Intermediate Data Caching Optimization for Multi-Stage and Parallel Big Data Frameworks

Zhengyu Yang*, Danlin Jia*, Stratis Ioannidis*, Ningfang Mi*, and Bo Sheng†
*Dept. of Electrical & Computer Engineering, Northeastern University, 360 Huntington Ave., Boston, MA 02115
†Dept. of Computer Science, University of Massachusetts Boston, 100 Morrissey Boulevard, Boston, MA 02125
{yang.zhe, jia.da}@husky.neu.edu, {ioannidis, ningfang}@ece.neu.edu, shengbo@cs.umb.edu

***Abstract*—In the era of big data and cloud computing, large amounts of data are generated from user applications and need to be processed in the datacenter. Data-parallel computing frameworks, such as Apache Spark, are widely used to perform such data processing at scale. Specifically, Spark leverages distributed memory to cache the intermediate results, represented as Resilient Distributed Datasets (RDDs). This gives Spark an advantage over other parallel frameworks for implementations of iterative machine learning and data mining algorithms, by avoiding repeated computation or hard disk accesses to retrieve RDDs. By default, caching decisions are left at the programmer's discretion, and the LRU policy is used for evicting RDDs when the cache is full. However, when the objective is to minimize total work, LRU is woefully inadequate, leading to arbitrarily suboptimal caching decisions. In this paper, we design an algorithm for multi-stage big data processing platforms to adaptively determine and cache the most valuable intermediate datasets that can be reused in the future. Our solution automates the decision of which RDDs to cache: this amounts to identifying nodes in a direct acyclic graph (DAG) representing computations whose outputs should persist in the memory. Our experiment results show that our proposed cache optimization solution can improve the performance of machine learning applications on Spark decreasing the total work to recompute RDDs by 12%.**

***Keywords*—Cache Optimization, Multi-stage Framework, Intermediate Data Overlapping, Spark**

I. Introduction

With the rise of big data analytics and cloud computing, cluster-based large-scale data processing has become a common paradigm in many applications and services. Online companies of diverse sizes, ranging from technology giants to smaller startups, routinely store and process data generated by their users and applications on the cloud. Data-parallel computing frameworks, such as Apache Spark [1], [2] and Hadoop [3], are employed to perform such data processing at scale. Jobs executed over such frameworks comprise hundreds or thousands of identical parallel subtasks, operating over massive datasets, and executed concurrently in a cluster environment.

The time and resources necessary to process such massive jobs are immense. Nevertheless, jobs executed in such distributed environments often have significant computational overlaps: different jobs processing the same data may involve common intermediate computations, as illustrated in Fig. 1. Such computational overlaps arise naturally in practice. Indeed, computations performed by companies are often applied to the same data-pipeline: companies collect data generated by their applications and users, and store it in the cloud. Subsequent operations operate over the same pool of data, e.g., user data collected within the past few days or weeks. More importantly, a variety of prominent data mining and machine learning operations involve common preprocessing steps. This includes database projection and selection [4], preprocessing in supervised learning [5], and dimensionality reduction [6], to name a few. Recent data traces from industry have reported $40 \sim 60\%$ recurring jobs in Microsoft production clusters [7], and up to 78% jobs in Cloudera clusters involve data re-access [8].

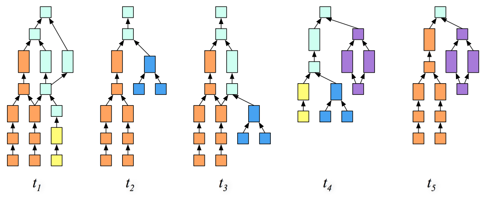

Fig. 1. **Job arrivals with computational overlaps.** Jobs to be executed over the cluster arrive at different times $t_1, \ldots, t_5$. Each job is represented by a Directed Acyclic Graph (DAG), whose nodes correspond to operations, e.g., map, reduce, or join, while arrows represent order of precedence. Crucially, jobs have *computational overlaps*: their DAGs comprise common sets of operations executed over the same data, indicated as subgraphs colored identically across different jobs. Caching such results can significantly reduce computation time.

Exploiting such computational overlaps has a tremendous potential to drastically reduce job computation costs and lead to significant performance improvements. In data-parallel computing frameworks like Spark, computational overlaps inside each job are exploited through caching and *memoization*: the outcomes of computations are stored with the explicit purpose of significantly reducing the cost of subsequent jobs. On the other hand, introducing caching also gives rise to novel challenges in resource management; to that end, the purpose of this paper is to design, implement and evaluate caching algorithms over data-parallel cluster computing environments.

Existing data-parallel computing frameworks, such as Spark, incorporate caching capabilities in their framework in a non-automated fashion. The decision of which computation results to cache rests on the developer that submits jobs: the developer explicitly states which results are to be cached, while cache eviction is implemented with the simple policy (e.g., LRU or FIFO); neither caching decisions nor evictions are part of an optimized design. Crucially, determining which outcomes to cache is a hard problem when dealing with jobs that consist of operations with complex dependencies. Indeed, under the Directed Acyclic Graph (DAG) structures illustrated in Fig. 1, making caching decisions that minimize, e.g., total work is NP-hard [9], [10].

In this paper, we develop an adaptive algorithm for caching in a massively distributed data-parallel cluster computing en-

This work was supported by CNS-1452751, CNS-1552525, CCF-1750539, and NeTS-1718355.

vironment, handling complex and massive data flows. Specifically, a mathematical model is proposed for determining caching decisions that minimize total work, i.e., the total computation cost of a job. Under this mathematical model, we have developed new *adaptive* caching algorithms to make online caching decisions with optimality guarantees, e.g., minimizing total execution time. Moreover, we extensively validate the performance over several different databases, machine learning, and data mining patterns of traffic, both through simulations and through an implementation over Spark, comparing and assessing their performance with respect to existing popular caching and scheduling policies.

The remainder of this paper is organized as follows. Sec. II introduces background and motivation. Sec. III presents our model, problem formulation, and our proposed algorithms. Their performance is evaluated in Sec. IV. Sec. V reviews related work, and we conclude in Sec. VI.

