CoLoc: Distributed Data and Container Colocation for Data-Intensive Applications

Thomas Renner, Lauritz Thamsen, Odej Kao
Technische Universität Berlin, Germany
{firstname.lastname}@tu-berlin.de

***Abstract*—The performance of scalable analytic frameworks supporting data-intensive parallel applications often depends significantly on the time it takes to read input data. Therefore, existing frameworks like Spark and Flink try to achieve a high degree of data locality by scheduling tasks on nodes where the input data resides. However, the set of nodes running a job and its tasks is chosen by a cluster resource management system like YARN, which schedules containers without taking the location of data into account. Yet, the scheduling of the frameworks is restricted to the set of nodes the containers are running on. At the same time, many jobs in productive clusters are recurring with predictable characteristics. For these jobs, it is possible to plan in advance on which nodes to place a job's input data and execution containers.**

In this paper we present *CoLoc*, a lightweight data and container scheduling assistant for recurring data-intensive analytic jobs. CoLoc allows users to define related files that serve as input for the same job. It colocates related files on a set of nodes and offers this scheduling hint to the cluster manager to also place the jobs container on these nodes. The main advantage of CoLoc is a reduction of network transfers due to a higher data locality and locally performed operators like grouping or joining two or more datasets. We implement CoLoc on Hadoop YARN and HDFS, then evaluate it on a 40 node cluster using workloads based on Apache Flink and the TPC-H benchmark suite. Compared to YARN's default scheduler and HDFS's block placement scheduler, CoLoc reduces the execution time up to 35% for the tested data-intensive workloads.

***Index Terms*—Resource Management, Data Placement, Parallel Dataflows, Scheduling, Data-Intensive Applications**

I. Introduction

Gaining insights into the increasing volume of data is becoming relevant for more and more applications and businesses. As a result, various scalable analytic frameworks supporting data-intensive applications (i.e. jobs) have been developed over the last years. Prominent examples include MapReduce [1], Spark [2], and Flink [3], [4]. Often, jobs of these frameworks run side-by-side in containers, managed by a cluster resource management system like Mesos [5] or YARN [6]. Furthermore, the data that is to be analyzed typically resides in a colocated distributed file system such as HDFS [7]. This design is attractive for companies and data center providers, because it allows to run workloads consisting of different applications and using multiple frameworks on the same datasets. Thus, different users and organizations can share a cluster in a cost-effective manner [8] [9].

However, when the number of nodes and data-intensive jobs increase, the network can become a crucial factor for various reasons. For example, data residing in a distributed file system is stored in series of data blocks, which are distributed across all nodes. When the number of nodes increases, these data blocks are distributed across more nodes and thus, it is likely that a job's input data is not locally available and needs to be accessed from remote disks. For instance, it was reported that MapReduce jobs spent up to 59% of their runtime in map stages reading input data from remote disks [10]. Thus, most frameworks try to schedule tasks on nodes storing the input data and, thereby, provide high data locality. In cluster resource management systems, tasks are running in distributed containers scheduled by the cluster resource manager. However, existing cluster resource managers like YARN [6] allocate available resources to jobs without taking input data into account when launching containers. Therefore, the scheduling possibilities of jobs and, thus, the level of data locality is limited by the set of containers offered by the resource managers. In addition, data analytic frameworks such as Spark and Flink support data flow operations that can be network intensive, like joining or grouping. The more the related data is distributed across different nodes, the more data blocks need to be shuffled across the cluster. Previous work minimizes network demand and execution time by colocating data blocks of related files on the the same set of nodes [11]. However, not taking into account that tasks can be executed in containers.

At the same time, studies of productive clusters show that up to 40% of all jobs are recurring [12]–[14]. In these jobs, the execution logic of a job stays the same for every execution, but the input data is changing for every run. Recurring jobs are for instance triggered when new data becomes available or at a discrete time for further analysis (e.g. hourly or nightly batch jobs). In such scenarios, it is possible to decide where to store input data as well as containers before the job execution takes place.

In this paper we explore the problem of sharing compute and data resources between multiple data-intensive jobs. In particular, we present *CoLoc*, a lightweight scheduling assistant for Hadoop that helps to reduce the network demand of recurring jobs that run in shared data analytic clusters. CoLoc users can specify related files that, for instance, serve as input for a job. CoLoc then automatically identifies cluster nodes with sufficient resources to store all related data blocks and to host all necessary compute containers. The proposed placement strategy consists of two stages. First, a data colocation phase to place job input data on a specific set of nodes, instead of distributing them highly over all available nodes. CoLoc thereby also places blocks of related files on the same set of nodes, so that more operations can be performed locally. Second, a container colocation phase to place the job containers on the same set or subset of nodes, where most input data blocks of the first phase are stored. As a result, these jobs can benefit from a higher degree of data locality, so less data needs to be shipped over the network. CoLoc is implemented as a pluggable blockplacement policy and container scheduler for Hadoop and, thus, can be used by the different scalable data analytic frameworks that support YARN and HDFS.

