2017 IEEE 37th International Conference on Distributed Computing Systems

Stark: Optimizing In-Memory Computing For Dynamic Dataset Collections

Shen Li† Md Tanvir Amin* Raghu Ganti† Mudhakar Srivatsa† Shaohan Hu† Yiran Zhao* Tarek Abdelzaher*
†IBM Research *University of Illinois at Urbana-Champaign
{shenli, rganti, msrivats, shaohan.hu}@us.ibm.com, {maamin2, zhao97, zaher}@illinois.edu

***Abstract*— Emerging distributed in-memory computing frameworks, such as Apache Spark, can process a huge amount of cached data within seconds. This remarkably high efficiency requires the system to well balance data across tasks and ensure data locality. However, it is challenging to satisfy these requirements for applications that operate on a collection of dynamically loaded and evicted datasets. The dynamics may lead to time-varying data volume and distribution, which would frequently invoke expensive data re-partition and transfer operations, resulting in high overhead and large delay. To address this problem, we present Stark, a system specifically designed for optimizing in-memory computing on dynamic dataset collections. Stark enforces data locality for transformations spanning multiple datasets (*e.g.*, join and cogroup) to avoid unnecessary data replications and shuffles. Moreover, to accommodate fluctuating data volume and skewed data distribution, Stark delivers elasticity into partitions to balance task execution time and reduce job makespan. Finally, Stark achieves bounded failure recovery latency by optimizing the data checkpointing strategy. Evaluations on a 50-server cluster show that Stark reduces the job makespan by 4X and improves system throughput by 6X compared to Spark.**

I. Introduction

In the past few years, in-memory computing frameworks [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11] have made big-data analytics fast, supercharging a rapidly increasing number of applications in the fields of social networks, e-commerce, finance, and telecommunications [12], [13], [14], [15]. Spark [1], as the state-of-the-art in-memory computing system, attracts a tremendous amount of attention from both academia and industry. Many Spark applications operate on dynamic collections of datasets. For example, an advertising optimization system [16] may store user browsing histories into a dataset every hour, and execute algorithms using data of the past few hours. An IT administrator may dynamically load and evict various system log datasets for diagnosis [17], [18], and run interactive queries on subsets of those datasets. Another example is Spark Streaming [2], which divides stream data into timesteps, and relies on the batch-process Spark core to operate on multiple timesteps within a time window. These applications require Spark core not only process a single dataset efficiently, but also excel at computations across a dynamic collection of datasets.

Spark delivers high efficiency when all task inputs are available in local RAM. Violating this data locality condition would force Spark to access disk and network to construct and load data into memory, leading to deteriorated delay. Default Spark relies on the delay scheduling policy [19] to preserve data locality, which allows a task at the front of the queue to wait for a small amount of time if its data-local servers are all busy. Although this policy improves data locality probability for applications dealing with a single dataset, data *co-locality* may still be a far less likely contingency when applications work on a collection of datasets. Data co-locality refers to the property that multiple input datasets are partitioned using the same scheme and cached in the same set of servers correspondingly. Without proper management, a data-local execution slot may not even exist, as multiple input data partitions of the same task can fall into different servers, leaving no chance for the scheduling policy to pursue the co-locality property. Moreover, even if the system preserves data co-locality, time-varying data volume and distribution may result in excessive data re-partitions and transfers, which would inevitably slow down job executions.

In this paper, we present Stark, a system specifically designed for optimizing in-memory computing on dynamic dataset collections. Stark achieves data co-locality by judiciously managing partitioning strategies and data placement policies. There are three major contributions. First, Stark allows applications to preserve data partitioning strategy across a collection of datasets, and arranges corresponding partitions into the same set of physical servers (*i.e.*, co-locality), avoiding huge data shuffling overheads when processing multiple datasets. Second, Stark handles time-varying data volume and distribution by delivering elasticity into partitions, such that partitions may split or merge without re-partitioning the entire dataset collection. Third, Stark achieves bounded failure recovery delay with minimum checkpointing overhead. Stark is implemented based on Spark-1.3.1 by adding 2.9K lines of Scala code. Experiments on a 50-server cluster show that Stark reduces the delay by 4X and improves the throughput by 6X compared to Spark.

The remainder of the paper is organized as follows. Section II describes the background and motivations. System design and implementation details are elaborated in Section III. Section IV discusses evaluation results. We survey related work in Section V. Finally, Section VI concludes the paper.

II. Motivation

This section first provides a high-level Spark background and then discusses observations of inefficient Spark use cases that motivate this work.

1063-6927/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDCS.2017.143

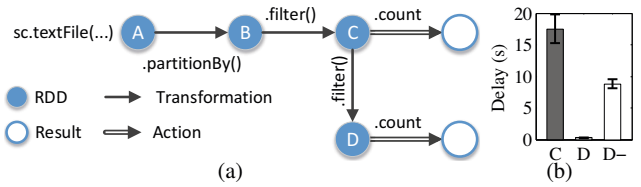


Fig. 1: Data Locality Benefits: (a) shows the lineage graph. In (b), bar `C` represents the delay of the first `C.count` action, and `D` the delay of `D.count` when data locality is preserved. `D-` refers to the case where data locality is violated.

A. Spark Background

Spark [1], [2], [4] is a renowned open-source in-memory computing system that offers both streaming and batch processing capabilities. This section explains two of its major components, Spark Core [1] and Spark Streaming [2].

A Spark job involves a driver program, a cluster manager, and a set of worker nodes. The driver program takes care of the task scheduling of the job. The cluster manager monitors and manages worker nodes that host executors to run tasks and cache data partitions.

Spark Core is the cornerstone of the entire project. Its fundamental programming abstraction is Resilient Distributed Datasets (RDD), an immutable logical collection of data partitioned across machines. RDDs can be created from importing external data or applying transformations on existing RDDs. Transformations indicate dependency relationships between RDDs, connecting RDDs into a Directed Acyclic Graph (DAG) which is called the lineage graph. The immutability allows RDDs to be recomputed when corrupted or lost using only the lineage information and data of any cut on the DAG. Spark Core materializes RDDs in a lazy fashion when some action requires the RDD. Spark supports two types of transformation dependencies, narrow and wide, judged by whether the transformation shuffles data to alter the partitioning strategy. A chain of narrowly dependent transformations is packed into a single stage. Barriers between the map and the reduce phases in wide transformations become stage boundaries. For the sake of accelerating failure recovery, shuffle maps always commit outputs into persistent storage (*e.g.*, HDFS).

Spark Streaming, stacked on top of Spark Core, works as a micro-batching stream processing framework. It batches incoming stream data of each timestep into in-memory data blocks and creates an RDD per timestep. Such series of RDDs are named as a DStream. Spark Streaming relies on Spark Core to handle Further transformations and actions on those RDDs.