II. Background and Motivation

A. *Resilient Distributed Datasets in Spark*

Apache Spark has recently been gaining ground as an alternative for distributed data processing platforms. In contrast to Hadoop and MapReduce [11], Spark is a memory-based general parallel computing framework. It provides *resilient distributed datasets* (RDDs) as a primary abstraction: RDDs are distributed datasets stored in RAM across multiple nodes in the cluster. In Spark, the decision of which RDDs to store in the RAM-based cache rests with the developer [12]: the developer explicitly requests for certain results to persist in RAM. Once the RAM cache is full, RDDs are evicted using the LRU policy. Alternatively, developers are further given the option to store evicted RDDs on HDFS, at the additional cost of performing write operations on HDFS. RDDs cached in RAM are stored and retrieved faster; however, cache misses occur either because an RDD is not explicitly cached by the developer, or because it was cached and later evicted. In either case, Spark is resilient to misses at a significant computational overhead: if a requested RDD is neither in RAM nor stored in HDFS, Spark recomputes it from scratch. Overall, cache misses, therefore, incur additional latency, either by reading from HDFS or by fully recomputing the missing RDD.

An example of a job in a data-parallel computing framework like Spark is given in Fig. 2. A job is represented as a DAG (sometimes referred to as the *dependency graph*). Each node of the DAG corresponds to a parallel operation, such as reading a text file and distributing it across the cluster, or performing a map, reduce, or join operation. Edges in the DAG indicate the order of precedence: an operation cannot be executed before all operations pointing towards it are completed, because their outputs are used as inputs for this operation. As in existing frameworks like Spark or Hadoop, the inputs and outputs of operations may be distributed across multiple machines: e.g., the input and output of a map would be an RDD in Spark, or a file partitioned across multiple disks in HDFS in Hadoop.

B. *Computational Overlaps*

Caching an RDD resulting from a computation step in a job like the one appearing in Fig. 2 can have significant computational benefits when jobs may exhibit *computational overlaps*: not only are jobs executed over the same data, but

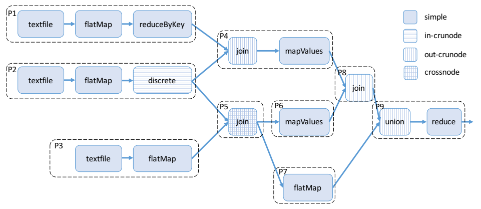

Fig. 2. **Job DAG example**. An example of a parallel job represented as a DAG. Each node corresponds to an operation resulting RDD that can be executed over a parallel cluster (e.g., a map, reduce, or join operation). DAG edges indicate precedence. Simple, crunodes (in/out) and cross nodes are represented with solid or lined textures.

also consist of operations that are repeated across multiple jobs. This is illustrated in Fig. 1: jobs may be distinct, as they comprise different sets of operations, but certain subsets of operations (shown as identically colored subgraphs in the DAG of Fig. 1) are (a) the same, i.e., execute the same primitives (maps, joins, etc.) and (b) operate over the same data.

Computational overlaps arise in practice for two reasons. The first is that operations performed by companies are often applied to the same data-pipeline: companies collect data generated by their applications and users, which they maintain in the cloud, either directly on a distributed file system like HDFS, or on NoSQL databases (like Google's Datastore [13] or Apache HBase [14]). Operations are therefore performed on the same source of information: the latest data collected within a recent period of time. The second reason for computational overlaps is the abundance of commonalities among computational tasks in data-parallel processing. Commonalities occur in several classic data-mining and machine learning operations heavily utilized in inference and prediction tasks (such as predictions of clickthrough rates and user profiling). We give some illustrative examples below:

Projection and Selection. The simplest common preprocessing steps are *projection* and *selection* [4]. For example, computing the mean of a variable age among tuples satisfying the predicate $\texttt{gender} = \texttt{female}$ and $\texttt{gender} = \texttt{female} \wedge \texttt{income} \geq 50\text{K}$ might both first reduce a dataset by selecting rows in which $\texttt{gender} = \texttt{female}$. Even in the absence of a relational database, as in the settings we study here, projection (i.e., maintaining only certain feature columns) and selection (i.e., maintaining only rows that satisfy a predicate) are common. For example, building a classifier that predicts whether a user would click on an advertisement relies upon first restricting a dataset containing all users to the history of the user's past clicking behavior. This is the same irrespective of the advertisement for which the classifier is trained.

Supervised Learning. Supervised learning tasks such as regression and classification [5], i.e., training a model from features for the purpose of predicting a label (e.g., whether a user will click on an advertisement or image) often involve common operations that are label-independent. For example, performing ridge regression first requires computing the covariance of the features [5], an identical task irrespective of the label to be regressed. Similarly, kernel-based methods like support vector machines require precomputing a kernel function across points, a task that again remains the same irrespective of the labels to be regressed [15]. Using either method to, e.g., regress the click-through rate of an ad, would involve the same preprocessing steps, irrespectively of the labels (i.e., clicks pertaining to a specific ad) being regressed.

Dimensionality Reduction. Preprocessing also appears in the form of *dimensionality reduction*: this is a common preprocessing step in a broad array of machine learning and data mining tasks, including regression, classification, and clustering. Prior to any such tasks, data is first projected in a lower dimensional space that preserves, e.g., data distances. There are several approaches to doing this, including principal component analysis [16], compressive sensing [6], and training autoregressive neural networks [17], to name a few. In all these examples, the same projection would be performed on the data prior to subsequent processing, and be reused in the different tasks described above.

To sum up, the presence of computational overlaps across jobs gives rise to a tremendous opportunity of reducing computational costs. Such overlaps can be exploited precisely through the caching functionality of a data-parallel framework. If a node in a job is cached (i.e., results are *memoized*), then neither itself nor any of its predecessors need to be recomputed.

C. Problems and Challenges

Designing caching schemes poses several significant challenges. To begin with, making caching decisions is an inherently combinatorial problem. Given (a) a storage capacity constraint, (b) a set of jobs to be executed, (c) the size of each computation result, and (d) a simple linear utility on each job, the problem is reduced to a knapsack problem, which is NP-hard. The more general objectives we discussed above also lead to NP-hard optimization problems [18]. Beyond this inherent problem complexity, even if jobs are selected from a pool of known jobs (e.g., classification, regression, querying), the sequence to submit jobs within a given time interval *is a priori unknown*. The same may be true about statistics about upcoming jobs, such as the frequency with which they are requested. To that end, a practical caching algorithm must operate in an *adaptive* fashion: it needs to make online decisions on what to cache as new jobs arrive, and adapt to changes in job frequencies.