Our evaluation is based on a 40 node cluster running Hadoop YARN, HDFS, and Flink as exemplary scalable data analytic framework. Our data-intensive workload is based jobs of the Transaction Processing Performance Council (TPC) benchmark H [15], a decision support benchmark that consists of a suite of business-oriented queries. We evaluated two different workload scenarios. First, a high cluster utilization scenario with many jobs running at the same time. Second, a mixed workload scenario consisting of different jobs submitted with some delay between jobs. CoLoc reduces completion time up to 34.9 percent for the first and 11.9 percent for the second scenario.

In summary, our contributions are as follows:

- An approach to reduce network utilization and execution time for recurring data-intensive data analytic workloads.
- An implementation of our approach as a scheduling assistant for Hadoop, which we call CoLoc and which provides data and container colocation.
- An evaluation using different data-intensive workloads consisting of different TPC-H benchmark queries on Flink.

The remainder of the paper is structured as follows. Section II presents the background for our research. Section III explains CoLoc's system architecture. Section IV discusses CoLoc's scheduling strategies in more detail. Section V presents results based on Flink and different workload combinations. Section VI summarizes the related work. Section VII concludes this paper.

II. Background & Motivation

This section describes the design of a data analytic cluster based on Hadoop that allows to run different frameworks and jobs side-by-side. Furthermore, we discuss the motivation for our data and container colocation approach.

A. Design of a Data Analytic Cluster Based on Hadoop

Many data analytic clusters are based on Hadoop [16]. Two main components of Hadoop are the cluster resource management system YARN [17] and the distributed file system HDFS [7]. As shown in Figure 1, both systems typically are running next to each other on all cluster nodes and follow the master/slave pattern. The HDFS master node (i. e. NameNode) monitors and coordinates data storage. YARN's master node (i. e. ResourceManager) monitors and coordinates computing functionalities. The slave nodes make up the majority of hosts and do the work of storing the data in series of distributed and replicated blocks (i. e. DataNode) and running different data analytic jobs within containers (i. e. NodeManager).

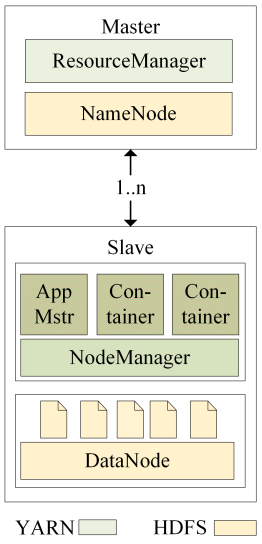

Fig. 1: Design of a data analytic cluster based on Hadoop [16].

In YARN, a user submits a job to the central ResourceManager, which is responsible for resource allocation and container scheduling. The ResourceManager is pluggable and allows different customizable scheduling algorithms, such as capacity or fair schedulers [18]. A container in YARN provides the execution environment for a job and is bound to a node running a NodeManager. Every job consists of one container running an ApplicationMaster (i.e. AppMstr) that coordinates and monitors all n execution containers of a job. After a job is finished, its containers are released and freed-up resources are available for upcoming jobs.

Data residing in the distributed file system HDFS is stored in series of replicated datablocks with a fixed size. The default HDFS configuration comes with a replication factor of three and places the first block locally on the writer node, the second on a random node whose rack is different from replica one, and the third on a random node who shares the same rack with the second replica. This block placement policy results in a high distribution of all data blocks across many nodes and, thus, provides high fault tolerance. However, the more the datablocks are distributed across different nodes, the more data needs be shuffled across the cluster for data access.

Distributed data analytic frameworks such as MapReduce [1], Spark [2], and Flink [3], [4] are often executed on top of a cluster resource management system and a distributed file system. In general, these frameworks process data through graphs of tasks (e.g. map, reduce, filter, and join). Each task is executed parallely and processes a partition of the data. Thus, a lot of data is split across various nodes, processed in parallel, transferred, and merged. Many frameworks try to exploit data locality by scheduling tasks close to the input data to maximize system throughput, because available network bandwidth is often lower than the clusters disk bandwidth [19]. However, the possibility of jobs running on top of cluster resource management systems to achieve data locality is restricted by the set of containers that has local access to the data.

B. Data and Container Colocation: A Motivating Example

CoLoc is a data and container placement strategy for recurring data-intensive jobs running in shared data analytic clusters. The main purpose is to place related files on a specific set of nodes instead of distributing all data blocks across all available cluster nodes. A partitioning or placing semantics for related files can be for example ”hourly” or ”daily” folders, which contain data with timestamps of that hour or day [20]. Afterwards, containers of a job that uses this data as input for, for example, hourly or nightly Extraction, Transformation, and Load (ETL) jobs are scheduled on the same set or subset of nodes.

Figure 2 illustrates CoLoc approach and shows eight nodes running a colocated HDFS for data block placement and YARN for container placement as well as two different jobs. The upper section of the figure shows the default data and container placement without CoLoc. For this case, data and containers of both jobs are distributed randomly across all available cluster nodes. The bottom section shows the colocation approach, where all input data and blocks of Job A are stored on node one to four. Note that Job A has two related files that are also colocated. Afterwards, all eight containers of Job A are scheduled on the same nodes. As a result, it is more likely to benefit from a higher degree of data locality, so that containers and task operators that run inside these containers access data from local disks. In addition, colocating data and placing it on the same set of nodes can improve the efficiency of dataflow operations like grouping and joining two or more datasets, because less data needs to be shipped over the network [11]. Also, it is more likely that containers of a job are scheduled on the same nodes and, thus, exchange more data locally.

