Tangram: Bridging Immutable and Mutable Abstractions for Distributed Data Analytics

Yuzhen Huang, Xiao Yan, Guanxian Jiang, Tatiana Jin, James Cheng, An Xu, Zhanhao Liu, and Shuo Tu, *The Chinese University of Hong Kong*

https://www.usenix.org/conference/atc19/presentation/huang

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Tangram: Bridging Immutable and Mutable Abstractions for Distributed Data Analytics

Yuzhen Huang*, Xiao Yan*, Guanxian Jiang, Tatiana Jin, James Cheng, An Xu, Zhanhan Liu, Shuo Tu
The Chinese University of Hong Kong

Abstract

Data analytics frameworks that adopt immutable data abstraction usually provide better support for failure recovery and straggler mitigation, while those that adopt mutable data abstraction are more efficient for iterative workloads thanks to their support for in-place state updates and asynchronous execution. Most existing frameworks adopt either one of the two data abstractions and do not enjoy the benefits of the other. In this paper, we propose a novel programming model named *MapUpdate*, which can determine whether a distributed dataset is mutable or immutable in an application. We show that MapUpdate not only offers good expressiveness, but also allows us to enjoy the benefits of both mutable and immutable abstractions. MapUpdate naturally supports iterative and asynchronous execution, and can use different recovery strategies adaptively according to failure scenarios. We implemented MapUpdate in a system, called Tangram, with novel system designs such as lightweight local task management, partition-based progress control, and context-aware failure recovery. Extensive experiments verified the benefits of Tangram on a variety of workloads including bulk processing, graph analytics, and iterative machine learning.

1 Introduction

Existing offline data analytics frameworks can be roughly classified into two categories according to their data abstractions: **immutable** or **mutable**. The choice of data mutability results in two sets of fundamentally different system features and complex trade-offs between *efficiency* and *robustness* [1].

MapReduce [17] and Spark [70] are representative systems that adopt immutable data abstractions, where data accesses are bulk data movements. MapReduce and Spark provide effective *straggler mitigation* (by speculative execution) and efficient *failure recovery* (by recomputing only the lost partitions), which are critical for large-scale production deployment. As their immutable data abstractions imply a *bulk synchronous parallel* (*BSP*) execution model and lack support for in-place update, MapReduce and Spark do not perform well for workloads that benefit from fine-grained state access and asynchronous execution [2], e.g., sparse logistic regression and single source shortest path (SSSP).

There are also many systems that adopt mutable data abstractions to accelerate iterative workloads, such as vertex-centric graph systems (e.g., Pregel [36], GraphLab [34, 35], PowerGraph [21]) and machine learning systems based on the parameter server architecture (e.g., Parameter Server [31], Petuum [60], TensorFlow [1]). Mutable abstractions enable features such as fine-grained state access and asynchronous execution, which result in enhanced performance for iterative graph analytics and machine learning workloads. However, mutable abstractions make failure recovery and straggler mitigation more challenging. In these systems, failure recovery usually relies on a full restart from the latest checkpoint, and straggler mitigation with speculative execution is not supported as the backup tasks will conduct repetitive updates.

In general, immutable data abstraction leads to efficient failure recovery and effective straggler mitigation, while mutable data abstraction supports a richer set of features at the expense of weaker robustness. We discuss in greater details the interplay among data abstractions, system features and robustness in Section 2. In summary, a clear distinction in existing systems is that they support either mutable or immutable data abstraction, and lack a mechanism to choose which abstraction to use according to a given workload. Our analysis on the trade-offs between the two data abstractions leads to the following questions: *Can we model both mutable and immutable data abstractions under a unified framework? Can the system determine which abstraction to use according to the workloads? Is it possible to provide efficient failure recovery and straggler mitigation under mutable abstraction?*

*Co-first-authors ordered alphabetically.

[1]We refer robustness to efficient failure recovery and straggler mitigation.

[2]Compared with BSP, asynchronous execution by *stale synchronous parallel* (*SSP*) [24, 60] or *asynchronous parallel* (*ASP*) [49, 58] allows the machines/objects to have different progresses in iterative applications. The progress differences among the machines are bounded in SSP but unbounded in ASP.

We propose a distributed data analytics system, called **Tangram**, to bridge the gap between mutable and immutable data abstractions. Tangram adopts a new programming model, **MapUpdate**, which is used in the form ***A.map(B).update(C)***, where *A*, *B* and *C* are distributed data collections. Similar to MapReduce, the map tasks in MapUpdate are coarse-grained and side-effect-free, which allows speculative execution for straggler mitigation and failure recovery by only recomputing the lost partitions. However, the side-input (i.e., collection *B*) is read on a per-record basis and the update tasks conduct in-place update on states (i.e., collection *C*) to make recent updates visible to the map tasks. On this basis, MapUpdate also inherently supports iterative and asynchronous execution.

MapUpdate provides a simple rule to determine whether a collection should be mutable (Section 3), thus enabling the system to use mutable or immutable data abstraction for each collection adaptively according to a given workload. With this adaptability, Tangram provides elegant implementations for workloads including *bulk processing*, *vertex-centric graph analytics* and *iterative machine learning* (Section 4). In addition to good expressiveness, the ability to determine data mutability also enables the system to apply different failure recovery strategies for immutable and mutable collections, providing similar robustness as immutable systems (Section 5).

Tangram translates a MapUpdate plan (i.e., an invocation of "*A.map(B).update(C)*") into a number of *map* and *update* tasks. To reduce the overhead of centralized scheduling, Tangram uses a lightweight local task management strategy to schedule the execution of the tasks and to resolve access conflicts on each machine. A partition-based progress control mechanism is introduced to support iterations and asynchronous execution (Section 5). To achieve high efficiency, Tangram also incorporates optimizations such as delay combiner, process cache, and local zero-copy communication.

Our experiments show that Tangram provides efficient failure recovery and effective straggler mitigation. We also implemented a variety of workloads (e.g., bulk processing, machine learning, graph analytics, distributed crawler) on Tangram and compared their performance with specialized systems and highly optimized low-level MPI implementations. We found that Tangram can concisely express these workloads in an intuitive manner and the experiments show that Tangram's performance is comparable with that of specialized systems. Tangram's expressiveness and efficiency are especially useful for pipelined workloads consisting of multiple types of tasks as it eliminates context switch overheads.

Our main contributions can be summarized as follows:

- An in-depth analysis of the interplay among data abstraction, system features and robustness in existing systems. (Section 2)
- A novel programming model that can determine data mutability and can model both mutable and immutable data abstractions, resulting in good expressiveness. (Sections 3 and 4)
- A set of novel designs (e.g., partition-based progress control) and optimizations (e.g., delay combiner) that support different workloads efficiently on a common runtime. (Section 5)
- A comprehensive evaluation of Tangram's performance on a variety of workloads. (Section 6)

2 Immutable and Mutable Abstractions

We review related systems and analyze the complex interplay among their data abstractions, key features, failure recovery and straggler mitigation strategies. For convenience of discussion, we refer to systems that adopt an immutable/mutable data abstraction as **immutable/mutable systems**. We give a summary in Table 1 and discuss the details below.