B. Observations

In Spark, the lineage graph represents RDDs and transformations that connect RDDs. Spark divides the lineage graph into sub-graphs (stages) using shuffle transformations. More specifically, a shuffle transformation contains a map phase and a reduce phase, and creates a ShuffledRDDs. Spark breaks the lineage graph at the barriers between the map and the reduce phases, leaving each connected component as a stage. Each stage contains a set of tasks that runs the same code on different partitions. The task can be either a ShuffleMapTask or a ResultTask, depending on whether the stage ends at the map phase or the last RDD. The ShuffleMapTask commits map output data into persistent storage, from where the reducers retrieve data to continue the computation. Therefore, as reducers read map outputs from multiple servers through the network, the reducing phases of ShuffledRDDs gain little performance improvements from enforcing data locality. However, data locality can significantly reduce the execution time of transformations with narrow dependencies. For example, the code below generates two jobs, `C.count` and `D.count`. The lineage graph is shown in Figure 1 (a).

```
val A = sc.textFile(...).map(_ => (getTime(_), _))
val B = A.partitionBy(new HashPartitioner(2))
val C = B.filter(_ => _.startsWith("ERROR"))
val D = C.filter(_ => _.length > 30)
C.cache.count;    D.count
```

After Spark executes `C.count`, RDD `C` is cached in memory. RDD `D` relies on `C` as its input. Figure 1 (b) shows how much execution time could be saved by preserving data locality when `sc.textFile` loads a $700MB$ text file. The bars with letters `C` and `D` represent the execution times of `C.count` and `D.count` as the code shows. The job `C.count` creates two sequential stages. The first stage loads the text file, and commits all mapper outputs of RDD `B` into disk. The second stage starts from the reducers of RDD `B`, creates RDD `C`, and computes the count. When `C` is cached, `D.count` starts from the cached data, and the response time stays below 200ms. The `D-` bar represents the execution time of `D.count` after removing `.cache` from the last line of the code. The job `D-.count` creates a single stage that skips the `partitionBy` transformation, which helps to save 8s execution time compared to `C.count`. But, without data locality, `D-` has to start from the reducing phase of `B`, leading to an execution time increase from 0.2s to 9s.

Data locality reduces the job execution time by allowing stages to start from cached RDDs. When Spark fails to preserve data locality, instead of fetching from the executors where the RDD partitions are cached, it recomputes all transformations from the very beginning of the stage—reading data from the map outputs of ShuffledRDDs through the network. Although this design decision considerably reduces the complexity and overhead of keeping track of all cached and evicted data across the entire cluster, it, on the other hand, amplifies the penalty of scheduling a task onto a remote node. In the example shown in Figure 1 (a), forgoing data locality for action `D.count` forces Spark to start computing the stage from the reducing phase of RDD `B`, recompute RDD `C`, and finally create RDD `D`.

The example above demonstrates benefits of running Spark tasks with parent RDD partitions available in the local cache, and penalties otherwise. Working on a collection of dataset exacerbates both the benefits and the penalties, as instead of a single RDD, each job deals with a collection that may contain a large number of RDDs. If a collection partition is scattered in different places, no single executor could preserve data

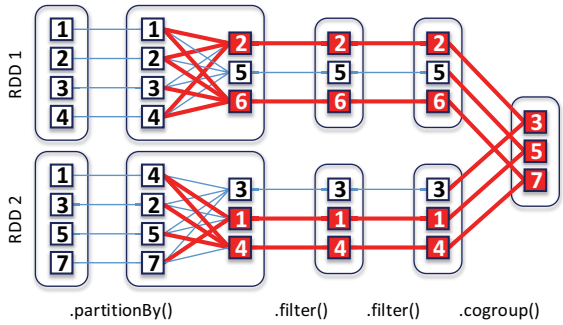


Fig. 2: CoGroup Two RDDs in Spark

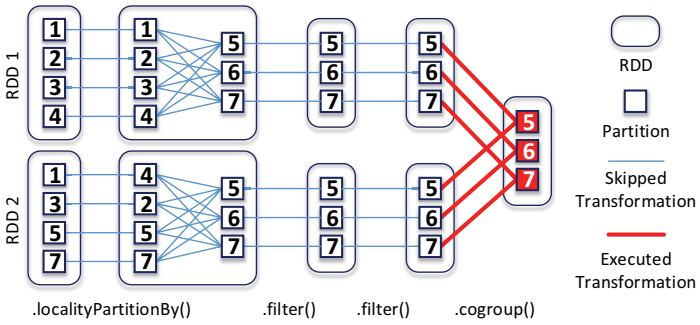


Fig. 3: CoGroup Two RDDs in Stark

locality for the task. Therefore, a job may trigger a massive amount of network and computation overhead. Figure 2 shows an example.

Compared to Figure 1, Figure 2 presents a more detailed view by emphasizing partitions of RDDs. Rectangles with round corners represent RDDs, and squares represent partitions in each RDD. The number in each partition shows the ID of the executor where the partition is cached. For simplicity, the figure only shows cogrouping two RDDs, whereas the number of involved RDDs in a real application might be much larger. On the right side lies the last cogrouped RDD. The red bold lines indicate which transformations on which partitions need to be re-computed. As can be seen in Figure 2 and Figure 3, when co-locality is violated, a single job may trigger many partitions to be recomputed, even if those partitions have already been computed and cached somewhere else.

III. System Design and Implementation

This section first presents the overall design of Stark. Then, we elaborate details of LocalityManager, GroupManager, and CheckpointOptimizer.

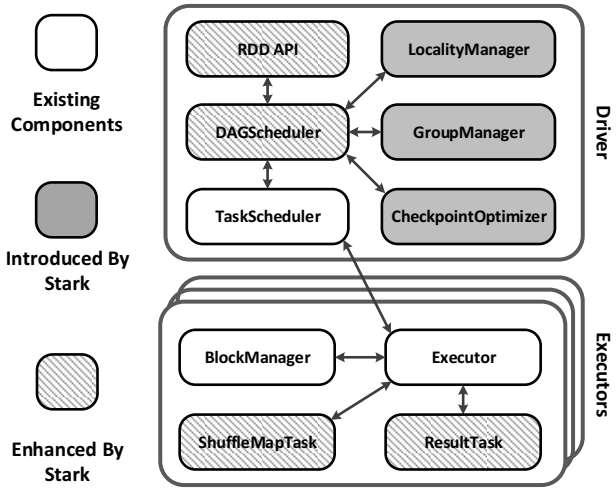


Fig. 4: Stark Architecture

A. Architecture

Although, by orchestrating various components of Spark, a large set of applications have significantly boosted their efficiency [20], applications that operate on dynamic dataset collections can still be improved in the following three aspects. First, to maintain the co-locality property, applications need to organize the in-memory data layout across the cluster deliberately. However, Spark randomly scatters partitions of independent RDDs into servers, exposing no control over partition placements. Second, as applications dynamically load and evict datasets, the data volume and computational demand received by the collection may vary over time. This requires those immutable RDDs in the collection to react to size and popularity dynamics, retaining load balancing. Third, dynamic collections of interdependent datasets may foster an ever-growing lineage graph. The system needs to minimize the overhead to achieve data persistence, and at the same time preserve failure recovery delay bounds. Therefore, we design Stark to handle these three problems accordingly. Figure 4 illustrates the high-level architecture of Stark. Besides enhancing multiple important building pieces of Spark, Stark introduces three novel components:

- **LocalityManager** enforces the same partitioner across multiple user-specified RDDs, and allocates corresponding partitions into same worker nodes in a best-effort manner. This helps to achieve data co-locality that could significantly benefit operations working on multiple RDDs (*e.g.*, cogroup and join).
- **GroupManager** introduces an extendable hashing policy to achieve partition elasticity, which allows immutable RDDs to shrink or expand partitions without repartitioning.
- **CheckpointOptimizer** employs a variant of network flow algorithm to optimally select the minimum amount of data to checkpoint, and at the same time fulfill a user-defined failure recovery delay bound.