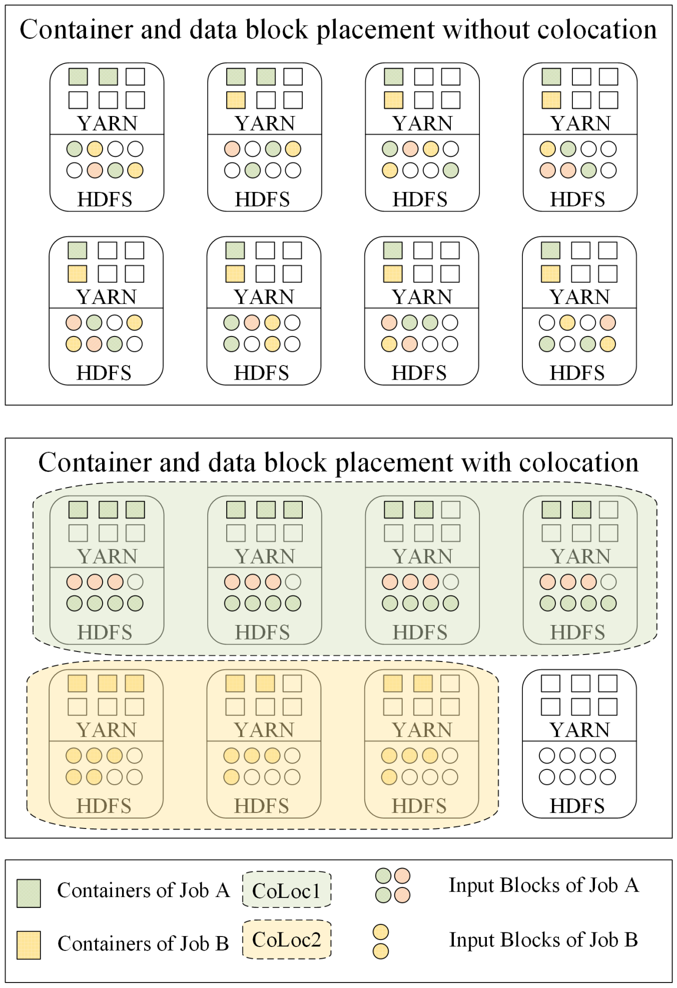
Container and data block placement without colocation
YARN
HDFS
Container and data block placement with colocation
Containers of Job A
CoLoc1
Input Blocks of Job A
Containers of Job B
CoLoc2
Input Blocks of Job B

Fig. 2: Comparison of container and data block placement with and without colocation.

III. System Overview

This section describes the system architecture and implementation details of CoLoc, a lightweight data and container placement assistant for Hadoop. As shown in Figure 3, CoLoc extends Hadoop’s scheduling capabilities with a Colocation Assistant, Block Placement Manager, and Container Placement Manager. The Block Placement and the Container Placement Manager are both implemented as pluggable Hadoop schedulers. Thus, CoLoc can be used without source code modification to Hadoop.

The *Scheduling Assistant* provides node hints as guideline to the underlying scheduler for data block and job container placement. Users and applications can define colocation entries, which describe files that are related and may be processed jointly by an upcoming job. For instance, a user could define that the files ’users.tbl’ and ’orders.tbl’ are related and should be stored on at least six data nodes. In addition, the Scheduling Assistant holds information about node locations of related files that are already stored in HDFS. With this information about related files and their locations, the Data Manager and the Container Manager are able to colocate related files and containers on the same set of nodes.

The *Block Placement Manager* is a pluggable block placement policy for HDFS. Whenever a user or application stores a file into HDFS, CoLoc’s Block Placement Manager requests the Scheduling Assistant and checks if a colocation entry for that file exists. If this is the case, the Block Placement Manager will try to colocate these files on the same set of nodes for improved efficiency. The mapping of specific nodes to a colocation entry is done, when the first file of an entry is stored into the system. The Block Placement Manager then chooses nodes with sufficient storage, stores the data blocks on it, and returns the locations as hints to the Scheduling Assistant. If no colocation entry exists, HDFS default block placement policy takes over. For instance, if a colocation entry for ’users.tbl’ and ’orders.tbl’ exists and ’users.tbl’ is already stored on HDFS, then CoLoc’s Block Placement Manager tries to place the data blocks of ’orders’ on the same set of nodes.