Data-parallel analytics frameworks such as MapReduce [17], DryadLINQ [68] and Spark [70] are typical examples of immutable systems. They use functional dataflow graphs to model the dependency among datasets and break a job into multiple stages with dependency. The parallel tasks in each stage are independent and the stages are executed in a synchronous manner in which a stage can only start after its predecessors finish. This execution model enables straggler mitigation with speculative execution, which has been widely adopted and optimized in practice [3–5, 17, 71].

Immutable systems also provide efficient lineage-based failure recovery, for which only the lost data partitions are reconstructed from their parent partitions in the lineage graph. Compared with checkpoint-based recovery, lineage-based recovery can distinguish failure scenarios and does not need to roll back to the latest checkpoint upon every failure. For example, in K-means, if a machine holding a part of the training samples fails (i.e., narrow dependency), Spark only needs to reload the lost samples in parallel and recompute their updates to the centers. Only in cases such as PageRank, when the rank values of some vertices are lost (i.e., wide dependency), a full re-computation from the latest checkpoint is required. Moreover, Spark only checkpoints/reloads datasets that have a long lineage graph containing wide dependency (e.g., the rank RDD in the above example), which is more efficient than checkpointing all RDDs involved in computation.

Immutable systems are inherently stateless and only support BSP. However, many iterative workloads have intuitive stateful representations (e.g., the rank values in PageRank, and the model parameters in sparse logistics regression) and can benefit from asynchronous execution. For example, machine learning algorithms such as stochastic gradient descent (SGD) converge faster under SSP and ASP [24, 49, 69], and it has also been proven that a number of asynchronous graph algorithms have faster convergence compared to their synchronous counterparts [49, 73]. Therefore, many specialized mutable systems such as vertex-centric graph systems [21, 35, 36] and parameter-server-based machine learning systems [31, 60] support in-place state updates and asynchronous execution.

Table 1: The data abstractions and key features of some representative systems

Category	Systems	Usability	Abstraction		System Support		
		Programming Model	State Representation	Access Pattern (Shuffle)	Execution Model	Straggler Handling	Failure Recovery
Stateless Dataflow	MapReduce [17], DryadLINQ [68], Spark [70]	functional	immutable	coarse-grained	BSP	speculative execution	lineage and checkpoint
Distr. Shared Mem.	Piccolo [46]	push/pull	mutable kv table	fine-grained	BSP	task stealing	checkpoint
Parameter Server based	Parameter Server [31]	push/pull	mutable kv table	fine-grained	BSP/SSP/ASP	N/A	replication
	Petuum [60]	push/pull	mutable kv table	fine-grained	BSP/SSP/ASP	N/A	checkpoint
Distributed Graph	Pregel [36]	vertex-program	mutable state	coarse-grained	BSP	N/A	checkpoint or message replay
	GraphLab [34, 35], PowerGraph [21]	vertex-program	mutable state	fine-grained	BSP/ASP	N/A	checkpoint
	Tangram	functional map in-place update	immutable/mutable	partition-based	BSP/SSP/ASP	partition migration	lineage and checkpoint

Asynchronous execution also makes mutable systems more robust to micro stragglers [3] as fewer barriers are enforced. In contrast, immutable systems are prone to micro stragglers as their BSP execution model enforces a synchronization barrier in every iteration.

However, mutable systems only provide sub-optimal system-level solutions to straggler mitigation and failure recovery. Most of the mutable systems in Table 1 rely on the nature of the applications for straggler mitigation and do not provide a system-level support. For example, graph systems such as Pregel [36] and PowerGraph [21] rely on graph partitioning to ensure a balanced workload distribution among workers. Parameter Server [31], Petuum [60], GRACE [58] and Maiter [73] utilize the asynchronous nature of the application algorithms to mitigate micro stragglers. Mutable systems do not support speculative execution as updates are conducted in a fine-grained manner and it is costly to keep track of the committed writes in order to avoid repetitive updates. Instead, mutable systems typically use task stealing to handle stragglers [23, 46].

For fault tolerance, mutable systems usually require a full restart from the latest checkpoint (e.g., Petuum, Pregel), or use expensive replication when recovery time is critical (e.g., Parameter Server). Contrary to immutable systems, mutable systems often recompute everything from the latest checkpoint. In addition, any failure would cause these systems to discard and reload all data. The key problem is that these systems do not distinguish the mutable and immutable parts in an application. Although existing mutable systems can be modified individually to support more efficient fault tolerance, we offer a unified mechanism to solve this problem, which is especially useful for pipelined workloads where datasets can change between mutable and immutable status (e.g., the TF-IDF vectors in the pipelined workload in Section 6.2).

[3]*Micro stragglers* are transiently stalling workers and may be caused by packet loss, system cron jobs, etc. *Macro stragglers* are slow due to more persistent reasons, such as workload imbalance and resource contention.

3 Programming Model

In this section, we introduce our MapUpdate programming model and discuss its differences from MapReduce and stream processing frameworks.

3.1 MapUpdate

The basic data abstraction in our MapUpdate programming model is *collection*, which contains a set of objects (or records) and is usually kept in memory. A collection is divided into partitions and partitions are distributed across machines in the cluster (with hash partitioner by default, but configurable by users). The **MapUpdate** programming model is typically used in the form of

A.map(B, map_func).update(C, update_func),

in which A, B, and C are *map collection*, *side-input collection*, and *update collection*, respectively. The *map_func* has a signature $(T, [S, \ldots]) => seq(K, V)$, which takes in an object of type T in the map collection and (optionally) handler(s) S to the side-input collection(s), and generates some key/value (K/V) pairs. The *update_func* has a signature $(U*, V) => nil$, which takes in a pointer $U*$ to an object in the update collection and an update value V, and returns nothing.

To execute a MapUpdate command (also called a *plan*), a machine launches parallel map tasks on its local partitions of the map collection, where each map task performs *map_func* on the objects in one partition of A to generate intermediate results. The *map_func* may use the information (parameters) provided in the side-input collection B. The intermediate results (of a map task) are then shuffled according to their keys and committed to the corresponding objects in update collection C with the *update_func*. By default, the retrieval of the objects in the side-input collection is fine-grained (i.e., on a per-key basis), while the shuffle of intermediate results and modification to the update collection are conducted on a per-partition basis. MapUpdate associates progress with each partition and allows different partitions to have different progresses, and state access is also progress dependent (Section 5.2). This *partition-based state access pattern* of MapUpdate is different from the *coarse-grained state access pattern* in dataflow systems (e.g., Spark), in which all partitions have the same progress (i.e., BSP). Task execution in the Tangram system is also partition-based, i.e., a partition is the granularity of task execution.