In Spark, LRU is the default policy for evicting RDDs when the cache is full. There are some other conventional caching algorithms such as LRU variant [19] that maintains the most recent accessed data for future reuse, and ARC [20] and LRFU [21] that consider both frequency and recency in the eviction decisions. When the objective is to minimize total work, these conventional caching algorithms are woefully inadequate, leading to arbitrarily suboptimal caching decisions [9]. Recently, a heuristic policy [22], named "Least Cost Strategy" (LCS), was proposed to make eviction decisions based on the recovery temporal costs of RDDs. However, this is a heuristic approach and again comes with no guarantees. In contrast, we intend to leverage Spark's internal caching mechanism to implement our caching algorithms and deploy and evaluate them over the Spark platform, while also attaining formal guarantees.

III. Algorithm Design

In this section, we introduce a formal mathematical model for making caching decisions that minimize the expected total work, i.e., the total expected computational cost for completing all jobs. The corresponding caching problem is NP-hard, even in an offline setting where the popularity of jobs submitted to the cluster is *a priori known*. Nevertheless, we show it is possible to pose this optimization problem as a submodular maximization problem subject to knapsack constraints. This allows us to produce a $1 - 1/e$ approximation algorithm for its solution. Crucially, when job popularity is *not known*, we have devised an adaptive algorithm for determining caching decisions probabilistically, that makes caching decisions lie within $1 - 1/e$ approximation from the offline optimal, in expectation.

A. DAG/Job Terminology

We first introduce the terminology we use in describing caching algorithms. Consider a job represented as a DAG as shown in Fig. 2. Let $G(V, E)$ be the graph representing this DAG, whose nodes are denoted by V and edges are denoted by E. Each node is associated with an operation to be performed on its inputs (e.g., map, reduce, join, etc.). These operations come from a well-defined set of operation primitives (e.g., the operations defined in Spark). For each node v, we denote as $\texttt{op}(v)$ the operation that $v \in V$ represents. The DAG G as well as the labels $\{\texttt{op}(v), v \in V\}$ fully determine the job. A node $v \in V$ is a *source* if it contains no incoming edges, and a *sink* if it contains no outgoing edges. Source nodes naturally correspond to operations performed on "inputs" of a job (e.g., reading a file from the hard disk), while sinks correspond to "outputs". Given two nodes $u, v \in V$, we say that u is a *parent* of v, and that v is a *child* of u, if $(u, v) \in E$. We similarly define *predecessor* and *successor* as the transitive closures of these relationships. For $v \in V$, we denote by $\texttt{pred}(v) \subset V$, $\texttt{succ}(c) \subset V$ the sets of predecessors and successors of v, respectively. Note that the parent/child relationship is the opposite to usually encountered in trees, where edges are usually thought of as pointing away from the root/sink towards the leaves/sources. We call a DAG a *directed tree* if (a) it contains a unique sink, and (b) its undirected version (i.e., ignoring directions) is acyclic.

B. Mathematical Model

Consider a setting in which all jobs are applied to the same dataset; this is without loss of generality, as multiple datasets can be represented as a single dataset–namely, their union–and subsequently adding appropriate projection or selection operations as preprocessing to each job. Assume further that each DAG is a directed tree. Under these assumptions, let $\mathcal{G}$ be the set of all possible jobs that can operate on the dataset. We assume that jobs $G \in \mathcal{G}$ arrive according to a stochastic stationary process with rate $\lambda_G > 0$. Recall that each job $G(V, E)$ comprises a set of nodes V, and that each node $v \in V$ corresponds to an operation $\texttt{op}(v)$. We denote by as $c_v \in \mathbb{R}_+$ the time that it takes to execute this operation given the outputs of its parents, and $s_v \in \mathbb{R}_+$ be the size of the output of $\texttt{op}(v)$, e.g., in Kbytes. Without caching, the *total-work* of a job G is then given by $W(G(V, E)) = \sum_{v \in V} c_v$. We define the *expected total work* as:

$$\bar{W} = \sum_{G \in \mathcal{G}} \lambda_G \cdot W(G) = \sum_{G(V,E) \in \mathcal{G}} \lambda_{G(V,E)} \sum_{v \in V} c_v. \quad (1)$$

We say that two nodes u, u' are *identical*, and write $u = u'$, if both these nodes and all their predecessors involve exactly the same operations. We denote by $\mathcal{V}$ the union of all nodes of DAGs in $\mathcal{G}$. A *caching strategy* is a vector $x = [x_v]_{v \in \mathcal{V}} \in \{0, 1\}^{|\mathcal{V}|}$, where $x_v \in \{0, 1\}$ is a binary variable indicating whether we have cached the outcome of node v or not. As jobs in $\mathcal{G}$ are directed trees, when node v is cached, *there is no need to compute that node or any predecessor of that node.*

Hence, under a caching strategy x, the total work of a job G becomes:

$$W = \textstyle\sum_{v\in V} c_v(1-x_v)\prod_{u\in \mathtt{succ}(v)}(1-x_u). \tag{2}$$

Intuitively, this states that the cost c_v of computing $\mathtt{op}(v)$ needs to be paid if and only if *neither* v *nor* any of its successors have been cached.

C. Maximizing the Caching Gain: Offline Optimization

Given a cache of size K Kbytes, we aim to solve the following optimization problem:

MAXCACHINGGAIN

$$\text{Max:} \quad F(x) = \bar{W} - \sum_{G\in\mathcal{G}} \lambda_G W(G,x) \tag{3a}$$

$$= \sum_{G(V,E)\in\mathcal{G}} \lambda_G \sum_{v\in V} c_v \big[1 - (1-x_v)\prod_{u\in\mathtt{succ}(v)}(1-x_u)\big] \tag{3b}$$

$$\text{Sub. to:} \quad \textstyle\sum_{v\in\mathcal{V}} s_v x_v \le K, \quad x_v\in\{0,1\}, \text{ for all } v\in\mathcal{V}. \tag{3c}$$

Following [9], we call function $F(x)$ the *caching gain*: this is the reduction on total work due to caching. This offline problem is NP-hard [10]. Seen as an objective over the set of nodes $v \in \mathcal{V}$ cached, F is a *monotone, submodular* function. Hence, (3) is a submodular maximization problem with a knapsack constraint. When all outputs have the same size, the classic greedy algorithm by Nemhauser et al. [23] yields a $1 - 1/e$ approximation. In the case of general knapsack constraints, there exist well-known modifications of the greedy algorithm that yields the same approximation ratio [24]–[26].