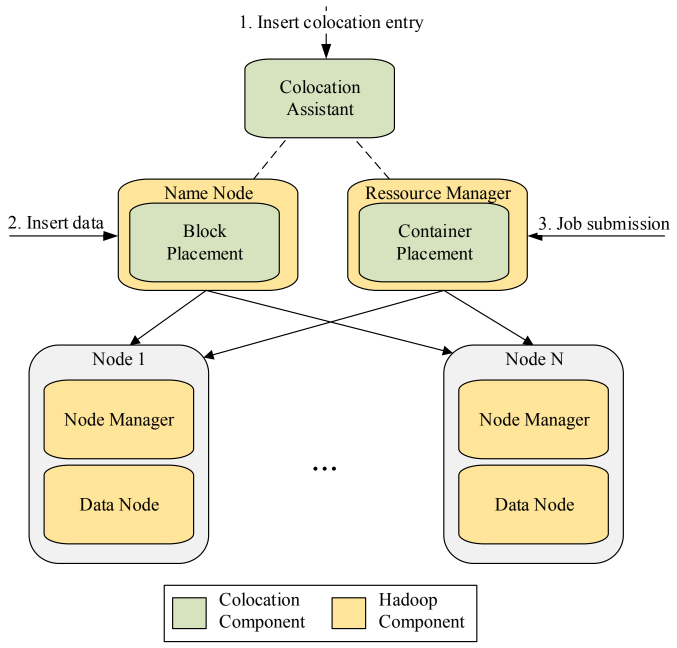

Fig. 3: System overview for distributed data and container colocation integrated into Hadoop.

The *Container Placement Manager* is a pluggable scheduler for YARN. When a job is submitted to the cluster, the Container Placement Manager requests the Scheduling Assistant whether a colocation entry for its job input data exists. If this is the case, the container scheduler receives hints about the nodes on which the data is stored and tries to place the container on the same set or subset of nodes. Besides file colocation, this improves data locality because most of the input data is stored on the nodes an execution container is assigned to.

IV. CoLoc Placement Strategy

This section describes CoLoc's two-stage placement strategy, a data and a container colocation stage, in more detail.

A. Data Colocation Stage

The general purpose of the data colocation stage is to colocate data blocks of related files that, for instance, serve as input for the same recurring job. In HDFS, a file f_i is stored in series of data blocks db_i with a fixed data block size, which are all replicated with a factor rep_i across all datanodes. Thus, formally a file can be defined as $f_i : \{db_{i_1}, db_{i_2}, ..., db_{i_n}\} * rep_i$. For recurring jobs, users usually know the inputs (i.e. related files) and the job size in terms of number of containers in advance. With this motivation, CoLoc allows users and applications to define two different colocation entries c_i with the following characteristics.

- *File Colocation* $c_i : \{fl_i, ns\}$, where fl_i: $\{f_1, f_2, ...f_n\}$ is a list of related files and ns is the minimum number of nodes (i.e. node size), on which all data blocks of the c_i entry will be distributed.
- *Folder Path Colocation* $c_i : \{fp_i, ns\}$, where fp_i is a unique folder path containing files fl_i that are stored under the path. These files are defined as related without knowing in advance how many files will be stored under this path.

The mapping of specific nodes to a colocation entry c_i is done when the first file of an entry is stored in the distributed file system. CoLoc's Block Placement Manager for HDFS then chooses the ns nodes with the most remaining disk space for balancing the cluster's disks usage. If there is not enough space on ns disks, the number of ns is automatically incremented until the nodes have enough disk space available. Afterwards, the data blocks are evenly assigned and stored on these selected nodes. In addition, the set of nodes is saved for upcoming files that belong to the previously stored file or Folder Path Colocation entry as Colocation Nodes cn_i with following characteristics:

- *Colocation Nodes* cn_i : $\{c_i, nl_i\}$, where nl_i: $\{n_1, n_2, ...n_n\}$ is a list of nodes, where all data blocks and containers of colocation entry c_i should be stored.

When a file fl_a with an existing File or Folder Path Colocation entry c_i and Colocation Nodes entry cn_i is put into HDFS, CoLoc tries to maximize the colocation of fl_a with all related and already stored data blocks of c_i. For example, file fl_a already exists in HDFS and a user stores file fl_b that should be colocated with fl_a. First, all data block locations of fl_a are requested from HDFS NameNode. Afterwards, for every data block of fl_2 that should be colocated into HDFS, a random location of the data block pool of fl_1 is selected. If both data blocks are not already assigned to the same node, the data block of fl_1 is stored on that node. Otherwise, another location is selected. If $fl_2 > fl_1$, the remaining data blocks are distributed randomly over all colocation nodes cn_i.

Figure 4 describes the data colocation staging process in more detail. It shows the interaction between a user that puts data into the distributed file system, the Block Placement Manager that is responsible for placing data blocks, and the Colocation Assistant. First, a user creates a colocation entry c_i for the file path fp 'click-logs/2016-01' with a node size ns of 6. Thus, CoLoc will store all data stored under the path on six nodes, which are determined when the first file is stored under the path 'click-logs/2016-01'. When the user now stores a file under the path '/click-logs/2016-01/', CoLoc's Block Placement Manager requests the Scheduling Assistant for a colocation entry with the path 'click-logs/2016-01'. At this step, three different cases are possible: First, there is no colocation entry and we use the HDFS default scheduling policy. Second, there is an entry and the assistant returns the nodes on which the block will be stored. Third, there is an entry but the associated nodes are not determined yet. In this case, which is shown in Figure 4, the Block Placement Manager will determine ns nodes with sufficient space, which are six in this example.