MapUpdate has an explicit side-input collection. In contrast, without the side-input, existing systems (e.g., Spark) may use broadcast for state sharing, which is inefficient for large and sparse states (e.g., sparse logistic regression). Some other systems (e.g., Google Dataflow [2]) also support the side-input collection but require it to be small and immutable. MapUpdate does not have such constraints, enabling it to succinctly and efficiently express workloads that have intuitive stateful representations (see examples of machine learning and graph analytics applications in Section 4). MapUpdate also does not require A, B and C to be different collections. When $A = C$ or $B = C$, by default, MapUpdate does not make a copy of C for read. Instead, MapUpdate reads and writes the same collection, which enables the map tasks to see the latest (maybe inconsistent) updates. As we will show in Section 4, this is important for the asynchronous execution of workloads such as distributed crawler as they can tolerate inconsistent states and benefit from fewer synchronization barriers.

MapUpdate ensures consistency [4] if a plan (e.g., word count) does not write/read the same collection. For plans that write/read the same collection, consistency is not guaranteed. This is not problematic because applications such as SGD can trade consistency for efficiency without sacrificing correctness, and many specialized systems deliberately incorporate designs to benefit from that. When strict consistency is required, users can create another copy of the read collection for write as in Spark and Piccolo. MapUpdate does not enforce any order when committing the updates of the map tasks, and the results of a plan may not be deterministic for some applications (e.g., SGD based logistic regression). In the face of failure, MapUpdate ensures that each update is committed exactly once.

In general, the map collection A contains the input data, such as samples in machine learning and documents in word count, while the side-input collection B provides information needed in computation, such as model parameters in machine learning. The update collection C holds the computation results, e.g., the final count values in word count. The side-input collection can be omitted, for example, a user can write *docs.map(map_func).update(count, update_func)* for word count. Instead of a single collection and function, users can provide multiple side-input collections, update collections, and update functions. Users can easily specify how a plan is executed using the configurations in Table 2, for example, *A.map(B).update(C).setIter(100).setStaleness(2)* will

[4]At iteration t, a read on data sees all updates from iteration smaller than t but not updates from iteration equal or larger than t.

Table 2: Configurations in MapUpdate

Configuration	Description
setIter(*int n*)	Run for *n* iterations
setStaleness(*int s*)	Set staleness to *s*
setCombine(*func*)	Register combiner
setCheckpointInterval(*int n*)	Set checkpoint interval

conduct the MapUpdate plan for 100 iterations using SSP with *staleness* $= 2$. Additionally, *setCombine(func)* provides a function to combine the map outputs before shuffling for communication reduction, and *setChecktpointInterval(n)* configures the checkpoint interval in an iterative application. We will show how these flexibilities of MapUpdate translate into good expressiveness in Section 4.

3.2 Comparison with Existing Frameworks

We highlight the main differences between MapUpdate and MapReduce [17], Flink [10] and Spark Structured Streaming [6] in this section.

MapReduce. MapUpdate differs from MapReduce in several important aspects. First, while *map* in MapUpdate is functional (similar to *map* in MapReduce), *update* allows for asynchronous in-place modification to the stateful collection. Second, MapUpdate allows the side-input collections to be specified explicitly and access to the side-input collections is fine-grained, which improves efficiency in many applications such as machine learning and graph analytics. Third, with designs to be introduced in Section 5, MapUpdate provides support for iteration and consistency protocols including BSP, SSP and ASP (configurable by *setStaleness*). Although there are attempts to support key-value-style update (IndexedRDD [26]) and iteration (HaLoop [9], iterative MapReduce [18], Map-Reduce-Update [8]) under the MapReduce framework, MapUpdate is fundamentally different as these systems do not support in-place updates and asynchronous execution due to their data immutability.

The most important contribution of MapUpdate, however, is that it provides a simple mechanism for the system to determine whether a collection is mutable in a plan from the API call: *the update collection is mutable, and other collections, if different from the update collection, are considered immutable*. For example, a MapUpdate plan training a logistic regression model may be expressed as *samples.map(param).update(param)*, and the system can infer that collection *param* (storing the parameters) is mutable and collection *samples* (storing data samples) is immutable. The ability to determine data mutability allows the system to distinguish failure scenarios and provides efficient failure recovery strategies accordingly as in immutable systems. For example, when a machine fails, the system can determine whether the failed machine holds partitions of *param*. If not, only the lost partitions of *samples* need to be reloaded. Otherwise, the system rolls back to the latest checkpoint for *param*, but the partitions of *samples* on the healthy machines do not need to be reloaded.

To support speculative execution, MapUpdate restricts updates to be conducted on a per-partition basis, in which updates from a map partition are committed together. This per-partition update strategy enables Tangram to record which partition has already committed update and is crucial for speculative execution.

Stream Processing Frameworks. Modern stream processing systems such as Flink and Spark Structured Streaming also support both mutable and immutable abstractions but with restricted applicability. We discuss how MapUpdate is different from them here.

First, states in MapUpdate are shared and can be accessed globally, which allows Tangram to support workloads such as machine learning and graph analytics more efficiently. In contrast, states in Flink are bounded with operators and states in Spark Structured Streaming are restricted to key groups. In fact, states in Flink and Spark Structured Streaming are mainly designed for maintaining states across streaming records (e.g., for session tracking). Thus, they are not efficient for read/write in machine learning and graph analytics workloads, which introduce loops in the computation graph. Specifically, using loops in Flink requires to limit the input rate of the input stream to avoid deadlocks caused by cyclic backpressure [20], while Spark Structured Streaming does not allow loops in the dataflow graph, which is necessary when using the stateful operators for iterative workloads. In contrast, MapUpdate naturally supports iteration and in-place update.

Second, checkpointing in Flink and Spark Structured Streaming is more complicated (as they are designed for stream processing), while Tangram is designed for batch processing and only checkpoints mutable collections. Spark Structured Streaming uses checkpointing and write-ahead logs for fault tolerance. Flink also needs to restart from the latest checkpoint for any failure.

4 Applications

Programming with MapUpdate to construct data-parallel applications is simple: users define the collections, construct the MapUpdate plan by providing the *map*/*update* functions and specify the plan configurations. Low-level system issues such as parallelism and fault tolerance are hidden from users. In this section, we demonstrate how MapUpdate can be used to implement a wide range of applications.