Beyond the above generic approximation algorithms for maximizing submodular functions, (3) can be solved by *pipage rounding* [27]. In particular, there exists a concave function $L : [0,1]^{|\mathcal{V}|}$ such that:

$$(1-1/e)L(x) \le F(x) \le L(x), \quad \text{for all } x\in[0,1]^{|\mathcal{V}|}. \tag{4}$$

This *concave relaxation* of F is given by:

$$L(x) = \sum_{G(V,E)\in\mathcal{G}} \lambda_G \sum_{v\in V} c_v \min\big\{1, x_v + \sum_{u\in\mathtt{succ}(v)} x_u\big\}. \tag{5}$$

Pipage rounding solves (3) by replacing objective $F(x)$ with its concave approximation $L(x)$ and relaxing the integrality constraints (3c) to the convex constraints $x \in [0,1]^{|\mathcal{V}|}$. The resulting optimization problem is convex–in fact, it can be reduced to a linear program, and thus solved in linear time. Having solved this convex optimization problem, the resulting fractional solution is subsequently rounded to produce an integral solution. Several polynomial time rounding algorithms exist (see, e.g., [27], [28], and [26] for knapsack constraints). Due to (4) and the specific design of the rounding scheme, the resulting integral solution is guaranteed to be within a constant approximation of the optimal [26], [27].

D. An Adaptive Algorithm with Optimality Guarantees

As discussed above, if the arrival rates λ_G, $G\in\mathcal{G}$, are known, we can determine a caching policy within a constant approximation from the optimal solution to the (offline) problem MAXCACHINGGAIN by solving a convex optimization problem. In practice, however, the arrival rates λ_G may *not* be known. To that end, we are interested in an *adaptive* algorithm, that converges to caching decisions *without any prior knowledge of job arrival rates* λ_G. Building on [9], we propose an adaptive algorithm for precisely this purpose. We describe the details of this adaptive algorithm in our technical report [29]. In short, our adaptive algorithm performs *projected gradient ascent* over concave function L, given by (5). That is, our algorithm maintains at each time a fractional $y \in [0,1]^{|\mathcal{V}|}$, capturing the probability with which each RDD should be placed in the cache. Our algorithm collects information from executed jobs; this information is used to produce an estimate of the gradient $\nabla L(y)$. In turn, this is used to adapt the probabilities y that we store different outcomes. Based on these adapted probabilities, we construct a randomized placement x satisfying the capacity constraint (3c). We can then show that the resulting randomized placement has the following property:

Theorem 1: If $x(t)$ is the placement at time t, then $\lim_{t\to\infty}\mathbb{E}[F(x(t))] \ge \big(1-1/e\big)F(x^*)$, where x^* is an optimal solution to the offline problem MAXCACHINGGAIN (Eq. (3)).

The proof of Thm. 1 can be found in our technical report [29].

E. A Heuristic Adaptive Algorithm

Beyond attaining such guarantees, our adaptive algorithm gives us a great intuition to prioritize computational outcomes. Indeed, the algorithm prioritizes nodes v that have a high gradient component $\partial L/\partial x_v$ and a low size s_v. Given a present placement, RDDs should enter the cache if they have a high value w.r.t. the following quantity [29]:

$$\tfrac{\partial L}{\partial x_v}/s_v \simeq \big(\textstyle\sum_{G\in\mathcal{G}:v\in G}\lambda_G \times \Delta(w)\big)/s_v, \tag{6}$$

where $\Delta(w)$ is the difference in total work if v is not cached. This intuition is invaluable in coming up with useful heuristic

Algorithm 1: A Heuristic Caching Algorithm.

```
Procedure processJobs($\mathcal{G}$)
    $C_{\mathcal{G}}$ = Historical RDD access record;
    $C_G$ = Current job RDD access record;
    for $G \in \mathcal{G}$ do
        processJob($G(V,E)$, $C_G$);
        updateCache($C_G$, $C_{\mathcal{G}}$);
Procedure processJob($G(V,E)$, $C$)
    $C_G$.clear();
    for $v \in V$ do
        v.accessed=False;
        toAccess=set(DAG.sink());
        while toAccess$\neq \emptyset$ do
            v=toAccess.pop();
            $C_G$[v]=estimateCost(v);
            if not v.cached then
                for $u \in$ v.parents do
                    if not u.accessed then
                        toAccess.add(u);
            access(v); /* Iterate RDD $v$. */
            v.accessed=True;
    return;
Procedure estimateCost($v$)
    cost=compCost[v]; /* If all parents are ready. */
    toCompute=v.parents /* Check each parent. */
    while toCompute $\neq \emptyset$ do
        u=toCompute.pop();
        if not (u.cached or u.accessed or u.accessedInEstCost) then
            cost+=compCost[u];
            toCompute.appendList(u.parents);
            u.accessedInEstCost=True;
    return cost;
Procedure updateCache($C_G$, $C_{\mathcal{G}}$)
    for $v \in C_{\mathcal{G}}$ do
        if $v \in C_G$ then
            $C_{\mathcal{G}}[v] = (1-\beta)\times C_{\mathcal{G}}[v] + \beta \times C_G[v]$;
        else
            $C_{\mathcal{G}}[v] = (1-\beta)\times C_{\mathcal{G}}[v]$;
        updateCacheByScore($C_{\mathcal{G}}$);
    return;
```