B. Container Colocation Stage

YARN's default Container Placement Manager allocates available resource to jobs randomly when launching containers. The idea of the container colocation stage is to place a

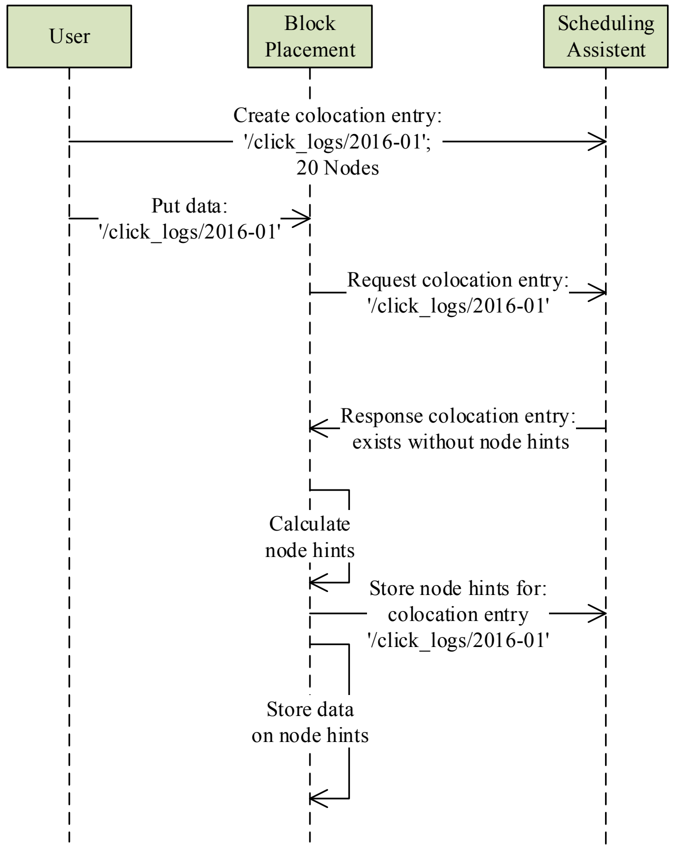
User
Block Placement
Scheduling Assistent
Create colocation entry: '/click_logs/2016-01'; 20 Nodes
Put data: '/click_logs/2016-01'
Request colocation entry: '/click_logs/2016-01'
Response colocation entry: exists without node hints
Calculate node hints
Store node hints for: colocation entry '/click_logs/2016-01'
Store data on node hints

Fig. 4: Sequence diagram illustrating the data colocation stage with an example.

job's containers on the set or subset of nodes, where most blocks of its input data are already stored. As a result, these jobs experience a higher data locality and, thus, read more input data from local disks, requiring fewer network transmissions. In addition, jobs can exchange more data between tasks that run within containers locally, if multiple containers of a job run side-by-side on the same set of nodes.

When a job is submitted to the cluster, CoLoc's container scheduler asks the Scheduling Assistant whether a colocation entry for its job's input data exists. If yes, the scheduler will receive location hints and place containers on that nodes. Therefore, the user can specify a colocation parameter that contains a list of files that are used by the submitted job. The container scheduler then asks the Colocation Assistant on which nodes these data are stored and returns node hints to the container scheduler. Next, the container scheduler tries to place all containers on the set of nodes. If not enough resources are available on these nodes, the remaining containers are distributed randomly across other available nodes with sufficient resources.

Figure 5 describes the container placement stage in more detail. First, the user submits a job to the YARN cluster in the default way. Afterwards, our pluggable yarn scheduler checks the submission parameter for HDFS paths and asks the Scheduling Assistant, if a matching colocation entry exists. If this is the case, the containers will be placed evenly on top of these nodes and, thus, more data can be read locality. If not, the containers will be placed with the default scheduling approach, similar to our HDFS block placement manager. It is important to mention that the approach is best effort. If a cell size is not big enough and there is not enough disk space or compute resources available for execution, an application can also be executed on other compute nodes or data stored on other data nodes.

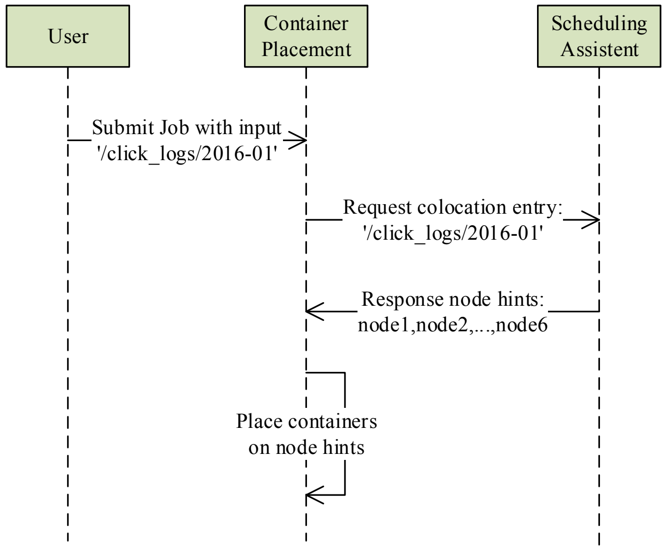
User
Container Placement
Scheduling Assistent
Submit Job with input '/click_logs/2016-01'
Request colocation entry: '/click_logs/2016-01'
Response node hints: node1,node2,...,node6
Place containers on node hints

Fig. 5: Sequence diagram that illustrates the container colocation stage, continuing the example of Figure 4

V. Evaluation

This section describes the evaluation of CoLoc using two different workload scenarios and a 40 node cluster. First, the experimental setup is described in more detail. Afterwards, we define the workloads and present the results of these experiments.