4.1 Bulk Processing

MapUpdate can easily implement the bulk processing workloads targeted by MapReduce, which are usually stateless, non-iterative and involve only bulk data movement. We illustrate by the word count example, which is similar to the one in Spark: the map function generates (word, count) pairs when scanning local documents, while the update function aggregates the (word, count) pairs for final counts. Note that in the plan of word count, there is no side-input collection.

```
// Doc: (word1, word2...): (string, string...)
// WordCount: (word, count): (string, int)
// docs: collection<Doc>
// wordcount: collection<WordCount>
docs.map(doc => (w, 1) for each word w in doc)
    .update(wordcount, (wc, c) => wc.count += c)
```

4.2 Iterative Machine Learning

Iterative machine learning algorithms repeatedly refine a set of model parameters with updates computed from the training samples. These algorithms (e.g., SGD) are usually robust to asynchronous execution, in which update is calculated using outdated or inconsistent model parameters. Parameter-sever-based systems (e.g., Parameter Server [31], Petuum Bösen [60]) are widely used for distributed machine learning and support SSP and ASP to benefit from asynchronous execution. Tangram can model parameter server by using the model parameters as both the side-input collection and update collection. We show an example of training logistic regression using SGD with SSP ($s = 2$). The map function calculates the stochastic gradient of local samples using the model parameters, while the update function commits the gradient updates to the model parameters. Iteration and asynchronous execution can be configured using the *setIter* and *setStalenss* commands in Table 2. Note that when *setStalenss* is not configured, Tangram uses BSP by default.

```
// Sample: (label,(k,v)..): (int, (int,float)..)
// Param: (k,v): (int, float)
// data: collection<Sample>
// params: collection<Param>
map_func(Sample sample, Params params, Output o):
  grad = CalcGrad(sample, params)
  o <- grad  // grad: ((k,v)...)

update_func(Param param, float update):
  param.val -= learning_rate * update

data.map(params, map_func)
    .update(params, update_func)
    .setIter(100).setStalenss(2)
```

4.3 Vertex-Centric Graph Analytics

Vertex-centric graph analytics systems (e.g., Pregel [36], PowerGraph [21]) usually update vertex states iteratively according to the states of neighboring vertexes. Tangram can model vertex-centric graph processing by using the vertex state collection as both map collection and update collection [5]. We use PageRank as an example. The map function calculates the contribution of a vertex's PageRank value to its out-neighbors, while the update function merges the contributions from the in-neighbors. The ranks and the links collection are

[5]Using vertex state as side-input and update collection is also feasible.

co-partitioned (by using the same partitioner) to reduce communication overhead. Similarly, Tangram can also implement the edge-centric model [51].

```
// Rank: (id, pr): (int, float)
// Link: (id, nb1, nb2...): (int, int, int...)
map_func(Rank r, Links links, Output o):
  for each neighbor nb in links[r.id]:
    o <- (nb.id, 0.85 * r.pr/len(links[r.id]))

update_func(Rank r, float contrib):
  r.pr += contrib

ranks.map(links, map_func)
     .update(ranks, update_func)
     .setIter(30)
```

4.4 Distributed Crawler

Tangram supports crawler by using urls as both the map collection and update collection. The map function downloads the web page pointed by the current url and extracts new urls, while the update function inserts the new urls into the urls collection and marks the processed urls as visited. Note that there is no side-input collection. *setIter(-1)* keeps executing the iteration (i.e., keep crawling), while *setStaleness(-1)* means using ASP.

```
// Url: (url, status): (string, ToFetch/Done)
map_func(Url url, Output o):
  if url.status is ToFetch:
    new_urls = DownloadAndExtractNewUrls(url)
    for each new_url in new_urls:
      o <- (new_url, ToFetch)
    o <- (url, Done)

update_func(Url url, Status s):
  if url.status is not Done:
    url.status = s

urls.map(map_func)
    .update(urls, update_func)
    .setIter(-1).setStaleness(-1)
```

In the above applications, we use different combinations of the three collections (A,B,C) to achieve different computation patterns. Tangram also supports many other applications (e.g., Nomad [69] and graph matching [12]) that are hard to be implemented in existing systems.

4.5 Pipelined Workloads

MapUpdate is especially useful for pipelined workloads. In fact, the Tangram project was motivated by production data analytics workloads that are common in companies such as Alibaba, which consist of pipelines involving different types of tasks. Typical pipelines begin with MapReduce-style data processing, then conduct various advanced analytics (e.g., parameter-server-style model training), and end with testing and verification. We briefly describe a *user classification* pipeline and a *fraud detection* pipeline as examples.

In a user classification pipeline, users are divided into groups according to their purchase records to generate labels. The basic information (e.g., age, gender and location), search history and activity patterns (e.g., log-in frequency, active time period) are gathered from multiple tables using MapReduce-style join to produce features. Then, various machine learning models (e.g., logistic regression, SVM) are trained using a parameter-server-based framework to classify users into different purchase pattern groups. Lastly, the models are tested on a held-out dataset to select the best-performing one for use. We remark that user classification is only a component of the much larger item recommendation pipeline, which involves a more diverse set of workloads such as graph analytics and matrix factorization.

In a fraud detection pipeline, the goal is to find malicious sellers who use fake transactions to bump up their scales records [47]. The static relationship among users (i.e., buyers or sellers) and the dynamic payment activities are first processed, and a graph is extracted from the pre-processed data to model the buyer-seller interaction. Then, graph matching is applied to find interaction patterns that match some predefined templates corresponding to fraud patterns. Finally, these interactions are verified by further analysis and the results are used to update the fraud template library. The verification process typically involves MapReduce (e.g., joins to obtain details of suspected users) and graph analytics such as computing the distances from suspected users to blacklisted users.

As we will show in the experiments, processing different tasks in a pipeline with respective specialized systems introduces expensive context switch overheads for dumping/loading output/input data by the systems. Using many systems for a single pipeline also hurts robustness because different systems provide different fault tolerance semantics and require engineers to learn/tune all the systems. With the expressive API of MapUpdate, unified fault tolerance semantics and high efficiency, Tangram (our system that implements MapUpdate) can handle the entire pipeline in a unified framework and thus completely remove the context switch overheads. Moreover, the unified MapUpdate API also significantly reduces development costs without users' need to learn many systems.

5 System Design

Designing a system to support the MapUpdate API is challenging in the following aspects: (1) As tasks in MapUpdate have complicated interactions and dependencies (e.g., read/write conflicts, requiring remote data transfer), a low-overhead task management and scheduling strategy is crucial for efficiency. (2) MapUpdate supports iterative plans and flexible consistency control, which requires a distributed progress control protocol that enables various execution models (i.e., BSP, SSP and ASP) under a unified framework. (3) To achieve efficient failure recovery, effective mechanisms are needed to distinguish failure scenarios and apply different recovery strategies accordingly as analyzed in Section 2.