B. Locality Manager

Partition is the unit for computation and storage management in Spark. Co-partition and co-locality are two important features that closely relate to system performance. The former partitions multiple RDDs using the same partitioning strategy, which helps to avoid shuffling overhead when applying join or co-group transforms on those RDDs. Let us use the notion *collection partition* to denote the corresponding partitions across co-partitioned RDDs. For example, collection partition 1 of two RDDs refers to the first partitions in both RDDs. The latter (co-locality) places an entire collection partition into

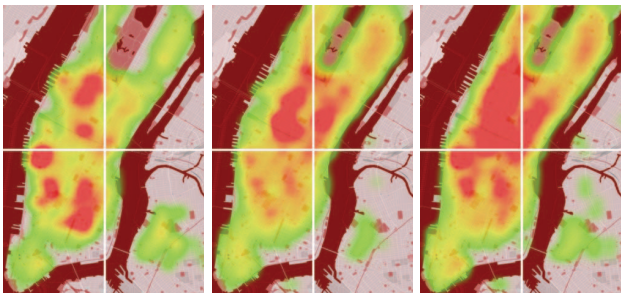
Fig. 5: Example of Violating Co-Locality

the same executor, avoiding the cost of aggregating the data. Users may easily achieve co-partition by passing the same deterministic partitioner when constructing RDDs, while co-locality is more difficult to preserve. Spark randomly places partitions in the cluster, which means it is unlikely that data in the same collection partition would reside in the same server. Figure 5 demonstrates an example that violates data co-locality. The dataset collection contains two RDDs with each divided into three partitions. The number inside each partition represents its ID. In this example, a join or cogroup operation will create three tasks to process the three collection partitions respectively. Regardless of how Spark schedules those tasks, every task will encounter at least one remote RDD partition, as no collection partition locates in the same server.

LocalityManager helps to allocate RDD partitions in the same collection partition onto the same executor. Later in Section III-C, we discuss techniques for solving potential side-effects of a collection partition growing too large to fit into a single executor. The LocalityManager internally remembers the mapping from collection partitions to executors. Users may create such mappings by calling the `localityPartitionBy` API on an RDD or a DStream. During task scheduling, the DAGScheduler first consults the LocalityManager to get the preferred executor ID, and then follows the default delay scheduling algorithm. A collection partition maps to a set of executors instead of a single one. Because whenever a task for a collection partition runs on a remote executor, the partition data is computed and cached in that executor, immediately making the partition data locally available for subsequent tasks on that same executor. This may happen when some collection partition takes too long to process or becomes too popular, overloading local executors. Therefore, the default delay scheduling policy tends to create more replications for those hotspot collection partitions.

Preserving co-locality significantly reduces job makespan. Based on the same example demonstrated in Figure 2, Figure 3 shows how partitions are allocated onto executors when LocalityManager is enabled. The three collection partitions consistently map to executors 5, 6, and 7 respectively, preventing jobs from having to read data from shuffled map outputs, and at the same time saving computational overhead of two transformations.

Enforcing collection partition co-locality also alters the behavior of memory allocation and task scheduling, both fundamentally affecting the system performance. Stark needs to make sure that all negative consequences are taken care of. As one obvious side-effect, a collection partition may grow too large or become too popular that overwhelms memory and/or computation resources of its corresponding executor set. Section III-C discusses this problem in detail and presents solutions. Besides that, another important core function is failure recovery. Spark recovers by recomputing unavailable



(a) (b) (c)

Fig. 6: Time-varying distribution of NYC taxi pick-up/drop-off events in 2013: (a) July 1^{st} morning, (b) July 1^{st} evening, (c) July 4^{th} evening.

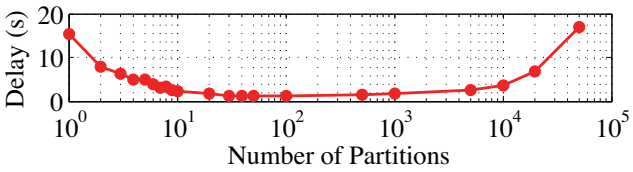


Fig. 7: Partition Number Trade-Off

RDD partitions from checkpoints and/or ShuffledRDDs. It speeds up this process by employing multiple executors to recover RDDs in parallel. At the first glance, collection partition co-locality would slow down the recovery process, as a collection partition could be destroyed by even a single executor failure. Although this observation is true, we claim that the failure recovery with collection partition co-locality is at least as efficient as default Spark from the perspective of job makespan. Because the default Spark scheduler creates one task per result cogrouped RDD partition, which has to construct the entire collection partition in a single executor. Hence, even if a collection partition could be recovered using multiple executors, it has to aggregate into the same server before performing any further operations. Therefore, co-locality introduces no penalty for failure recovery.

C. Group Manager

In this section, we first consider trade-offs between different partitioning schemes in Section III-C1. Then, Section III-C2 presents the idea of extendable partition group that helps to achieve elasticity. Finally, Section III-C3 describes how Stark schedules partition groups when it fails to preserve data locality.

1) Partitioning Trade-Offs: A dataset collection may be subject to time-varying volume and distribution as datasets are dynamically inserted and evicted from the collection. For example, Figure 6 illustrates taxi pick-up/drop-off events [21], [22] heatmap of three different 4-hour time slots in the Manhattan area. Suppose an application creates a dataset every ten minutes, and uses the dataset collection of the past hour to calculate taxi trajectories and optimize the advertisements displayed in each taxi's monitor. The three figures correspond to July 1^{st} morning, July 1^{st} evening, and July 4^{th} evening of 2013 respectively. The white grid helps to emphasize the locations of event hotpots. They clearly show that data distri-

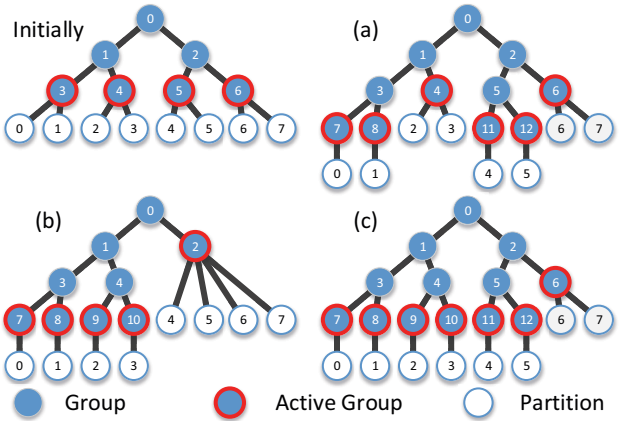


Fig. 8: Extendable Partition Groups for NYC Taxi Data

bution drastically changes over time, and there are much larger hotspot areas in (c) compared to (a) and (b). Therefore, no static partitioning algorithm could always preserve collection partition size under a reasonable threshold over time.

One straightforward solution to this problem would be to further divide data into finer granularity, which reduces the absolute size of partitions. However, this solution creates too many partitions that would overload the scheduler, and amplify system scheduling and monitoring overheads. Figure 7 depicts how the number of partitions affects the execution time of a Spark job. The experiment runs the same code as shown in Figure 1, and records the execution time of `C.count`. We manipulate the number of partitions by tuning the argument of `HashPartitioner`. The result shows that using more partitions initially does help reduce job execution time. However, as the number of partitions increases, the overheads gradually dwarf, and eventually completely overshadow the benefit of using higher parallelism. Another solution would be to repartition the RDDs when partitions are severely skewed, which unfortunately leads to significant shuffling overhead as it may break all partition boundaries.

