableSlots}_r = \sum_{k=0}^{K} n_r^k \]

(4)

In brief, we calculate how many resources a job on a workflow DAG can get based on the workload this path left, average process time and the total amount of resources this workflow application can get. When a job belongs to multiple DAG paths, the slots it can share is decided by the maximum number of its paths need dynamically.

4.3.2 Scheduling Algorithms

The optimization of workflow jobs is based on the following principles:

1) Jobs and their tasks on different paths get only enough resources they need so different paths of a workflow application end at similar time, which makes sure that the path with the heaviest workload get the most resources and highest running priority comparing to jobs on other paths of this workflow application.

2) Based on workflow DAG structure and jobs’ data dependency, tasks are scheduled to be data-local to the utmost.

3) Workflow applications that are in sleep state waiting for user specified events should release their resources and take only minimum space, by which approach the workload of scheduler will be kept in an acceptable level.

To achieve these goals, the WFScheduler sometimes need to temporarily hold certain resources and reserve them for the right job to be launched or initialized.

**Algorithm 1 Comparator<JobSchedulingInfo>**

**Input:** Job \( j_1 \), Job \( j_2 \)

**Output:** -1 or 1

1. if \( j_1 \in \text{WorkflowJob} \) and \( j_2 \in \text{WorkflowJob} \) then

2. if \( o_1.WfApp = o_2.WfApp \) then

3. \( pathInfo1 \leftarrow \text{getPathInfo}(j1) \)

4. \( pathInfo2 \leftarrow \text{getPathInfo}(j2) \)

5. if \( pathInfo1.\text{progressRate} < pathInfo2.\text{progressRate} \) then

6. return -1

7. else

8. if \( pathInfo1.\text{progressRate} > pathInfo2.\text{progressRate} \) then

9. return 1

10. else

11. return \( \text{compareStartTime}(j1,j2) \)

12. end if

13. end if

14. else

15. if \( j1.WfApp \in \text{eagerList} \&\& j2.WfApp \notin \text{eagerList} \) then

16. return -1

17. else

18. if \( j1.WfApp \notin \text{eagerList} \&\& j2.WfApp \in \text{eagerList} \) then

19. return 1

20. else

21. return \( \text{compareEagerness}(j1.WfApp, j2.WfApp) \)

22. end if

23. end if

24. end if

25. end if

26. \( \text{compareStartTime}(j1,j2) \);

27. end if

Shown in Algorithm 1, we implement a custom comparator to maintain the priority of slots allocation between jobs of same or different workflow applications. During the task assignment, we dynamically calculate the slots need for the next job to execute on the path and start to reserve slots before the current job ends, through this method, IWFMS ensures a significantly higher percentage of local-tasks.

Compared to the time saved of communication cost of intermediate data, the temporary
reservation cost can almost be ignored, which we will testify by experiments in Section 5. The task assignment and reservation strategy is presented in Algorithm 2, the pseudo-code describes the main idea of the techniques, while in practice, this strategy is actually implemented by multiple modules and is hard to be covered in one single function.