Tangram adopts a master-worker architecture. The master is responsible for DAG scheduling (coordinating the workers to execute runnable plans), progress tracking (managing progress and collecting execution statistics from workers for fault tolerance and straggler mitigation), and partition management (keeping track of the location of the partitions by maintaining the master copy of the partition map). The workers serve as the distributed in-memory storage for the partitions and each worker uses a local controller to manage local task execution. For scheduling, the master only issues control commands (start, update progress, migrate, recover, etc.) to workers and the local controller is responsible for scheduling its own tasks. The local controller also synchronizes the local copy of the partition map and the execution progress with the master. This design reduces centralized scheduling overhead and is crucial for scaling to large clusters.

5.1 Local Task Management

The local controller in each machine manages three kinds of tasks, i.e., map task, respond task, and update task. A map task runs the user-defined map function for every object in a local map partition, combines the intermediate results locally if a combine function is provided, serializes the (combined) results and adds the results to the sender, which will send them to remote machines (according to the partition map) for update. A map task invokes a fetcher if it needs to fetch some records (e.g., parameters in machine learning) in the side-input collection. The requested records are indexed by keys and the fetcher splits these keys into multiple subsets, each corresponding to a partition of the side-input collection. Then the fetcher sends out the fetch requests to the remote controllers holding the records and blocks the map task. The fetch request will invoke a respond task in the remote controller and the map task will be unblocked when all the responses are received. An update task updates a local partition with the received intermediate results (from a map task) using the update function, while a respond task answers a fetch request using a local partition of the side-input collection.

Different from the pull-based shuffle mechanism in MapReduce-like systems (where reducers pull intermediate results from mappers), a push-based shuffle mechanism is used in Tangram, in which updates are pushed to the update partitions on a per-partition basis. Push-based shuffle can overlap network communication with the computation of map tasks, but the system needs to handle more complex read/write conflicts between tasks. To resolve the read/write conflicts for a partition, the controller enforces a simple access control strategy. It assumes that map and respond tasks read a partition, while update tasks write a partition. The controller ensures that writes to a partition are exclusive while reads are not. If there is an ongoing update task on a partition, then map tasks, respond tasks and other update tasks on the same partition will be blocked. If there is an ongoing map or respond task on a partition, then other map and respond tasks on this partition can still run but update tasks will be blocked.

The execution of a plan starts when the global scheduler instructs the local controller to push a number of map tasks to the map thread pool. When the controller receives a fetch/update request, it invokes a respond/update task. The respond/update task is pushed to the thread pool for execution if it satisfies the access control policy; otherwise, it will be inserted into a pending buffer. Once a task finishes, the controller will be notified and it will check the pending buffer to find tasks satisfying the access control policy and push them to the thread pool for execution. The local controller is implemented as a single-thread event loop and manages the pending buffer and all control-related data structures. The event-loop simplifies the implementation logic by avoiding complex locking.

5.2 Partition-Based Progress Control

As a partition is the granularity of task execution in Tangram, each partition can have its own progress. Therefore, Tangram uses a partition-based progress control mechanism to support the BSP, SSP and ASP execution models. Different from parameter-server-based systems, in which progress is associated with a worker, Tangram associates progress with a partition. The local controller records the map progresses of the local map partitions and the update progresses of the local update partitions. Note that when the map collection and the update collection are the same (e.g., in PageRank), a partition has both map progress and update progress. We will show that partition-based progress control also ensures correctness upon failure and improves recovery efficiency in Section 5.3.

At the start of a plan, the map progresses and the update progresses are initialized as zero. When a map task finishes, the map progress of the corresponding partition is incremented by one, and the update requests generated by this map task also carry the map progress (before increment). For an update partition, the local controller uses a bitmap (for each map progress) to record the map partitions for which update has already been committed. The controller sets the update progress of a partition as the minimum progress for which there are still missing updates. The controller assumes that a map partition will generate update for all partitions in the update collection and can confirm that all update requests are committed using the bitmap.

The controller sets its local progress as the minimum update progress of its partitions and reports it to the global scheduler. The global scheduler regards the minimum progress among workers as the global progress and broadcasts it to all workers upon changes. If the staleness is k and the global progress is m, the local controller will only schedule map tasks for its partitions with a progress no larger than $m+k$. An example of progress control is provided in Figure 1. Partition P1 in worker 0 has an update progress of 2 as it has not received

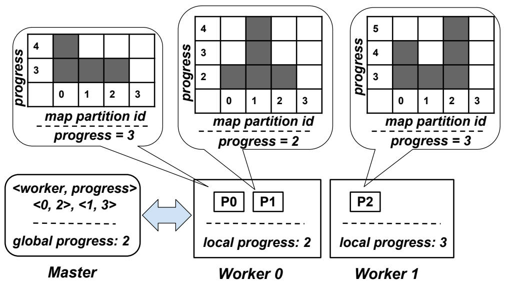


Figure 1: Example of progress management. Dark squares denote the received updates for each update collection.

the update with progress 2 from map partition 3. The local progress of worker 0 is 2 as the update progress of P0 and P1 are 3 and 2, respectively. Collecting the local progresses from worker 0 and worker 1, the master sets the global progress as 2. Once the update with progress 2 from map partition 3 is committed to P1, both the local progress of worker 0 and the global progress will be updated to 3.

Assume that there are M map partitions in total and a machine hosts n update partitions, the machine needs $O(MnT)$ memory to store the bitmap for a plan, where T is the number of active iterations (iterations for which there are uncommitted updates) and is usually small. We reduce the memory consumption of the bitmaps by deleting the bitmap for which all update requests have been committed and creating a bitmap only when receiving a new map progress. As the number of partitions is usually not large, the cost of progress control is acceptable as we will show in our experiments.

5.3 Context-Aware Failure Recovery

Tangram distinguishes two failure scenarios, i.e., *local failure* and *global failure*, and applies different recovery strategies.

Local failure is the case that the failed machines do not hold update partitions. Local failure does not directly affect the task execution on the healthy machines and is similar to losing RDDs with narrow dependency in Spark. In this case, Tangram only reloads the lost partitions on the healthy machines in parallel and sets their progresses as the current global progress. Some of the updates from the lost partitions may have been committed to the update collection and setting their progresses as the global progress may result in repetitive map tasks. Tangram rejects the repetitive updates generated by these map tasks using the bitmap.

Global failure happens when the failed machines contain partitions of the update collection. Examples include losing machines holding the model parameters in logistic regression or the rank values in PageRank. Global failure directly affects the computation on all machines and is similar to losing RDDs with wide dependency in Spark. In this case, Tangram reloads the lost mutable collection from the latest checkpoint and resets the global progress and the progresses of all partitions to the latest checkpoint. But for immutable collections, such as the graph links in PageRank, Tangram only reloads the lost partitions. The master assigns the tasks of loading the lost partitions to the healthy machines in a balanced manner so that the machines can recover from failure in parallel. Moreover, Tangram also respects the co-partitioning relation of the collections in failure recovery.