To solve this dilemma, Stark first divides data into small partitions and then organizes partitions into non-overlapping groups. Partitions in the same group are packed into the same task to reduce scheduling overhead. As each group contains multiple partitions, it may split into smaller groups without violating partition boundaries, avoiding the shuffle phase. More specifically, Stark handles the time-varying data and computation distributions by employing two mechanisms:

- **Extendable Partition Group** mitigates time-varying data distribution in a collection by splitting excessively large partitions to make use of memory and computation resources on multiple machines, and merging tiny partitions to reduce the scheduling and control overhead.
- **Contention Aware Replication** scheme materializes multiple copies of collection partitions based on their demand, minimizing potential cache eviction penalties caused by excessive replications.

2) Achieving Partition Elasticity: In Spark, partitioners share the same `getPartition` API to map a data key onto a partition ID. Achieving elasticity using the `getPartition` API would alter the key-to-partition mapping, resulting in exorbitant data shuffling cost when resizing partitions. To avoid this shuffling overhead, Stark attains elasticity from a higher level, which respects the mapping by keeping the `getPartition` API intact. Extendable partitioning enhances existing partitioners by introducing the concept of *partition group*. A partition group is a set of consecutive partitions, which can split into sub-groups or merge with other partition groups when necessary. Initially, the extendable partitioner creates g partition groups, with each group containing e partitions. Both parameters are configurable. The i^{th} group contains partitions $e \cdot i$ to $e \cdot (i+1) - 1$. To simplify the presentation, we require the configurable parameters g and e both to be powers of 2. This requirement can be easily relaxed by using the smallest complete binary tree that contains exactly g leaves.

Stark initially constructs a full binary tree where each leaf node in the tree corresponds to a partition group. We call this binary tree the *Group Tree*. Figure 8 elucidates example group trees constructed based on the data distribution as shown in Figure 6, assuming coordinates map to ordered-partitioned one-dimensional keys using Z encoding algorithm [23]. The four active groups in the initial status correspond to the four geographic regions as highlighted by the white grid. In Figure 6 (a), workloads overload the left two regions, which correspond to group 3 and 5 according to Z encoding algorithm. Hence, the Group Tree splits group 3 and 5 as shown in Figure 8 (a). Examples (b) and (c) work in the same fashion. A Group Tree supports two basic operations, split and merge. Split can be applied to any leaf node with more than one partition, dividing those partitions into two smaller groups. The merge operation can only be applied to two leaf node groups under the same parent node, concatenating partitions from the two groups into a larger group. At the invocation time, splitting and merging groups only change the mapping from partitions to groups. Therefore, the computational complexity is linear to the number of leaves involved. The real data materialization is delayed till the next Spark action.

The split and merge operations are triggered by the total size of partitions in each group. Multiple RDDs may share the same Group Tree by using extendable partitioner under the same namespace, collectively affecting group sizes. The user may configure how many of the most recent RDDs are accounted when calculating the group size, as well as the upper and lower bounds of group sizes that trigger the split and merge operations.

A partition group is the minimum task scheduling unit in Stark. We introduce `GroupResultTask` and `GroupShuffleMapTask` as enhancements for `ResultTask` and `ShuffleMapTask` to allow multiple partitions in the same group to be packed into the same task. As we have discussed in Section III-C1, this feature helps to reduce scheduling and monitoring overhead by using a smaller number of tasks. Moreover, splitting (merging) a partition group also splits (merges) the corresponding local executors. This helps to minimize data movement during group dynamics by skipping cached partitions.

3) Contention Aware Replication: Extendable Partition Group solves the load balancing problem caused by data volume and distribution dynamics. As discussed in Section III-C1, there is another dimension of load balancing problem induced by time-varying and non-uniformly distributed computational demand on different partitions. To illustrate the problem, let us consider the same taxi advertising example mentioned in Section III-C1. Besides the spatial-temporal dynamics of taxi trajectories, the intensity of advertisement campaigns in certain areas may also change overtime. For example, theaters and shopping centers near the Time Square may want to deliver much more commercial messages to potential customers nearby in weekend evening compared to weekday morning. To address their demand, the system needs to first filter qualified trajectories using the location information, and then conduct subsequent optimization algorithms to match sponsor messages to taxi monitors. In this case, partitions that cover the Time Square receive higher computational workloads than others in weekend evening. Therefore, the amount of workload hitting the same partition changes over time, while such workload dynamics of different partitions may not follow the same pattern. To tackle this load balancing problem, Stark delivers computational resource elasticity to the partition level by independently replicating each partition on demand while preserving data co-locality in the best effort manner.

To achieve the computational resource elasticity, Stark needs to capture the right signal to trigger replicate and de-replicate operations. Failing to preserve data locality for a certain task on partition α conveys a relevant signal. This signal indicates either partition α has become a hotspot receiving higher computational demand, or the corresponding worker nodes where partition α resides has been assigned to too many different partitions. The latter situation forces every partition to replicate on an unnecessarily larger number of worker nodes, competing for limited computational resources. Although this could help to improve the CPU utilization, sharing worker nodes among many partitions catalyzes cache eviction, and makes data locality more difficult to achieve.

The delay scheduling policy improves data locality by allowing tasks to delay for a bounded amount of time to wait for local executors. When they fail to achieve data locality, all remote workers are treated equally. This design is reasonable and helpful when it was initially proposed for Hadoop MapReduce workloads. However, when imported into the realm of in-memory computation, scheduling a task on a remote node materializes all its narrowly-depended parents on that node, converting the node from `REMOTE` to `NODE_LOCAL` for subsequent tasks requiring the same input data. At the same time, this node's locality level may also revert back from `NODE_LOCAL` to `REMOTE` for some other partitions due to cache evictions. Therefore, schedulers for in-memory computation need to make more careful decisions when assigning tasks into remote worker nodes.

Figure 9 (a) and (b) illustrate two extremes: each worker node is dedicated to a single collection partition, or partitions can be assigned to any worker nodes in the cluster. In case (a),

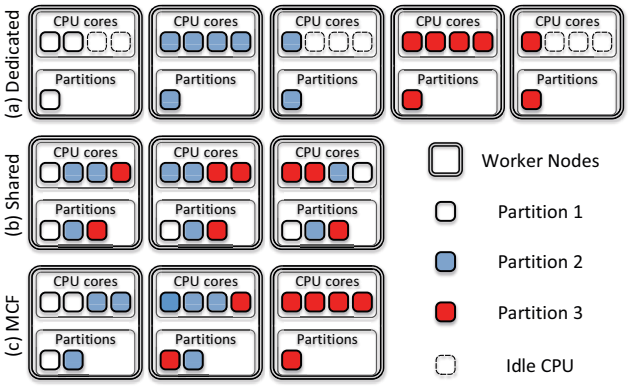


Fig. 9: Task Scheduling Algorithms

every partition exclusively consumes all RAM resources on its worker nodes, allowing it to fit more RDDs of the collection into the cache. Consequently, less tasks have to load data from disk. However, as we can clearly see in the example, many CPU cores stay idle under this scheme, as a price paid to guarantee the exclusiveness. In case (b), the scheduler blindly assigns tasks to remote resources, where any task may end up executing on any remote worker node. This scheduling policy optimally utilizes computational resources in the cluster at the cost of restricting every collection to use a small share of RAM on worker nodes, which may even cause cascading cache eviction that results in higher cache miss rate.