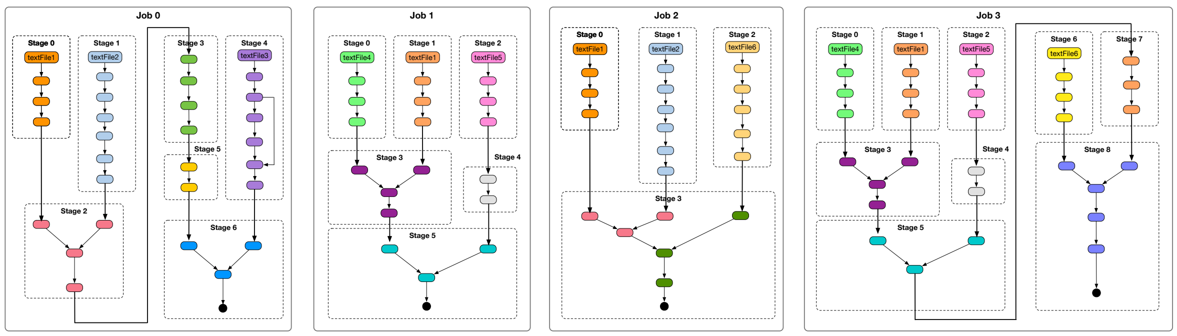

Fig. 3. **An example of RDD dependency in synthetic jobs.** Denote $J_x.S_y$ as stage y in job x, then we have $J_0.S_0 = J_1.S_1 = J_2.S_0 = J_3.S_1$, $J_1.S_{0\sim5} = J_3.S_{0\sim5}$, and $J_0.S_{0\sim1} = J_2.S_{0\sim1}$. Unfortunately, even sharing the same computational overlap, by default these subgraphs will be assigned with different stage/RDD IDs by Spark since they are from different jobs.

algorithms for determining what to place in a cache. In contrast to, e.g., LRU and LFU, that prioritize jobs with high request rate, Eq. (6) suggests that a computation should be cached if (a) it is requested often, (b) caching it can lead to a significant reduction on the total work, and (c) it has small size. Note that (b) is *dependent on other caching decisions made by our algorithm*. Observations (a), (b), and (c) are intuitive, and the specific product form in (6) is directly motivated and justified by our formal analysis. They give rise to the following simple heuristic adaptive algorithm: for each job submitted, maintain a moving average of (a) the request rate of individual nodes it comprises, and (b) the cost that one would experience if these nodes are not cached. Then, place in the cache only jobs that have a high such value, when scaled by the size s_v.

Alg. 1 shows the main steps of our heuristic adaptive algorithm. It updates the cache (i.e., storage memory pool) after the execution of each job (line 5) based on decisions made in the $updateCache$ function (line 6), which considers both the historical (i.e., $C_\mathcal{G}$) and current RDD (i.e., C_G) cost scores. Particularly, when iterating RDDs in each job following a recursive fashion, an auxiliary function $estimateCost$ is called to calculate and record the temporal and spatial cost of each RDD in that job (see line 14 and lines 22 to 31). Notice that $estimateCost$ does not actually access any RDDs, but conducts DAG-level analysis for cost estimation which will be used to determine cache contents in the $updateCache$ function. In addition, a hash mapping table is also used to record and detect computational overlap cross jobs (details see in our implementation in Sec. IV-C). After that, we iterate over each RDD's parent(s) (lines 16 to 18). Once all its parent(s) is(are) ready, we access (i.e., compute) the RDD (line 19). Lastly, the $updateCache$ function first updates the costs of all accessed RDDs to decide the quantities cost collected above with a moving average window using a decay rate of β, implementing an Exponentially Weighted Moving Average (EWMA). Next, $updateCache$ makes cross-job cache decisions based on the sorting results of the moving average window by calling the $updateCacheByScore$ function. The implementation of this function can (1) refresh the entire RAM by top score RDDs; or (2) evict lower score old RDDs to insert higher score new RDDs.

IV. Performance Evaluation

In this section, we first demonstrate the performance of our adaptive caching algorithm (Alg. 1) on a simple illustrative example. We then build a simulator to analyze the performance of large-scale synthetic traces with complex DAGs. Lastly, we validate the effectiveness of our adaptive algorithm by conducting real experiments in Apache Spark with real-world machine learning workloads.

A. *Numerical Analysis*

We use a simple example to illustrate how our adaptive algorithm (i.e., Alg. 1) performs w.r.t minimizing total work. This example is specifically designed to illustrate that our algorithm significantly outperforms the default LRU policy used in Spark. Assume that we have 5 jobs (J_0 to J_4) each consisting of 3 RDDs, the first 2 of which are common across jobs. That is, J_0's DAG is $R_0 \rightarrow R_1 \rightarrow R_2$, J_1 is $R_0 \rightarrow R_1 \rightarrow R_3$, J_2 is $R_0 \rightarrow R_1 \rightarrow R_4$, etc. The calculation time for R_1 is 100 seconds while the calculation time for other RDDs (e.g., R_2, R_3,...) is 10 seconds. We submit this sequence of jobs twice, with the interarrival time of 10 seconds between jobs. Thus, we have 10 jobs in a sequence of $\{J_0, J_1, ..., J_4, J_0, J_1, ..., J_4\}$. We set the size of each RDD as 500MB and the cache capacity as 500MB as well. Hence, at most one RDD can be cached at any moment.

Table I shows the experimental results of this simple example under LRU and our algorithm. Obviously, LRU cannot well utilize the cache because the recently cached RDD (e.g., R_2) is always evicted by the newly accessed RDD (e.g., R_3). As a result, none of the RDDs are hit under the LRU policy. By producing an estimation of the gradient on RDD computation costs, our algorithm instead places R_1 in the cache after the second job finishes and thus achieves a higher hit ratio of 36%, i.e., 8 out of 22 RDDs are hit. Total work (i.e., the total calculation time for finishing all jobs) is significantly reduced as well under our algorithm.

TABLE I. Caching results of the simple case.

Policy	J_0	J_1	J_2	J_3	...	J_4	hitRatio	totalWork
LRU	R_2	R_3	R_4	R_5	...	R_6	0.0%	1100
Adaptive	R_2	R_1	R_1	R_1	...	R_1	36.4%	300

B. *Simulation Analysis*

To further validate the effectiveness of our proposed algorithm, we scale up our synthetic trace by randomly generating a sequence of 1000 jobs to represent real data analysis applications with complex DAGs. Fig. 3 shows an example of some jobs' DAGs from our synthetic trace, where some jobs include stages and RDDs with the same generating logic chain. For example, stage 0 in J_0 and stage 1 in J_1 are identical, but their RDD IDs are different and will be computed twice. On average, each of these jobs consists of six stages and each stage has six RDDs. The average RDD size is 50MB. We use a decay rate of $\beta = 0.6$.