Tangram infers which collection is mutable and checkpoints only the mutable collections. Checkpointing is conducted in an asynchronous manner and on a per-partition basis, so that the execution of the entire plan does not need to be stopped. When the progress of an update partition reaches the checkpoint iteration, write access is blocked and a copy is written to disk along with the bitmap. Note that this checkpoint may be inconsistent, as the partition may have seen updated from iterations larger than the checkpoint iteration (under ASP or SSP). The bitmap is used to reject repetitive updates from these iterations during recovery. For pipelines that involve multiple plans, Tangram also checkpoints the mutable collections once a plan finishes so that failure recovery can be conducted inside a plan. Tangram only handles worker failures, while the master failure can be handled by a standby master (similar to the standby master in Spark [55]) but is not implemented in the current version.

5.4 Straggler Mitigation

Tangram uses partition migration for straggler handling. As update tasks and respond tasks are relatively lightweight, Tangram focuses on balancing the workload of map tasks by migrating map partitions. For non-iterative workloads (e.g., word count), the master monitors the number of ongoing and pending map tasks on each machine. If some machines do not have map tasks to run, the master migrates some of the map partitions from the heavily loaded machines to them. We allow the system to migrate map partitions with ongoing map tasks, which resembles speculative execution in Spark. Migration does not necessarily require data transfer from slow machines, as will be discussed later.

For iterative workloads, Tangram only handles macro stragglers, while micro stragglers can often be handled by asynchronous execution. By default (tunable by users), Tangram considers a machine as a macro straggler if its per-iteration time is more than 1.1 times of the median (of all machines) in three consecutive iterations or more than 1.5 times of the median in one iteration. Tangram also respects the co-partitioning relation among the collections to avoid high communication overhead after migration.

Tangram adopts different migration strategies for immutable and mutable partitions. For immutable partitions, the destination machines just load them from a shared storage system like HDFS rather than asking the source machine for transfer, since the source machine is already overloaded. For mutable partitions, Tangram uses a migration procedure similar to the two-stage migration in Piccolo [46].

The load of the system may be unbalanced due to skewed partition size (e.g., due to improper hash function). Currently, Tangram does not support online re-partitioning for workload redistribution. Similar to other systems, skewed partitions can be addressed by either fine-grained sharding (setting the number of partitions to be much larger than the number of machines) or providing a tailored partitioning function.

5.5 Communication Optimizations

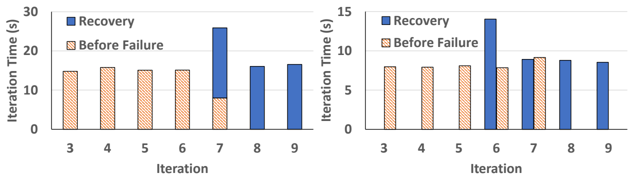
Delay Combiner. In Tangram, combining the updates from many map tasks leads to higher compression ratio, but sending out the updates immediately reduces latency. We provide a delay combiner, in which users can specify the granularity of combining with a *combine_timeout*. Setting *combine_timeout* to 0 sends out the updates immediately, while setting *combine_timeout* to *kMaxCombineTimeout* combines all local map outputs in an iteration.
Process Cache. The process cache in Tangram is similar to the one in Petuum. Previously fetched records of the side-input collection and their versions are kept in the cache. A new fetch request will not be sent if the records with the required version are already in the cache.
Local Zero-Copy Communication. Tangram utilizes local zero-copy communication whenever possible: if an update request is to be sent to a local partition, it will be moved to the local controller and can be directly accessed by the update task. Similarly, zero-copy communication is also used for fetching local objects.

6 Experiments

We implemented Tangram in about 16K lines of C++ code. The communication module was built using ZeroMQ [72] and libhdfs3 [32] was used to exchange data with HDFS without the JNI overhead. Source code for the system and the applications in Section 4 can be found at `https://github.com/Yuzhen11/tangram/`. We evaluated Tangram on a cluster of 20 machines connected with 1 Gbps Ethernet. Each machine is equipped with two 2.0GHz E5-2620 Intel(R) Xeon(R) CPU (12 physical cores in total), 48GB RAM, a 450GB SATA disk (6Gb/s, 10k rpm, 64MB cache), running on 64-bit CentOS release 7.2. We optimized the number of partitions for both Tangram and the systems we compared in the experiments.

6.1 Failure Recovery & Straggler Mitigation

Failure recovery and straggler mitigation are critical for data analytics in production. In this set of experiments, we show that Tangram achieves efficient failure recovery and effective straggler mitigation, even for workloads with mutable states, by distinguishing immutable and mutable collections.
Failure Recovery. We used two experiments to simulate different failure scenarios. In the first experiment, we unplugged



(a) Local Failure (b) Global Failure

Figure 2: Performance of failure recovery

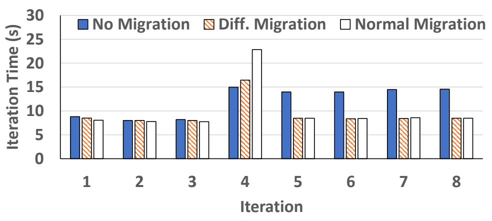


Figure 3: Performance of straggler handling

one machine holding the training data for K-means, which is a *local failure* according to Section 5.3. The second experiment unplugged one machine for PageRank, which corresponds to *global failure*.

We report the failure recovery performance under the two scenarios in Figure 2. For clearer presentation, we only plot the performance of Tangram, while we report the performance of Spark in text as a baseline. For local failure, Tangram took 17.8 seconds to reload the lost training data (∼6 GB) and finish the 7th iteration, while Spark took around 40 seconds to recover. Both Tangram and Spark did not restart the job from the latest checkpoint, but performed re-computation only on the lost partitions. In contrast, most of the mutable systems such as Petuum, PowerGraph and Naiad, would have to roll back to the latest checkpoint in case of any failure.

For the global failure (occurred at the 7th iteration), Tangram rolled back to the latest checkpoint (taken at the 5th iteration) and continued by re-executing the 6th iteration. The roll-back is necessary because some partitions of the (mutable) rank collection are lost. The longer recovery bar at the 6th iteration in Figure 2b includes the normal execution time for the iteration and another 5.2 seconds to reload the mutable collection (∼50 MB) and the lost immutable partitions (∼1 GB). In total, Tangram took 29 seconds to finish the 7th iteration, while Spark took 47 seconds. We note that Spark also requires a full re-computation from checkpoint in this case (i.e., long lineage with wide dependency [70]).