Algorithm 1: Minimum Contention First Scheduling

Input: task set of a stage $\mathbf{T}$, offers $\mathbf{R}$
Output: tasks to launch $\mathbf{L}$
1 **if** *max locality level is not* $REMOTE$ **then**
2 use default delay scheduling
3 **else**
4 $\mathbf{L} \leftarrow \emptyset$
5 use quicksort to ascendingly order $\mathbf{R}$ based on the number of unique collection partitions cached.
6 **for** $i \leftarrow 1 \sim |\mathbf{R}|$ **do**
7 $t \leftarrow$ pick one task from $\mathbf{T}$
8 $\mathbf{T} \leftarrow \mathbf{T} \setminus \{t\}$; $\mathbf{L} \leftarrow \mathbf{L} \cap (t, R[i])$
9 **return** $\mathbf{L}$

We propose the *Minimum Contention First* (MCF) Delay Scheduling algorithm. MCF algorithm works exactly the same as the default delay scheduler before the locality level rises to `REMOTE`. When MCF algorithm is enabled, each executor keeps track of unique collection partitions cached in RAM. The scheduler assigns higher priority to executors with a smaller number of unique collection partitions. Algorithm 1 shows the pseudo code. The most computationally expensive operation is sorting $\mathbf{R}$ on line 5. Therefore, the computational complexity is $O(|\mathbf{R}| \log |\mathbf{R}|)$.

D. Checkpoint Optimizer

Data co-locality and partition elasticity both help to avoid expensive data shuffling operations by preserving the same partitioning strategies across RDDs. Shuffling commits mapper outputs into persistent storage, which naturally breaks

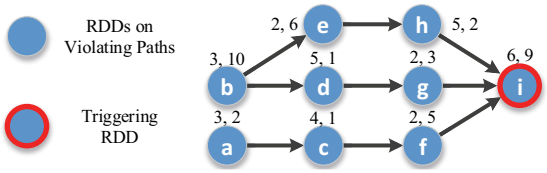


Fig. 10: Checkpoint Example: the two numbers near each RDD represent delay (left) and cost (right).

dependency chains to accelerate failure recovery. Hence, avoiding Shuffle transformations may lengthen failure recovery delay. Therefore, the system needs to proactively and judiciously checkpoint RDDs to bound the failure recovery delay. In a simple case where RDDs are all independent from each other, the system has no choice but checkpointing all RDDs. However, many real-world applications form iterative structures, such as the *runningReduce* concept (*i.e.*, the `updateStateByKey` API) introduced in Spark Streaming [2]. In this case, the system has to select RDDs to checkpoint in the ever-growing application data. Otherwise, the failure recovery has to reconstruct RDDs from the very beginning. The existing solution [5] checkpoints the entire level of recently generated RDDs in the DAG, which is called the Edge algorithm. It guarantees bounded recovery delay, but may lead to excessive checkpointing overhead. This section optimizes the checkpointing algorithm to minimize the amount of data written into persistent storage, and at the same time bounds failure recovery delay.

1) Measuring Parameters: From the perspective of failure recovery, each RDD associates with two important properties, the data size and the computation time. The system can estimate the data size by looking at the amount of RAM consumed by each cached RDD. Default Spark logs computation time at stage level, where a stage may contain multiple consecutive narrowly-dependent transformations, and thus span multiple RDDs. Hence, Stark needs to acquire the computation time from the transformation level. The delay of the same transformation can be different across tasks due to imperfectly balanced data distribution. Stark logs the delay of every transformation in every task, and takes the maximum across tasks as the estimation of the transformation delay. Data size represents the cost (c) of checkpointing an RDD, while computation time denotes the delay (d) of recovering the RDD. With these two notations, we design the optimal checkpointing algorithm for failure recovery.

2) Optimized Checkpointing Algorithm: In the RDD DAG, each node associates with a cost c and a delay d. Let an uncheckpointed path denote the path that contains no checkpointed RDD or ShuffledRDD. The path length is the sum of all RDDs' delay d along the path. Stark keeps track of all uncheckpointed RDDs, and triggers the checkpoint algorithm whenever the length of any path grows beyond the user defined failure recovery delay upper bound r. We call those paths violating paths.

The algorithm breaks all violating paths by checkpointing a set of RDDs with minimum total cost. Figure 10 illustrates an example with $r = 10$. The two numbers near each RDD represent the delay d and the cost c from left to right. Node i triggers the checkpointing algorithm due to three violating paths of length 16, 16, and 15 respectively. As the goal is to find a minimum cut to isolate node i from RDDs a and b, we apply a variant of maximum network flow algorithm to solve this problem. The algorithm splits each RDD node into an *in* node and an *out* node. It then connects the out node of i to a virtual sink node t, and connects a virtual source node s to the in nodes of a and b. The dependency relationships are preserved by linking predecessor's out node to successor's in node. The algorithm assigns the cost c to edge between the *in* node and the *out* node of the same RDD as its edge capacity. The costs of all other edges are set to infinity. Finally, stark uses a standard maximum network flow algorithm to find the minimum cut, representing the set of RDDs with minimum total cost to checkpoint.

Stark determines the set of RDDs to checkpoint by tracing back from virtual sink t to the first set of saturated cutting edges. However, searching for an exact cut may force the system to checkpoint RDDs that are too far away from latest RDDs, leaving a relatively long uncheckpointed path, which would inevitably trigger another checkpoint action soon. Hence, we relax this condition to allow Stark to stop at edges whose residual capacity is within f times of the amount of network flow over it. This is the same as relaxing the checkpointing cost by f times compared to the optimal solution.

E. Implementation

We implement co-locality by introducing the LocalityManager, the `localityPartitionBy` transformation, and the `LocalityShuffledRDD`. They can be enabled by setting the `spark.scheduler.localityEnabled` property to true in the configuration file. Users may call the API on `PairedRDDFunctions` (*i.e.*, RDDs that store key-value pairs) to construct a `LocalityShuffledRDD` as below:

```
localityPartitionBy(p: Partitioner, ns: String)
```

LocalityManager creates a namespace if it has not seen `ns` before, or checks whether the partitioner `p` agrees with the existing partitioner registered with namespace `ns`. All RDDs under the same namespace must use the same partitioner. A namespace starts from a `LocalityShuffledRDD` and automatically carries on to all following narrowly-dependent RDDs. During scheduling, the DAGScheduler consults the LocalityManager for the preferred locations if there is a namespace associated with the RDD. In this way, the system preserves the co-locality in the best effort manner.

The GroupManager manages a bidirectional mapping between partitions and groups. Users may get access to the GroupManager from `SparkEnv`, and report an RDD by calling the `reportRDD(rdd:RDD)` API. This API will trigger GroupManager to calculate the collection partition size across all currently cached RDDs. Users can define lower and upper bounds on the collection partition size (*i.e.*, `spark.locality.max(min)GroupMemSize`). When a collection

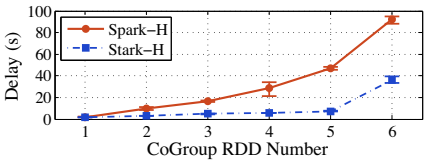


Fig. 11: Co-locality Job Delay

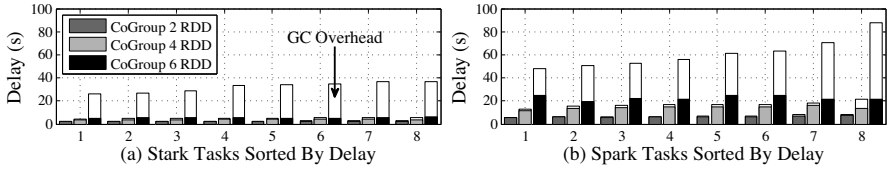


Fig. 12: Co-locality Task Delay

partition grows beyond those thresholds, the GroupManager splits or merges groups accordingly.

To reduce the scheduling overhead, we also introduce `GroupResultTask` and `GroupShuffleMapTask` that may operate on multiple partitions as a single task. Stark automatically creates group tasks if the target RDD associates with a namespace.