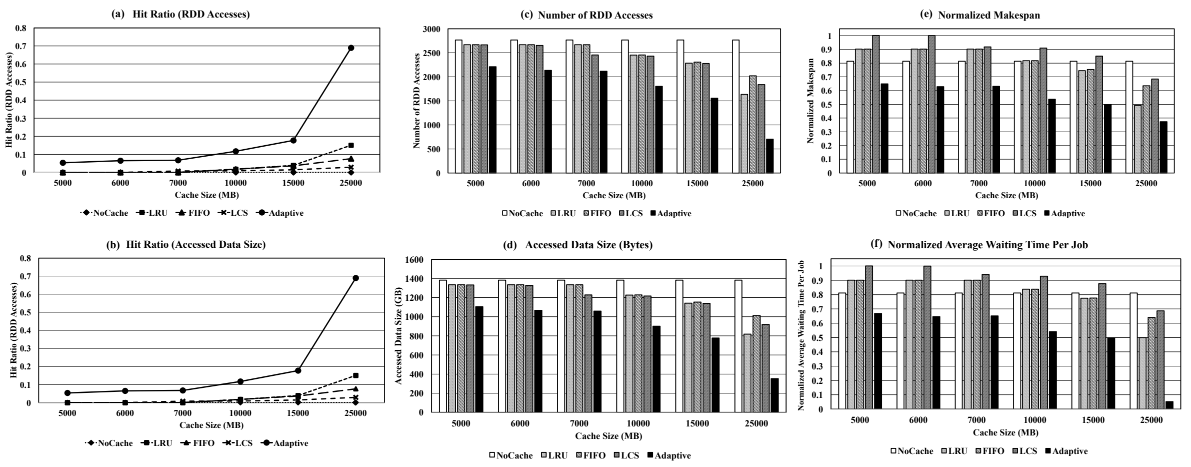

Fig. 4. Hit ratio, access number and total work makespan results of large scale simulation experiments.

We implement four caching algorithms for comparison: (1) NoCache: a baseline policy, which forces Spark to ignore all user-defined *cache/persist* demands, and thus provides the lower bound of caching performance; (2) LRU: the default policy used in Spark, which evicts the least recent used RDDs; (3) FIFO: a traditional policy which evicts the earliest RDD in the RAM; and (4) LCS: a recently proposed policy, called “Least Cost Strategy” [22], which uses a heuristic approach to calculate each RDD’s recovery temporal cost to make eviction decisions. The main metrics include (a) *RDD hit ratio* that is calculated as the ratio between the number of RDDs hit in the cache and the total number of accessed RDDs, or the ratio between the size of RDDs hit in the cache and the total size of accessed RDDs; (b) *Number of accessed RDDs* and *total amount of accessed RDD data size* that need to be accessed through the experiment; (c) *Total work* (i.e., makespan) that is the total calculation time for finishing all jobs; and (d) *Average waiting time* for each job.

Fig. 4 depicts the performance of the five caching algorithms. We conduct a set of simulation experiments by configuring different cache sizes for storing RDDs. Clearly, our algorithm (“Adaptive") significantly improves the hit ratio (up to 70%) across different cache sizes, as seen Fig. 4(a) and (b). In contrast, the other algorithms start to hit RDDs (with hit ratio up to 17%) only when the cache capacity becomes large. Consequently, our proposed algorithm reduces the number of RDDs that need to be accessed and calculated (see Fig. 4(c) and (d)), which further saves the overall computation costs, i.e., the total work in Fig. 4(e) and (f). We also notice that such an improvement from “Adaptive" becomes more significant when we have a larger cache space for RDDs, which indicates that our adaptive algorithm is able to better detect and utilize those shareable and reusable RDDs across jobs.

C. Spark Implementation

We further evaluate our cache algorithm by integrating our methodology into Apache Spark 2.2.1, hypervised by VMware Workstation 12.5.0. Table II summarizes the details of our testbed configuration. In Spark, the memory space is divided into four pools: storage memory, execution memory, unmanaged memory and reserved memory. Only storage and execution memory pools (i.e., $UnifiedMemoryManager$) are used to store runtime data of Spark applications. Our implementation focuses on storage memory, which stores cached data (RDDs), internal data propagated through the cluster, and temporarily unrolled serialized data. Fig. 5 further illustrates the main architecture of modules in our implementation. In detail, different from Spark’s built-in caching that responds to *persist* and *unpersist* APIs, we build an *RDDCacheManager* module in the *Spark Application Layer* to communicate with cache modules in the *Worker Layer*. Our proposed module maintains statistical records (e.g., historical access, computation overhead, DAG dependency, etc.), and automatically decides which new RDDs to be cached and which existing RDDs to be evicted when the cache space is full.

TABLE II. TESTBED CONFIGURATION.

Component	Specs
Host Server	Dell PowerEdge T310
Host Processor	Intel Xeon CPU X3470
Host Processor Speed	2.93GHz
Host Processor Cores	8 Cores
Host Memory Capacity	16GB DIMM DDR3
Host Memory Data Rate	1333 MHz
Host Hypervisor	VMware Workstation 12.5.0
Big Data Platform	Apache Spark 2.2.1
Storage Device	Western Digital WD20EURS
Disk Size	2TB
Disk Bandwidth	SATA 3.0Gbps
Memory Size Per Node	1 GB
Disk Size Per Node	50 GB

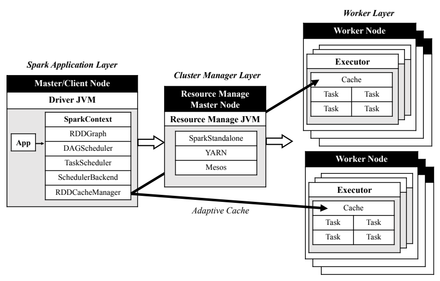

Fig. 5. Module structure view of our Spark implementation, where our proposed *RDDCacheManager* module cooperates with *cache* module inside each worker node.