To provide a better picture of Tangram's failure recovery performance, we also implemented the baseline strategies (e.g., full reload, full checkpoint) in Tangram for a fair comparison. A full reload (as used in existing mutable systems) needs to load 121GB and 23GB data, and took 56.8 and 15.7 seconds (vs. Tangram's 17.8 and 5.2 seconds) for K-means and PageRank, respectively. In addition, distinguishing the immutable and mutable parts also results in more efficient checkpointing for PageRank, as checkpointing only the ranks

(~1GB) took 5 seconds, while a full checkpoint (~23GB) took 127 seconds. Tangram also supports asynchronous background checkpointing, which makes a copy of the mutable collection and writes the checkpoint in the background. Without background checkpointing, each checkpoint would take an extra 5 seconds for PageRank.

Straggler Mitigation. To test the performance of straggler mitigation, we used *cpulimit* tool to restrict one Tangram worker to have only 600% of the total 2400% cpu shares in the PageRank job at the 4th iteration, and the per-iteration time is reported in Figure 3. *Diff. migration* asks the straggler only for mutable partitions, while *Normal migration* asks the straggler for both mutable and immutable partitions. Both strategies were implemented in Tangram. The result shows that partition migration effectively reduces per-iteration time as an iteration took about 14.5 seconds without migration, but only 8.4 seconds with migration. In addition, distinguishing immutable and mutable collections also speeds up the migration. *Diff. migration* only requires the straggler to transfer the mutable partitions (ranks: ~50MB) and reloads the immutable partitions (links: ~ 1GB) from HDFS, which improves migration speed by approximately 38% (10 seconds vs. 16 seconds) compared with *Normal migration*.

6.2 Expressiveness and Efficiency

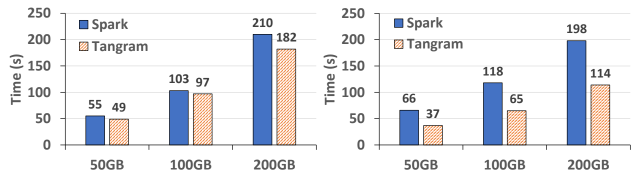
We have shown that the MapUpdate API is flexible and can express a wide variety of workloads in Section 4. In this set of experiments, we show that Tangram achieves comparable performance as specialized systems.

Bulk Processing. For non-iterative bulk processing workloads, e.g., MapReduce-style workloads, we tested word count and TF-IDF[6], and compared with Spark [70] (version 2.2.0). We replicated the Wikipedia corpus [19] to test the scalability of the systems and report the running time in Figure 4.

Tangram achieved slightly better performance compared with Spark for word count, but is 2x faster for TF-IDF. For fair comparison, we ensured that Spark does not write intermediate results to disk before shuffle. Both Tangram and Spark have high CPU utilization (over 80% for all cores overtime) and low disk utilization (less than 20% at most) for the two applications, similar to the results reported in [44]. We also tested the systems on a faster 10-Gbps network and on a single machine, and Tangram's performance advantage over Spark on TF-IDF is consistent in both settings, which shows that network communication is not the key factor that affects the performance of the systems on TF-IDF. We believe the language (C++ vs. Scala) and other system overheads are the main reasons for the performance difference. Tangram also achieves almost linear scaling when increasing dataset size.

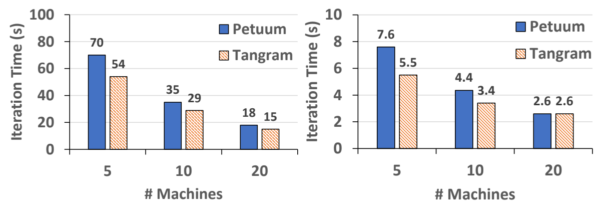
Iterative Machine Learning. For iterative machine learning workloads, we tested K-means [54] and SGD based logistic regression (LR). We used a dense dataset (mnist8m [33]) for

[6] https://en.wikipedia.org/wiki/Tf-idf



(a) Word Count (b) TF-IDF

Figure 4: Running time for word count and TF-IDF



(a) K-means (b) LR

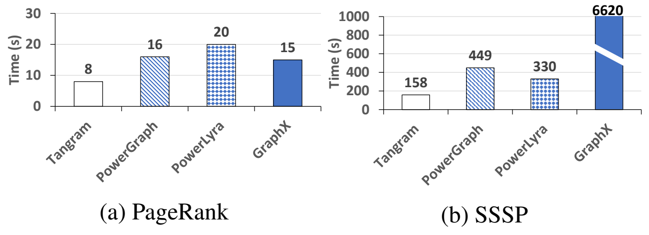
Figure 5: Per-iteration time for K-means and LR

K-means and a sparse dataset (webspam [59] with a sparsity of 2.24×10^{-4}) for LR, in order to test Tangram's performance under different data sparsity. We replicated the datasets 10 times for better scalability tests. BSP was used for K-means, while SSP ($s = 2$) was used for LR. We compared Tangram with the state-of-the-art parameter server system, Petuum Bösen [60]. We did not compare with Spark as it has been shown to be inefficient for iterative machine learning workloads compared with Petuum [29, 62].

Figure 5 reports the per-iteration time obtained by averaging 20 iterations, while varying the numbers of machines. Tangram's performance is very competitive compared with Petuum, as Tangram also supports optimizations generally used in parameter server based systems, such as process cache and message combining (Section 5.5). The scaling performance of Tangram and Petuum is better for K-means than for LR since K-means is CPU-bound, while LR is network-bound due to the large model.

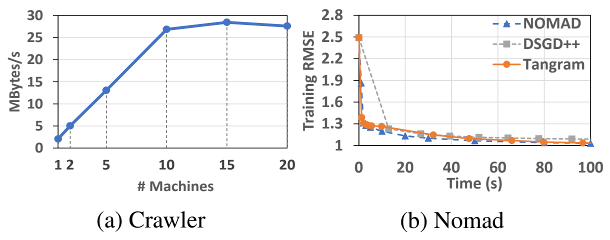
Graph Analytics. For graph analytics workloads, we compared with GraphX [22], PowerGraph [21] and PowerLyra [13]. GraphX is built on Spark and adopts immutable data abstraction, while PowerGraph and PowerLyra use mutable abstraction and support fine-grained state access. We tested PageRank and single source shortest path (SSSP), in BSP mode. We used the webuk graph [7], which has 133M vertices and 5.5B edges.