In Spark, users have to specify whether an RDD needs to be checkpointed before materializing it. However, this design prevents CheckpointOptimizer from applying the optimization algorithm. To break this constraint, we implement the `RDD.forceCheckpoint` API that creates an `RDDCheckpointData` object and calls the `doCheckpoint` method. This revision allows Stark to checkpoint an RDD after it has been materialized.

IV. Evaluation

This evaluation section first describes datasets and cluster configurations in the Section IV-A. Then, we evaluate improvements individually contributed by techniques proposed in Section III-B (co-locality), Section III-C (extendability), and Section III-D (failure recovery) respectively. Finally, Section IV-E measures the overall system throughput and response time.

A. Experiments Setup

Our experiments run on a 50-server cluster [24] that consists of 40 Dell PowerEdge R620 servers and 10 Dell PowerEdge R610 servers. The Spark/Stark cluster runs on 40 R620 servers, each equipped with 16GB RAM. The remaining 10 R610 servers generate workloads and log delay. Evaluations employ a Wikipedia trace [25], an NYC taxi trace [21], [22], and a Twitter dataset crawled using Twitter REST APIs [26]. The Wikipedia dataset contains the timestamp and URL of requests seen in January 2008. The NYC taxi dataset logs a comprehensive set of data fields, from which we extract the timestamp and location of pickup and drop-off events. Due to the space limit, this paper skips the statistics of these datasets. Dataset use-cases will be explained close to the corresponding experiments. Please refer to [27], [23], and [10] for detailed analyses on these three dataset respectively.

In the evaluation experiments, we compare five different configurations. The prefix indicates whether a configuration uses on Spark or Stark.

- *Spark-R* creates a new RangePartitioner per RDD.
- *Spark-H* shares the same HashPartitioner among all RDDs.
- *Stark-H* applies the same HashPartitioner across RDDs with only co-locality enabled.
- *Stark-S* applies the same StaticRangePartitioner across RDDs with only co-locality enabled.
- *Stark-E* enables Extendability in addition to Spark-S.

B. Data Co-Locality

In this section, we use Wikipedia trace data [25] and typical log mining jobs to measure the performance gain achieved by enabling data co-locality. We pick the log files that contain about 800MB data each, invoke the `testFile(...)` API to read the files from local file system to create an RDD for every file, and use the default hash partitioner with *eight* partitions (*i.e.*, this experiment only uses eight servers). Then, each job cogroups a range of trace RDDs, and counts the number of trace items that contain a randomly picked keyword. Figure 11 shows the results of the average delay of 10 queries. The x-axis is the number of RDDs that each job cogroups, and the y-axis the delay. The delay gap between *Spark-H* and *Stark-H* grows as the number of RDDs increases from 1 to 5, because cogrouping more RDDs in *Spark-H* requires moving more data into the same executor through the network. When cogrouping 5 RDDs, Stark reduces the delay from 46s to 9s by enforcing data co-locality.

The interesting behavior is that, when cogrouping 6 RDDs, the improvement ratio decreases to 3X. To understand the reason behind this phenomenon, let us take a closer look at the delay at the task level for specific jobs. Figure 12 (a) and (b) illustrate task delays when using *Stark-H* and *Spark-H* respectively. In each bar, the white portion denotes the garbage collection (GC) delay, and the remaining denotes the sum of all other delays. We can clearly see that the performance gain drops due to the garbage collection overhead. It is because cogrouping six RDDs consumes an excessive amount of RAM, which leads to more frequent and expensive garbage collections.

C. Extendable Partitioning

In this section, we measure how Stark adapts to data distribution dynamics using consecutive Wikipedia hourly log files. As analyzed in [27], a peak hour log file may contain as much as twice the amount of data compared to that of nadir hours. In the experiments, we compare *Spark-R*, *Stark-S*, and *Stark-E*. Figure 13 shows how these three strategies partition data into groups or partitions, which immediately determines the input data size of different tasks. Each row represents a collection of three RDDs with RDD ID marked on the y-axis. Each cell corresponds to a collection partition (or groups). Darker colors refer to larger partition sizes. In the ideal case, all tasks should have an equal amount of input data, such that no overloaded task would delay the entire job. *Stark-S* clearly suffers from skewed data distribution as some collection partitions are much darker than others. *Spark-R*

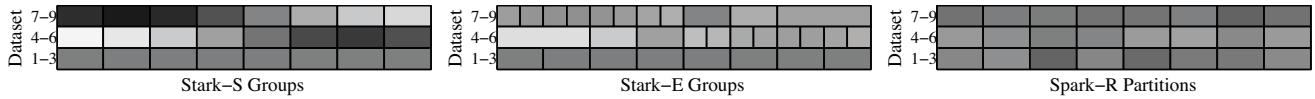


Fig. 13: Task Input Data Size: Each row represents a collection of three RDDs. Each cell represents a collection partition (group) that a task processes. Darker color refers to larger size.

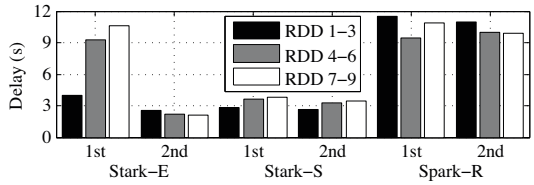


Fig. 14: Job Delay under Skewed Distribution

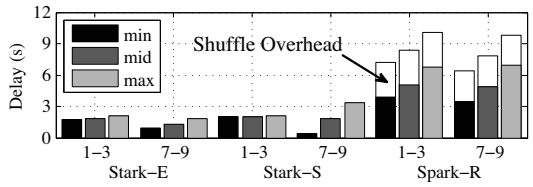


Fig. 15: Task Delay under Skewed Distribution

evenly distributes data across collection partitions, as different RDDs use different range partitioners specifically tailored for their own data distributions. In the *Stark-E* case, partition group boundaries (*i.e.*, sizes of cells in Figure 13) change across RDDs to cater skewed data distribution, resulting in a relatively balanced partition group size distribution.

Figure 14 shows job delays under three configurations. We distinguish the delay of the first job (1st) after group merges/splits from following jobs (2nd). Although *Spark-R* distributes data evenly, jobs take more than 10 seconds to process, regardless of if they are the first or subsequent jobs. This is because RangePartitioners use different boundaries to balance data across partitions, such that a cogroup operation inevitably triggers expensive data shuffling operations. *Stark-S* finishes jobs within 4s. As the partitioning strategy is static and data locality stays unchanged across RDDs, the performance of *Stark-S* ignores whether they are the first or subsequent jobs. However, it takes considerably longer delay to process data with skewed distributions (RDDs 4-6 and 7-9) compared to those with uniform data distributions (RDDs 1-3). *Spark-E* experiences large delay on the first job on RDD 4-6 and 7-9. As it alters the partition-to-executor mapping to regain load balancing, the first job has to reconstruct partition data in newly assigned executors, which contributes to the longer delay. These experiments convey that, if the data is only used once, then there is no need to enable extendable groups, as co-locality alone already achieves the shortest delay. However, many interactive and iterative applications [28] require running a series of jobs on the same set of data, which would considerably benefit from extendable groups.

Figure 15 illustrates task level delay when cogrouping RDDs 4-6. In each bar, the white portion refers to the shuffling delay. It confirms that the shuffling overhead contributes a huge part to *Spark-R* job delay. We can also see that the skewed data distribution causes imbalanced task completion

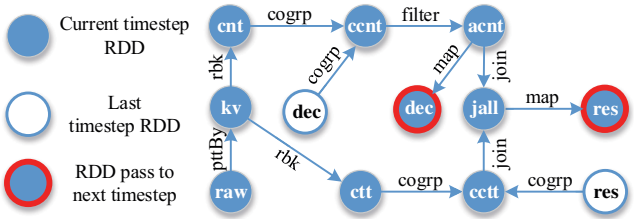


Fig. 16: Application Lineage Graph

times when using *Stark-S*, leading to a longer job delay.