We select *Ridge Regression* [30] as a benchmark because it is a ubiquitous technique, widely applied in machine learning and data mining applications [31], [32]. The input database we use is a huge table containing thousands of entries (i.e., rows), and each entry has more than ten features (i.e., columns). More than hundred Spark jobs are repeatedly generated with

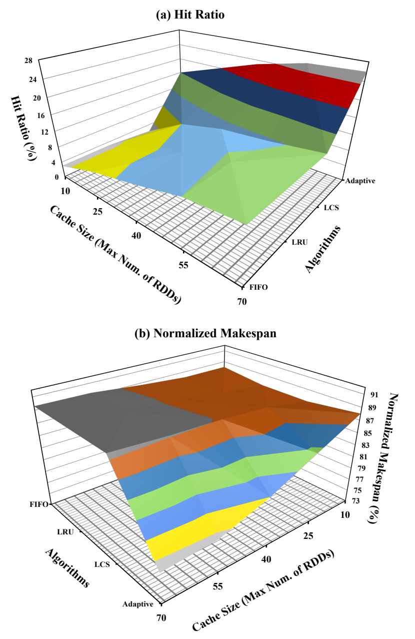

Fig. 6. Hit ratio and normalized makespan results of a stress testing on cache-unfriendly *Ridge Regression* benchmark with different cache sizes under four cache algorithms.

an exponential arrival rate. Each job's DAG contains at least one *Ridge Regression*-related subgraph, which regresses a randomly selected feature column (i.e., target) by a randomly selected subset of the remaining feature columns (i.e., source), i.e., $f_t = \Re(\vec{f_s})$, where f_t is the target feature, and $\Re(\vec{f_s})$ is the regressed correlation function with an input of source feature vector $\vec{f_s}$. Moreover, different jobs may share the same selections of target and source features, and thus they may have some RDDs with exactly the same generating logic chain (i.e., a subset of DAGs). Unfortunately, the default Spark cannot identify RDDs with the same generating logic chain if they are in *different* jobs. In order to identify these reusable and identical RDDs, our proposed *RDDCacheManager* uses a mapping table to records each RDD's generating logic chain *across* jobs (by importing our customized header files into the benchmark), i.e., we denote RDD_x by a hashing function $key \leftarrow hash(G_x(V,E))$, where $G_x(V,E)$ is the subgraph of RDD_x (V is the set of all ancestor RDDs and E is the set of all operations along the subgraph). Since not all operations are deterministic [33] (e.g., *shuffle* operation on the same input data may result in different RDDs), we only monitor those deterministic operations which guarantee the same output under the same input.

Rather than scrutinizing the cache-friendly case where our adaptive algorithm appears to work well as shown in Sec. IV-B, it will be more interesting to study the performance under the cache-unfriendly case (also called "stress test" [34]), where the space size of different combinations of source and target features is comparatively large, which causes the production of a large number of different RDDs across jobs. Moreover, the probability of RDDs reaccess is low (e.g., the trace we generated has less than 26% of RDDs are repeated across all jobs), and the temporal distances of RDDs reaccess are also relatively long [35]. Thus, it becomes more challenging for a caching algorithm to make good caching decisions to reduce the total work under such a cache-unfriendly case.

Fig. 6 shows the real experimental results under four different caching algorithms, i.e., FIFO, LRU, LCS, and Adaptive. To investigate the impact of cache size, we also change the size of storage memory pool to have different numbers of RDDs that can be cached in that pool. Compared to the other three algorithms, our adaptive algorithm achieves non-negligible improvements on both hit ratio (see in Fig. 6(a)) and makespan (see in Fig. 6(b)), especially when the cache size increases. Specifically, the hit ratio can be improved by 13% and the makespan can be reduced by 12% at most, which are decent achievements for such a cache-unfriendly stress test with less room to improve performance. Furthermore, we observe that Adaptive significantly increases the hit ratio and reduces the makespan when we have more storage memory space, which again indicates that our caching algorithm has the ability to make good use of memory space. In contrast, the other algorithms have less improvement on hit ratio and makespan, since they cannot conduct cross-job computational overlap detection. While, with a global overview of all accessed RDDs, our adaptive algorithm can effectively select proper RDDs from all jobs to be cached in the limited storage memory pool.

V. Related Work

Memory management is a well-studied topic across in-memory processing systems. Memcached [36] and Redis [37] are highly available distributed key-value stores. Megastore [38] offers a distributed storage system with strong consistency guarantees and high availability for interactive online applications. EAD [39] and MemTune [40] are dynamic memory managers based on workload memory demand and in-memory data cache needs. There are some heuristic approaches to evict intermediate data in big data platforms. Least Cost Strategy (LCS) [22] evicts the data which lead to minimum recovery cost in future. Least Reference Count (LRC) [41] evicts the cached data blocks whose reference count is the smallest where the reference count dependent child blocks that have not been computed yet. Weight Replacement (WR) [42] is another heuristic approach to consider computation cost, dependency, and sizes of RDDs. ASRW [43] uses RDD reference value to improve the memory cache resource utilization rate and improve the running efficiency of the program. Study [44] develops cost metrics to compare storage vs. compute costs and suggests when a transcoding on-the-fly solution can be cost-effective. Weighted-Rank Cache Replacement Policy (WRCP) [45] uses parameters as access frequency, aging, and mean access gap ratio and such functions as size and cost of retrieval. These heuristic approaches do use optimization frameworks to solve the problem, but they are only focusing on one single job, and ignoring cross-job intermediate dataset reuse.

VI. Conclusion

The big data multi-stage parallel computing framework, such as Apache Spark, has been widely used to perform data processing at scale. To speed up the execution, Spark strives to absorb as much intermediate data as possible to the memory to avoid repeated computation. However, the default in-memory storage mechanism LRU does not choose reasonable RDDs to cache their partitions in memory, leading to arbitrarily sub-optimal caching decisions. In this paper, we formulated the problem by proposing an optimization framework, and then developed an adaptive cache algorithm to store the most valuable intermediate datasets in the memory. According to our real implementation on Apache Spark, the proposed algorithm can improve the performance by reducing 12% of the total work to recompute RDDs. In the future, we plan to extend our methodology to support more big data platforms.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets." *HotCloud*, vol. 10, pp. 10–10, 2010.

[2] "Apache Spark," http://spark.apache.org/.

[3] Apache Hadoop. [Online]. Available: http://hadoop.apache.org/

[4] D. Maier, *Theory of relational databases*. Computer Science Pr, 1983.

[5] T. Hastie, R. Tibshirani, and J. Friedman, "The elements of statistical learning: data mining, inference and prediction," *New York: Springer-Verlag*, vol. 1, no. 8, pp. 371–406, 2001.