Algorithm 2

```java
addTasks(TaskTracker t, WSchedulerQueue queue)
```

1: if \( t \in \text{reserveList} \) then
2: \( \text{reserveInfo} \leftarrow \text{reserveList}(t) \)
3: \( \text{numReserved} \leftarrow \text{reserveSlots}(t, \text{reserveInfo}) \)
4: \( \text{reserveInfoneeded} \leftarrow \text{reserveInfoneeded} \leftarrow \text{numReserved} \)
5: if \( \text{reserveInfoneeded} == 0 \) then
6: remove (\( t, \star \)) from \text{reserveList}
7: end if
8: end if
9: \( \text{freeSlots} \leftarrow t.\text{freeSlots} \)
10: while \( \text{endOfQueue(queue)} \) \& \& \( \text{freeSlots} \neq 0 \)
11: \( j \leftarrow \text{nextJob(queue)} \)
12: if \( j \in \text{WorkflowJob} \)
13: // new workflow job launched
14: if \( j \in \text{reserveList} \) then
15: for \( \text{node} \in (j, \star) \) do
16: remove \( (t, \text{node}) \) from \text{reserveList}
17: end for
18: end if
19: \( \text{pathInfo} \leftarrow \text{getPathInfo}(j) \)
20: if \( \text{pathInfo.runningTasks} < \text{pathInfo.slotsNeeded} \)
21: \( \text{tasks} \leftarrow j.\text{getTasks()} \)
22: \( \text{freeSlots} \leftarrow \text{freeSlots} \leftarrow \text{tasks.size} \)
23: assignedTasks.append(tasks)
24: end if
25: if \( j.\text{timeLeft} < \text{threshold(pathInfo)} \) then
26: \text{reserveList.append}(\( \text{pathInfo.nextJobs, pathInfo.dataLocation} \))
27: end if
28: end if
29: end while
30: return assignedTasks
```

5 EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we evaluate the benefits of IWFMS, which contain the performance improvement of workflow applications and main IWFMS characteristics, including user customizations like deadline and triggers. We ran series of Hadoop workflow jobs that had unbalanced DAG structure and data dependency between jobs, which are common type of operation performed by MapReduce. Then we added various types of constraints like deadline, priorities and trigger events to the submitted workflow application.

5.1 Experimental Setup

Since building an industrial grade distributed system with thousands of machines is beyond the scope of our ability, to evaluate the performance of IWFMS, our experiments were conducted in an experimental cluster which consisted of 4 physical nodes with 3 as TaskTrackers and 1 as JobTracker. Each node specification was: 63GB main memory, 23 Intel Xeon 2.40GHz processors running Ubuntu. Hadoop version 1.2.1 was used in the cluster, along with Oozie version 3.2.0 on another node as baseline system. TaskTrackers had 8 map slots and 4 reduce slots, most configuration parameters were default values.

As baseline system, we installed Oozie version 3.2.0 on another machine which was used for submitting workflow jobs. In the baseline system, workflow jobs ran on the same cluster while without IWFMS installed.

5.2 Results

5.2.1 Performance Improvement

Figure 2 and Figure 3 shows the task allocation results for the same workflow application running in cluster with or without IWFMS. We can see that the tasks allocation is done quite differently, in IWFMS, jobs on light-workload path get limited amount of resources while ones on critical path can obtain more slots, the slots limits make each path of the workflow
application ends at similar time. On the other side, we notice that jobs running in traditional Hadoop system without IWFMS acquire slots based only on the time they are launched, which leads to a lower utilization rate and longer total time cost.

Another important reason for the performance speedup is distribution of slots allocated. In Hadoop with IWFMS, the majority of workflow tasks are data-local tasks, thus there are much fewer data transfer cost due to more local computations, which would benefit execution efficiency for both balanced and unbalanced workflow applications.

Table 1 Constituents of workflow applications

<table>
<thead>
<tr>
<th></th>
<th>Exp1</th>
<th>Exp2</th>
<th>Exp3</th>
<th>Exp4</th>
<th>Exp5</th>
<th>Exp6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balanced</td>
<td>100%</td>
<td>80%</td>
<td>60%</td>
<td>40%</td>
<td>20%</td>
<td>0%</td>
</tr>
<tr>
<td>Unbalanced</td>
<td>0%</td>
<td>20%</td>
<td>40%</td>
<td>60%</td>
<td>80%</td>
<td>100%</td>
</tr>
</tbody>
</table>

To evaluate the performance improvement further, we ran a series of workflow applications from Oozie examples that were slightly modified and had different DAG structures. Figure 4 illustrates that compared to baseline system; IWFMS exhibits 12%-36% makespan reduction.

5.2.2 Meeting workflow deadlines

In our next experiment we tested the effectiveness of workflow deadline constraint. Figure X shows the task allocation results for the same workflow application running on...
cluster with different deadlines or without deadlines specified. For 400s deadline, the task limit is 20; while for 800s deadline, maximum tasks allocated are limited to be no more than 10-12. In both cases with deadlines specified, the workflow deadline is met and the WFScheduler ensures that during the workflow applications execution, enough slots are met for each path of the workflow.

5.2.3 Other user constraints

![Figure 6: Task allocation with trigger event set](image)

Figure 6 shows the workflow execution when we set trigger event(s), in this case, the trigger event is when some jobs whose names contain one particular string are done, we can see that before the trigger event happened, the workflow application was asleep, and then the trigger event invoked it and WFScheduler added it to the running queue.

In fact, the trigger event configuration is quite flexible and can be easily set in the configure file submitted along with job files, for example, we can assign some workflow applications to be executed only when cluster is in light-workload, what we need to do is set the global configuration section in workflow.xml file. However, these user constraints cannot be satisfied using other workflow engines like Oozie and Azkaban, because they don’t maintain any information about the cluster and Hadoop, in fact, they don’t even stop submitting jobs when Hadoop is not in service.

6 CONCLUSION and FUTURE WORKS

In this paper, we introduce our implementation of workflow management system that has brand new design architecture. Unlike other workflow engines, it cares about when and where the workflow jobs will execute and knows about the conditions of Hadoop cluster. IWFMS provides us more efficient execution of workflow jobs, at the same time, workflow applications are no longer offline jobs with the applying of IWFMS. Future work includes designing and implementing a script language for the user constraint operations, through which users will gain much greater control to their workflow application.

REFERENCES

[10] Catlett, C. The philosophy of TeraGrid: building an open, extensible, distributed TeraScale facility.