A. Experimental Setup

All experiments were done using a 40 node cluster. Each node is equipped with a quad-core Intel Xeon CPU E3-1230 V2 3.30GHz, 16 GB RAM, and three 1 TB disks with 7200RPM organized in a RAID-0. All nodes are connected through a single switch with a 1 Gigabit Ethernet connection. Each node runs Linux (kernel version 3.10.0), Java 1.8.0, Flink 0.10.1 and Hadoop 2.7.1. One additional node acts as master and the other 40 nodes as slave nodes. The master node runs Hadoop YARN's ResourceManager with our pluggable container placement manager and HDFS's NameNode with our Block Placement Manager as well as our CoLoc Scheduling Assistant for providing scheduling hints to both schedulers. The 40 slaves are responsible for workload execution and run YARN's NodeManager and HDFS's DataNode. We chose Flink as scalable data analytic framework and configured YARN to allocate 14 GB memory of a node and use 3 GB memory and four task slots per Flink TaskManager container as well as 1 GB per Application Master container, which runs Flink's JobManager.

B. Results of Different Workload Scenarios

The data-intensive workload in our experiments is based on benchmark queries for databases and transaction processing systems defined by the Transaction Processing Performance Council (TPC) [15]. In particular, we chose the TPC Benchmark suite H (TPC-H) Query 3 and 10, which are business oriented ad-hoc analytical queries. TPC-H examines large volumes of data and provides a data generator tool (i.e. dbgen). We used dbgen to generate a 250 GB input dataset for each job that we executed. For instance, if we executed eight different TPC-H queries, we generated eight different datasets with a size of 250 GB and stored them into HDFS, so that multiple jobs do not access the same dataset.

The minimum number of nodes ns per colocation entry was set to five. Thus, all data and containers of a colocated TPC-H job were placed on five nodes in our evaluation. Each of the experiments was done five times. We report the median execution time.

Scenario 1: High Cluster Utilization. In this scenario, we execute eight TPC-H queries that use all available compute resources of our cluster at the same time. Each job is accessing its own dataset stored in HDFS as input. Figure 6 compares two workload situations, (a) where all data and jobs are fully colocated with (b) the default scheduling behavior, in which no data and jobs are colocated. Each of the stacked boxes in Figure 6 represents the execution time of a single job. That is, we sum up all execution times as performance indicator. For the TPC-H Query 3 workload CoLoc reduces the execution time 34.88% and for the TPCH-10 workload 28.19% with CoLoc.

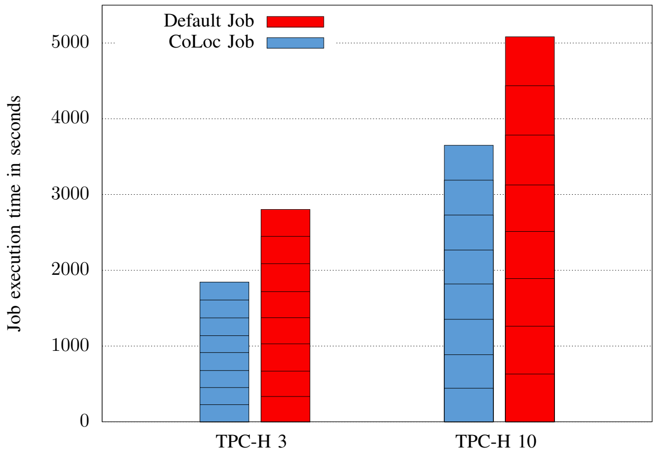

Fig. 6: Workloads consisting of eight colocated TPC-H Query 3 jobs improved by 34.88%, while workloads of eight colocated TPC-H Query 3 improved by 28.19%.

Figure 7 shows results for workloads consisting only of TPC-H Query 3 jobs (blue line) and only of TPC-H Query 10 jobs (red line). In both experiments, we changed the ratio between colocated and not colocated jobs for every run. For instance, in the first run, we executed eight jobs without colocation, while, in the second, we executed one with colocation and seven without colocation and so on. With the default scheduling behavior, the average execution time for a TPC-H Query 3 takes 353.38 seconds and for a TPC-H Query 10 635.37 seconds. Considering that up to 40% of jobs are recurring in productive clusters [12]–[14] and that in our experiments this share of recurring jobs is roughly given when the workload consists of three colocated and five not colocated jobs, CoLoc can reduce the execution time for 10.49% for the TPCH-3 and 10.86% for the TPCH-10 based workload.