[6] Y. C. Eldar and G. Kutyniok, *Compressed sensing: theory and applications*. Cambridge University Press, 2012.

[7] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni *et al.*, "Morpheus: Towards automated slos for enterprise clusters." in *OSDI*, 2016, pp. 117–134.

[8] 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.

[9] S. Ioannidis and E. Yeh, "Adaptive caching networks with optimality guarantees," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 44, no. 1. ACM, 2016, pp. 113–124.

[10] K. Shanmugam, N. Golrezaei, A. G. Dimakis, A. F. Molisch, and G. Caire, "Femtocaching: Wireless content delivery through distributed caching helpers," *Transactions on Information Theory*, vol. 59, no. 12, pp. 8402–8413, 2013.

[11] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[13] R. Barrett, "Under the covers of the google app engine datastore," *Google I/O, San Francisco, CA*, 2008.

[14] "Apache HBase," http://hbase.apache.org/.

[15] B. Scholkopf and A. J. Smola, *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2001.

[16] I. Jolliffe, *Principal component analysis*. Wiley Online Library, 2002.

[17] K. Gregor, I. Danihelka, A. Mnih, C. Blundell, and D. Wierstra, "Deep autoregressive networks," in *Proceedings of The 31st International Conference on Machine Learning*, 2014, pp. 1242–1250.

[18] L. Fleischer, M. X. Goemans, V. S. Mirrokni, and M. Sviridenko, "Tight approximation algorithms for maximum separable assignment problems," *Mathematics of Operations Research*, vol. 36, no. 3, pp. 416–431, 2011.

[19] E. O'Neil, P. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, Washington, DC, 1993, pp. 297–306.

[20] N. Megiddo and D. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 2003, pp. 115–130.

[21] D. Lee, J. Choi, J.-H. Kim, S. Noh, S. L. Min, Y. Cho, and C. S. Kim, "LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies," *IEEE Transactions on Computers*, vol. 50, no. 12, pp. 1352–1361, 2001.

[22] Y. Geng, X. Shi, C. Pei, H. Jin, and W. Jiang, "Lcs: an efficient data eviction strategy for spark," *International Journal of Parallel Programming*, vol. 45, no. 6, pp. 1285–1297, 2017.

[23] G. Nemhauser, L. Wolsey, and M. Fisher, "An analysis of approximations for maximizing submodular set functions," *Mathematical Programming*, vol. 14, no. 1, pp. 265–294, 1978.

[24] M. Sviridenko, "A note on maximizing a submodular set function subject to a knapsack constraint," *Oper. Res. Lett.*, vol. 32, no. 1, pp. 41–43, 2004.

[25] A. Krause and C. Guestrin, "A note on the budgeted maximization of submodular functions," CMU, Tech. Rep.

[26] A. Kulik, H. Shachnai, and T. Tamir, "Maximizing submodular set functions subject to multiple linear constraints," in *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2009, pp. 545–554.

[27] A. A. Ageev and M. I. Sviridenko, "Pipage rounding: A new method of constructing algorithms with proven performance guarantee," *Journal of Combinatorial Optimization*, vol. 8, no. 3, pp. 307–328, 2004.

[28] C. Chekuri, J. Vondrak, and R. Zenklusen, "Dependent randomized rounding via exchange properties of combinatorial structures," in *FOCS*. IEEE Computer Society, 2010.

[29] Z. Yang, D. Jia, S. Ioannidis, N. Mi, and B. Sheng, "Intermediate Data Caching Optimization for Multi-Stage and Parallel Big Data Frameworks, extended version," Tech. Rep., 2018, https://goo.gl/CpBijd.

[30] A. E. Hoerl and R. W. Kennard, "Ridge regression: Biased estimation for nonorthogonal problems," *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970.

[31] G. Li and P. Niu, "An enhanced extreme learning machine based on ridge regression for regression," *Neural Computing and Applications*, vol. 22, no. 3-4, pp. 803–810, 2013.

[32] G.-B. Huang, H. Zhou, X. Ding, and R. Zhang, "Extreme learning machine for regression and multiclass classification," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 42, no. 2, pp. 513–529, 2012.

[33] "Spark Programming Guide," http://spark.apache.org/docs/1.6.2/programming-guide.html.

[34] I. Wagner, V. Bertacco, and T. Austin, "Stresstest: an automatic approach to test generation via activity monitors," in *Proceedings of the 42nd annual Design Automation Conference*. ACM, 2005, pp. 783–788.

[35] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu, "vcacheshare: Automated server flash cache space management in a virtualization environment." in *USENIX Annual Technical Conference*.

[36] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux Journal*, vol. 124, no. 5, 2004.

[37] Redis. [Online]. Available: http://redis.io

[38] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the Conference on Innovative Data system Research (CIDR)*, 2011, pp. 223–234.

[39] Z. Yang, Y. Wang, J. Bhamini, C. C. Tan, and N. Mi, "EAD: Elasticity Aware Deduplication Manager for Datacenters with Multi-tier Storage Systems," *Cluster Computing, DOI: 10.1007/s10586-018-2141-z*, 2018.

[40] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu, "Memtune: Dynamic memory management for in-memory data analytic platforms," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 383–392.

[41] Y. Yu, W. Wang, J. Zhang, and K. B. Letaief, "Lrc: Dependency-aware cache management for data analytics clusters," in *Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[42] M. Duan, K. Li, Z. Tang, G. Xiao, and K. Li, "Selection and replacement algorithms for memory performance improvement in spark," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 8, pp. 2473–2486, 2016.

[43] K. Wang, K. Zhang, and C. Gao, "A new scheme for cache optimization based on cluster computing framework spark," in *Computational Intelligence and Design (ISCID), 2015 8th International Symposium on*, vol. 1. IEEE, 2015, pp. 114–117.

[44] A. Kathpal, M. Kulkarni, and A. Bakre, "Analyzing compute vs. storage tradeoff for video-aware storage efficiency." in *HotStorage*, 2012.

[45] S. Ponnusamy and E. Karthikeyan, "Cache optimization on hot-point proxy caching using weighted-rank cache replacement policy," *ETRI journal*, vol. 35, no. 4, pp. 687–696, 2013.