D. Failure Recovery

Failure recovery is evaluated by using an example application as shown in Figure 16. The application tracks popular keys and corresponding contents in a similar way as Twitter trends. Blue nodes without border represent RDDs generated and consumed by the current step. The previous step produces black RDDs which will be consumed by the current step as input. Then, the current step generates RDDs with bold red rims which will be consumed by the next step. In this way, steps are chained together into an ever-growing lineage graph. The application receives a *raw* RDD that contains key-value data in every step. The *raw* RDD partitions to the *kv* RDD by applying the `partitionBy` (pttBy) transformation. Then, the application aggregates the count (cnt) and content (ctt) by calling `reduceByKey` (rbk) API respectively. The *cnt* RDD cogroups (cogrp) with the decayed count RDD (dec) from the last step, and sums by key to create the *ccnt* RDD. Then, it only keeps the popular keys (acnt) by applying the filter API. The *ctt* RDD cogroups with the result RDD *res* from last step to generate the *cctt* RDD. Finally, *cctt* joins with popular keys (*acnt*) to create popular key-value pairs (*jall*) across steps, and cleans the data to arrive the final result (*res*) of the current step. Please notice that we are not arguing this is the optimal way to implement the application logic. How the application is implemented falls out of the scope of this paper.

We run the application for ten steps with different input data sizes. We feed the Wikipedia trace as the input data, and use a fixed-length prefix of the requested URL as the key. We first evaluate how accurate we could estimate checkpoint sizes using cached RDD sizes. As shown in Figure 17, where the white bars represent the cached RDD size and the gray bar the checkpoint size, there is a constant relationship between the cache and checkpoint sizes. Although this constant may change when we use different serialization algorithms, it does not affect the checkpoint algorithm, as the algorithm finds the relatively optimal set of RDDs to persist.

Then, we measure the amount of checkpointed data over steps. Figure 18 compares three schemes: 1) Stark enforces exact optimality (*Stark-1*); 2) Stark relaxes the optimality by using $f = 3$ (*Stark-3*); 3) A variant of Tachyon [5]

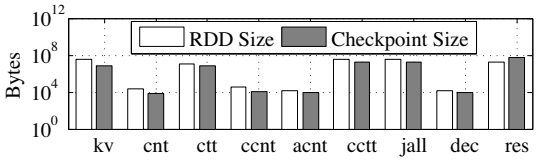


Fig. 17: Estimate Checkpoint Size

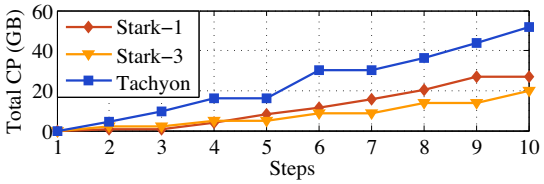


Fig. 18: Total Checkpoint Size

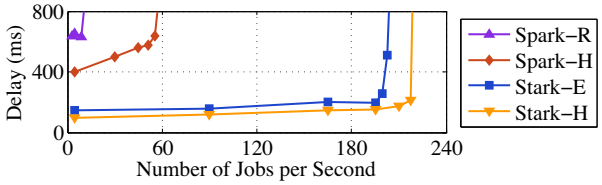


Fig. 19: System Delay

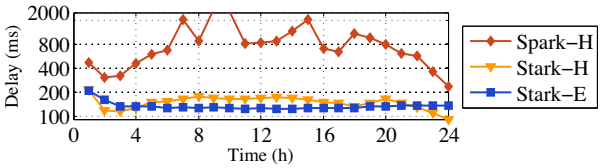


Fig. 20: Job Delay Over Time

Edge checkpointing algorithm that checkpoints all leaf RDDs when invoked. Edge algorithm assumes there is a backend thread checkpointing data whenever the thread is idle, which differs from our assumptions that checkpointing algorithm is proactively triggered to enforce recovery delay bound. So, we revise the Edge algorithm to allow checkpoint operations to be conducted whenever the length of any uncheckpointed path violates the delay threshold. From Figure 18, we can see that Stark writes much fewer data compared to Tachyon, which immediately translates to savings on cluster resource consumption. Because checkpointing all leaves does not guarantee optimality. For example, after calculating *cctt*, the system realizes its recovery chain is too long due to the dependencies among *cctt*, *res*, and *jall*. Hence, Tachyon would checkpoint the current leaf RDDs *dec* and *cctt*. However, after generating *jall*, its recovery chain violates the recovery bound again due to the *ccnt*, *acnt*, and *dec* dependencies. Tachyon would then checkpoint the leaf *jall*. However, as shown in Figure 17, the size of *jall* is much larger than the size of *acnt*. Therefore, Stark would choose to checkpoint *acnt* instead to reduce the overhead.

Stark-1 beats *Stark-3* in the first 4 steps as it enforces exact optimality. However, when the lineage grows larger, *Stark-3* outperforms *Stark-1*, as *Stark-1* tends to leave longer uncheckpointed paths after each action, forcing checkpointing actions to be invoked more frequently.

E. Throughput and Delay

We measure the delay and the throughput using NYC taxi traces and Twitter dataset. The experiment employs a 40-node Stark/Spark cluster, and a 10-node workload generator cluster. As only a small fraction of tweets are tagged with location information, we merge the NYC taxi trace and Twitter dataset together by appending a tweet after every taxi pick-up/drop-off event log, such that every tweet is associated with a geographic coordinate and a new timestamp. We then replay the result dataset as a TCP stream source. Stark uses the streaming component to process incoming data, creating one RDD for every 5-minute data. Spark streaming relies on a single node to create micro-batch RDDs, and then repartitions the RDD to distribute data onto multiple servers. The spatial coordinates are encoded into a one-dimensional space using the Z encoding algorithm [23]. Each job is based on the data within a random time range and a random geographic region, which triggers cogroup operations on multiple RDDs.

As the trace emits data at varying speed over time, the throughput may also change accordingly. In order to conclude a steady number, we first measure the system throughput by revising the trace timestamps and forcing the stream source to release an equal number of tweets per second. To calculate the throughput, we measure the number of jobs per second the system could handle when keeping the delay below 800ms. Figure 19 plots the result. The curve with purple triangles on the upper-left corner represents the *Spark-R* baseline. As it requires expensive data shuffling operation, jobs take 630ms to finish on average and handle only 9 queries per second. When using the same hash partitioner (*Spark-H*) across all RDDs, the response time drops to 405ms, and the throughput improves to 56 queries per second. After further enforcing data co-locality, response time immediately decreases to 109ms, and the system handles up to 220 queries per second. In this set of experiments, as both data generating speed and distribution stay unchanged, partition elasticity helps little. As shown by the *Spark-E* curve, the extra overhead introduced by grouping tasks slightly hurts the system response time and throughput.