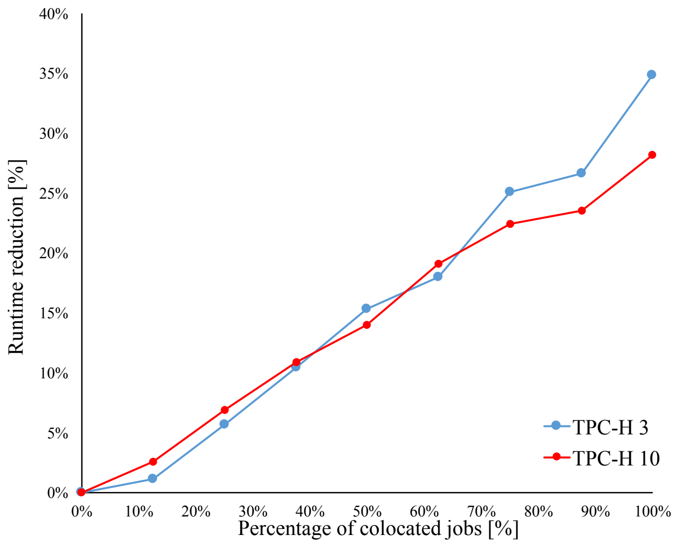

Fig. 7: Two Workloads consisting of eight TPC-H 3 and eight TPC-H 10 queries running in parallel with a varying numbers of colocated jobs.

Scenario 2: Delay and Mixed Workload. In this scenario, we run a mixed workload of four TPCH-3 and TPCH-10 queries. Always starting with a TPCH-3 query that was followed by a TPCH-10 query. In addition, we start executing a following job with a delay of one minute. In the first scenario, we started executing all jobs at the same time. Thus, we had clear network-intensive phases, in which all jobs are in the same stage such as at the beginning, when all jobs start reading the data from HDFS. Therefore, in this scenario, we run a more mixed workload and scheduled the jobs with delays of one minute, which provides a more realistic workload.

The results of the delay and mixed workload scenario are shown in Figure 8. As a performance indicator, we add up the execution times. With the default scheduling, the average execution time is 505.63 seconds. In the case, where all data and containers are colocated, the execution time decreases by 19.48% to 407.12 seconds. If taking into account that up to 40% of real cluster workloads are recurring [12]–[14], we decrease the time 7.27% to 468.88 seconds. We have less performance gain in comparison to workload scenario one, because less network phases are overlapping.

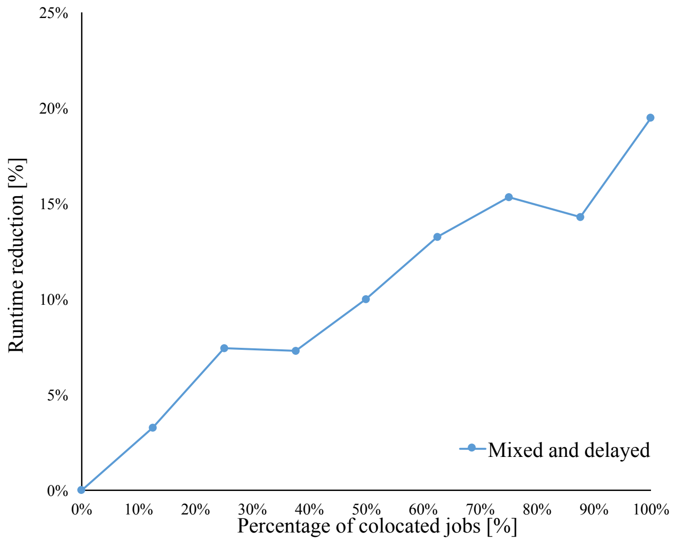

Fig. 8: Workloads consisting of four TPC-H Query 3 and four Query 10 queries running with a delay of 60 seconds and varying numbers of colocated jobs showing an up to 19.48% decreased execution time.

VI. Related Work

The techniques we used in CoLoc draw from a range of existing works in the context of scalable data analytic frameworks, clusters managers, and data placement techniques.

A. Cluster Resource Management Systems

YARN is the successor of MapReduce and decouples MapReduce's resource management and scheduling capabilities from the data processing component, enabling Hadoop to support multiple application frameworks for data processing, including frameworks like Spark or Flink. YARN's design also moves scheduling functions towards a per-job component. The current resource manager in YARN dynamically partitions the cluster compute resources among various jobs into different resource pools by letting the users specifying the required number and size of containers. Currently, YARN does not take data locality or network metrics into account. Mesos [5] is a similar system, also enabling users to run jobs from multiple dataflow frameworks efficiently in a single shared cluster. Mesos offers a scheduling mechanism, in which the central scheduler offers individual frameworks a number of nodes, while the frameworks decide which of these offers to accept and which tasks to run on these resources. Therefore, Mesos also delegates some of the scheduling logic to the frameworks. A key advantage of delegating container placement to processing systems is that the systems can optimize for goals like data locality with considerably more assumptions regarding the execution model. However, this leaves finding placements with good data locality to the framework. Custody [21] is a cluster resource management framework for Spark's standalone cluster manager that helps to maximize data locality by allocating Spark executors with local access to data to those jobs in need. However, the current implementation does not support multiple frameworks and does not take data placement for recurring jobs into account.

B. Scheduling for Data Analytic Frameworks

Cluster scheduling for data analytic frameworks has been an area of active research over the last years. Recent work has proposed techniques for increasing fairness across multiple tenants [18], [22]–[24], time-predictable resource allocation [25], improved data locality [8], [10], [26]–[29], or reduced interference in virtual environments [28]. However, most of these works focus on scheduling for a single application framework running in standalone. In addition, they assume data placement as given, and thus do not take data placement into account. On the contrary, CoLoc considers task execution in containers into account and tries to colocate them with its input data for recurring jobs.

C. Data Placement Techniques

Similar to our approach, CoHadoop [11] aims to colocate different datasets on the same set of nodes based on a user-defined property. However, it is limited to MapReduce. Techniques like Scarlett [10] and ERMS (Elastic Replica Management system) [30] use application access patterns to increase and decrease data replication factor in order to reduce job execution time. Coral [12] is a scheduling framework for recurring jobs that determines characteristics of future workloads to jointly place data and workers and isolate jobs by scheduling them in different parts of the cluster.

VII. Conclusion

In this paper we present the design, implementation and evaluation of a lightweight data and container scheduling assistant for recurring data-intensive analytic jobs called CoLoc. The main idea of CoLoc is to colocate data and containers for data-intensive jobs on the same set or subset of nodes. These jobs often run in cloud data centers managed by a cluster resource management system and read data from a distributed file system. Based on user provided hints on related file that, for instance, can serve as input data for a job, CoLoc first selects data nodes to colocate these data. Afterwards CoLoc schedules containers on these nodes, so that a job's execution can benefit from a higher level of data locality and, thus, less network traffic as well as execution time.

We implemented CoLoc in Hadoop as a pluggable resource manager for YARN and block placement policy for HDFS. Therefore, it can be used without Hadoop source code modification and is applicable for all application frameworks running on top of systems like MapReduce, Spark, or Flink. Our evaluation outperforms Hadoop's default strategy for data-intensive workloads based on different TPC-H benchmark queries running on Flink. For a high cluster utilization of simultaneously starting jobs up to 34.9% and for a workload of various and successively scheduled jobs up to 19.4%.

VIII. Acknowledgments

This work has been supported through grants by the German Science Foundation (DFG) as FOR 1306 Stratosphere and by the German Ministry for Education and Research (BMBF) as Berlin Big Data Center BBDC (funding mark 01IS14013A).

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[3] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl *et al.*, "The Stratosphere platform for big data analytics," *The VLDB JournalThe International Journal on Very Large Data Bases*, vol. 23, no. 6, pp. 939–964, 2014.

[4] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *IEEE Data Engineering Bulletin*, vol. 38, no. 4, pp. 28–38, July 2015.

[5] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," in *NSDI*, vol. 11, 2011, pp. 22–22.

[6] 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*. ACM, 2013, p. 5.

[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, May 2010, pp. 1–10.

[8] M. Zaharia, D. Borthakur, J. Sen Satodorma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 265–278.

[9] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, "Reservation-based Scheduling: If You're Late Don't Blame Us!" in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–14.

[10] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: coping with skewed content popularity in mapreduce clusters," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 287–300.

[11] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson, "Cohadoop: Flexible data placement and its exploitation in hadoop," *Proc. VLDB Endow.*, vol. 4, no. 9, pp. 575–585, Jun. 2011.

[12] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 407–420.

[13] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Reoptimizing data parallel computing," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 281–294.

[14] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "Scope: easy and efficient parallel processing of massive data sets," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1265–1276, 2008.

[15] "Tpc-h benchmark," http://www.tpc.org/tpch/ (accessed September 2016).

[16] "Apache hadoop," http://hadoop.apache.org (accessed September 2016).

[17] 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*. ACM, 2013, p. 5.

[18] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in *NSDI*, vol. 11, 2011, pp. 24–24.

[19] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI, "Making sense of performance in data analytics frameworks," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)(Oakland, CA*, 2015, pp. 293–307.

[20] L. Qiao, Y. Li, S. Takiar, Z. Liu, N. Veeramreddy, M. Tu, Y. Dai, I. Buenrostro, K. Surlaker, S. Das *et al.*, "Gobblin: Unifying data ingestion for hadoop," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1764–1769, 2015.

[21] S. Ma, J. Jiang, B. Li, and B. Li., "Custody: Towards data-aware resource sharing in cloud-based big data processing," in *Proceedings of IEEE Cluster 2016*, Taipei, Taiwan, 2016.

[22] C. Joe-Wong, S. Sen, T. Lan, and M. Chiang, "Multiresource allocation: Fairness–efficiency tradeoffs in a unifying framework," *Networking, IEEE/ACM Transactions on*, vol. 21, no. 6, pp. 1785–1798, 2013.

[23] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica, "Hierarchical scheduling for diverse datacenter workloads," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 4.

[24] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou, "Apollo: scalable and coordinated scheduling for cloud-scale computing," in *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2014.

[25] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao, "Reservation-based Scheduling: If You're Late Don't Blame Us!" in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–14.

[26] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The power of choice in data-aware cluster scheduling," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 301–316.

[27] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 261–276.

[28] X. Bu, J. Rao, and C.-z. Xu, "Interference and locality-aware task scheduling for mapreduce applications in virtual clusters," in *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '13. New York, NY, USA: ACM, 2013, pp. 227–238.

[29] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job scheduling for multi-user mapreduce clusters," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55*, 2009.

[30] Z. Cheng, Z. Luan, Y. Meng, Y. Xu, D. Qian, A. Roy, N. Zhang, and G. Guan, "Erms: An elastic replication management system for hdfs." in *CLUSTER Workshops*, 2012, pp. 32–40.