In Figure 20, we measure the system response time by replaying the trace at the real speed according to the NYC taxi trace timestamp. The timestep size is set to 5 minutes. Queries analyze a random range of timesteps in the past three hours. Due to the unacceptably high response time and low throughput as shown in Figure 19, the experiment excludes the *Spark-R* baseline. To compare the response time, the amount of workload has to stay within the capability of all baselines. Hence, workload emulators generate 20 jobs per second. As shown by the red curve with diamonds, the response time of *Spark-H* surpasses 800ms when the amount of data generated per second increases. *Stark-H* keeps the response time below 200ms. *Stark-E* shows the benefits of extendable partitions. As the amount of data per timestep increases, each job spans to a larger set of executors with each executor processing a smaller amount of data. We conclude from the results that, although *Stark-E* subjects to longer delays compared to *Spark-H* when facing static and light workloads, it outperforms *Spark-H* by elastically scaling out as the amount of workload grows.

V. Related Work

In-memory computation [1], [2], [6], [8], [10], [29], [30], [31], [32], [33], [34], [35] has recently attracted immense attentions from both academia and industry. Multiple efforts have been expended on investigating various aspects of the problem and pushing forward this direction. For example, Spark [1], as a generalized in-memory computation system, has already greatly boosted data processing solutions for many application needs throughout the industry [20], gaining rapid growth in user population. Spark improves on Hadoop [36] by storing the results of the intermediate stages in memory instead of disk, and thereby eliminating the bottlenecks caused by slow disk accesses. Its streaming components [2] sits on top of Spark core, and enables stream processing using micro-batching strategies. Naiad [8] unifies stream and batch processing systems, offer comparable functionalities as Spark. These systems offer limited optimizations for applications working on collections of datasets.

Spark has a rich ecosystem to handle specialized use cases. Spark SQL [6], [29] supports SQL-like queries on structured data in spark applications. It translates SQL queries into Spark jobs, and optimizes runtime code generation. MLlib [30], as another example, is a scalable machine learning library implemented on Spark. SocialTrove [10] applies Spark and in-memory caching to implement a content-agnostic summarization infrastructure for social data streams. GraphX [31] is a graph processing framework to provide vertex parallel abstractions using the concept of RDD. Several of the above applications also work on dataset collections. For example, the slice transformation in Spark Streaming allows jobs to operate on multiple timestep RDDs within a given time interval. Graphx relies on distributed joins to bring together edge and node data. Therefore, optimizations for dataset collection will benefit these applications as well.

VI. Conclusion

This paper presents the design, implementation, and evaluation of Stark, an optimized in-memory computing system for dynamic dataset collections. Stark enforces data co-locality on dataset collections to avoid unnecessary partition recomputations and data movements. Moreover, Stark delivers elasticity to data partitions (groups), which allows the system to accommodate workload dynamics with low overhead. Finally, it bounds the failure recovery delay using the minimum amount of checkpoint data. Compared to the state-of-the-art solutions, Stark reduces job makespans by 4X, and boosts system throughput by 6X.

Acknowledgments

Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-09-2-0053 (the ARL Network Science CTA) and NSF CNS 13-20209. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] 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 *USENIX NSDI*, 2012.

[2] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *ACM SOSP*, 2013.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *USENIX HotCloud*, 2010.

[4] "Spark: Lightning-Fast Cluster Computing," http://spark.apache.org/, September 2015.

[5] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *ACM SoCC*, 2014.

[6] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: Sql and rich analytics at scale," in *ACM SIGMOD*, 2013.

[7] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *ACM SIGMOD*, 2013.

[8] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *ACM SOSP*, 2013.

[9] "Apache Flink: an open source platform for distributed stream and batch data processing," https://flink.apache.org/, September 2015.

[10] M. T. Amin, S. Li, M. R. Rahman, P. Seetharamu, S. Wang, T. Abdelzaher, I. Gupta, M. Srivatsa, R. Ganti, R. Ahmed, and H. Le, "Socialtrove: A self-summarizing storage service for social sensing," in *USENIX ICAC*, 2015.

[11] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using rdma and htm," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[12] M. Zaharia, "How spark usage is evolving in 2015," in *Spark Summit Europe*, 2015.

[13] F. Abuzaid, J. K. Bradley, F. T. Liang, A. Feng, L. Yang, M. Zaharia, and A. S. Talwalkar, "Yggdrasil: An optimized system for training deep decision trees at scale," in *Advances In Neural Information Processing Systems*, 2016.

[14] R. Bosagh Zadeh, X. Meng, A. Ulanov, B. Yavuz, L. Pu, S. Venkataraman, E. Sparks, A. Staple, and M. Zaharia, "Matrix computations and optimization in apache spark," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[15] S. Chang, Y. Zhang, J. Tang, D. Yin, Y. Chang, M. A. Hasegawa-Johnson, and T. S. Huang, "Positive-unlabeled learning in streaming networks," in *ACM KDD*, 2016.

[16] W. Chen, "Spark and Shark Bridges the Gap Between Business Intelligence and Machine Learning at Yahoo! Taiwan," in *Spark Summit*, 2014.

[17] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. Mccauley, M. Franklin, S. Shenker, and I. Stoica, "Fast and interactive analytics over hadoop data with spark," *USENIX*, vol. 37, no. 4, pp. 45–51, 2012.

[18] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A cost-aware logging mechanism for performance diagnosis," in *USENIX ATC*, 2015.

[19] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys*, 2010.

[20] "Databricks: Make Big Data Simple," https://databricks.com/customers, September 2015.

[21] B. Donovan and D. B. Work, "Using coarse gps data to quantify city-scale transportation system resilience to extreme events," *Transportation Research Board 94th Annual Meeting*, 2014.

[22] New York City Taxi & Limousine Commission (NYCT&L), "Nyc taxi dataset 2010-2013," https://publish.illinois.edu/dbwork/open-data/, 2015.

[23] S. Li, S. Hu, R. Ganti, M. Srivatsa, and T. Abdelzaher, "Pyro: A spatial-temporal big-data storage system," in *USENIX ATC*, 2015.

[24] CyPhy Research Group, "UIUC Green Data Center," http://greendatacenters.web.engr.illinois.edu/index.html, 2015.

[25] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, July 2009.

[26] "Twitter Developers," https://dev.twitter.com/, September 2015.

[27] S. Li, S. Wang, F. Yang, S. Hu, F. Saremi, and T. F. Abdelzaher, "Proteus: Power proportional memory cache cluster in data centers," in *IEEE ICDCS*, 2013.

[28] R. Shiveley, "Changing the Way Businesses Comput and Compete: In-Memory Computing and Real-Time Business Intelligence," 2014.

[29] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *ACM SIGMOD*, 2015.

[30] "MLlib: Apache Spark's scalable machine learning library," http://spark.apache.org/mllib/, September 2015.

[31] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *USENIX OSDI*, 2014.

[32] A. Koliousis, M. Weidlich, R. C. Fernandez, P. Costa, A. L. Wolf, and P. Pietzuch, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *ACM SIGMOD*, 2016.

[33] J. S. Jeong, W.-Y. Lee, Y. Lee, Y. Yang, B. Cho, and B.-G. Chun, "Elastic memory: Bring elasticity back to in-memory big data analytics," in *USENIX HotOS*, 2015.

[34] L. Hu, K. Schwan, H. Amur, and X. Chen, "ELF: Efficient Lightweight Fast Stream Processing at Scale," in *USENIX ATC*, 2014.

[35] A. Bar, A. Finamore, P. Casas, L. Golab, and M. Mellia, "Large-scale network traffic monitoring with dbstream, a system for rolling big data analysis," in *IEEE BigData*, 2014.

[36] S. Li, S. Hu, S. Wang, L. Su, T. Abdelzaher, I. Gupta, and R. Pace, "Woha: Deadline-aware map-reduce workflow scheduling framework over hadoop clusters," in *IEEE ICDCS*, 2014.