We report the per-iteration time for PageRank and the total running time for SSSP in Figure 6. Tangram achieves better performance than even the specialized systems. We found that this is because both PageRank and SSSP are network-bound, and message combining in Tangram (edge-cut + delay combiner) is more effective in reducing communication than other systems (vertex/hybrid-cut + combiner). GraphX outperforms PowerGraph and PowerLyra on PageRank as the workload is heavy and balanced in each iteration. In contrast, the access



(a) PageRank (b) SSSP

Figure 6: Comparison on PageRank and SSSP



(a) Crawler (b) Nomad

Figure 7: Performance of distributed crawler and Nomad

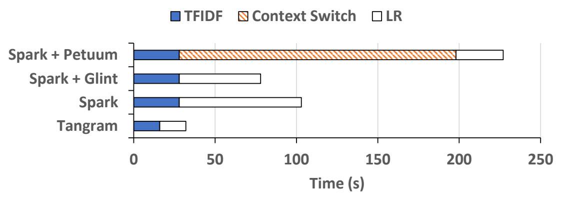


Figure 8: Completion time of the pipelined workload

pattern of SSSP is more sparse, and thus the lack of support for in-place updates renders GraphX inefficient.

Other Workloads. We also evaluated the performance of Tangram on a wider variety of computation patterns using distributed crawler and Nomad [69].

Figure 7a reports the download speed of Tangram-based crawler. The download speed of the crawler scales almost linearly with the number of machines and quickly consumes the download bandwidth of the whole cluster and reaches a plateau, which is similar to Piccolo-based crawler [46].

Nomad is an efficient SGD-based asynchronous algorithm for matrix factorization (MF) and has a complex computation pattern that migrates item latent factors among machines. We used the Yahoo! Music dataset [63] and compared Tangram-based Nomad with MPI-based Nomad [69] and DSGD++ [56] (another state-of-the-art MF algorithm). Figure 7b reports their training root mean square errors (RMSE). Tangram performs slightly worse than MPI-based Nomad initially but catches up later. Compared with MPI-based DSGD++, Tangram has better performance most of the time. Although the MPI-based implementations are efficient, Tangram offers very competitive performance and more user-friendly API.

Pipelined Workload. We implemented a simple pipelined workload that computes TF-IDF vectors with 2^{18} features from the 50GB English Wikipedia dataset and trains an LR model using gradient descent for 30 iterations. We compared

Table 3: Overhead of Task Management & Progress Control

App	Controller CPU %	Bitmap size
WordCount	1.12%	636KB
PageRank	0.99%	638KB
LR	0.29%	341KB
K-means	0.02%	4KB
Nomad	3.38%	153KB

Tangram with Spark, Spark + Glint [28], and Spark + Petuum. Glint is a built-in parameter server for Spark, and thus Spark + Glint uses Spark for TF-IDF and Glint for LR. Spark + Petuum uses Petuum for LR. Other systems (e.g., Naiad [39]) can also handle such a pipeline, but they mainly target at streaming workloads and have more expensive fault tolerance mechanisms. This experiment was conducted using a faster 10-Gbps network but the relative performance of the systems is consistent when running on a 1-Gbps network.

We report the execution time of the pipeline in Figure 8. Tangram used much less time than Spark mainly because Spark is not efficient for machine learning workloads [29, 62]. However, Spark + Petuum took even longer time, even though Petuum was much faster than Spark (29 seconds vs. 75 seconds) for LR. This is because there is a costly context switch overhead when moving from Spark to Petuum (around 170 seconds for dumping and loading the 45GB TF-IDF vectors) [7]. Spark + Glint removes the context-switch overhead and the performance is slightly better than Spark. However, adding dependencies (e.g., Glint) in Spark violates Spark's unified abstraction and breaks Spark's fault tolerance semantics.

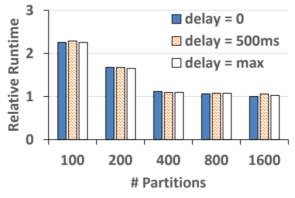
We also tested Flink [10], but LR was an order of magnitude slower on Flink compared with Spark [8]. Thus, we do not report the details of Flink's results. In comparison to these popular systems, the performance of Tangram in Figure 8 demonstrates its benefits as a general and efficient system for processing pipelined workloads.

6.3 Evaluation of System Designs

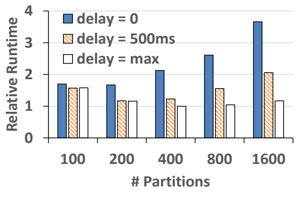
This set of experiments evaluates the effects of the system designs on the performance of Tangram. We first examine the average CPU consumption of the local controller on each worker and the total size of the bitmap used for progress control. Table 3 shows that the CPU consumption of the local controller is consistently low and the bitmap has a small memory footprint for all workloads. Nomad has the highest controller overhead as the algorithm needs to handle the frequent migration of item latent vectors. K-means has the smallest bitmap size because a single partition is used for the centers. In general, both controller CPU consumption and bitmap size increase with the number of partitions. Empirically, setting the number of partitions to 1-3 times the number

[7] Although in-memory caching systems like RAMCloud [41] or optimized distributed file systems like Tachyon [30] may reduce the context switch cost, the cost of dumping and loading the datasets is still non-negligible.

[8] The per-iteration time of LR using Flink Machine Learning library and using Flink DataSet API is 97x and 21x of that of Spark, respectively.



(a) K-means



(b) PageRank

Figure 9: Effects of partitioning and delay combiner

of cores achieves good performance. Thus, the cost of local task control and progress management is acceptable.

Next we examine the effect of the number of partitions. Figure 9 shows that the per-iteration time (as a ratio to the optimal setting) of K-means first decreases and is then stabilized as the number of partitions increases. This is because increasing the number of partitions improves the parallelism, until the cores are fully occupied (the cluster has 480 virtual cores). Beyond that point, using more partitions does not improve performance. For PageRank, the per-iteration time first slightly decreases and then increases with the number of partitions. This is because PageRank is network-bound and thus using more partitions results in more communication. The delay combiner in Section 5.5 can be used to reduce the communication overhead by merging the map outputs from multiple local partitions. By setting the combine timeout to maximum, the per-iteration time of PageRank stays almost constant when increasing the number of partitions beyond 400. For K-means, as it is CPU-bound due to a small number of parameters, using the delay combiner has almost no effect.

7 Related Work

Programming Model. MapReduce [17] has inspired the development of many data-parallel analytics frameworks with two coarse-grained high-order functions, *map* and *reduce*. Map-Reduce-Merge [67] extended MapReduce with a Merge phase to support the join of multiple heterogeneous datasets. HaLoop [9], Twister [18] and Map-Reduce-Update [8] adapted MapReduce for iterative computation. Other frameworks, e.g., Dryad [27], FlumeJava [11], Spark [70], Tez [52], etc., generalized the coarse-grained functional model by introducing dataflow graph, which enables easy construction of pipelines involving multiple stages. CIEL [40] and Ray [38] further support dynamic task graph. Flink [10] and Naiad [39] support the dataflow model on top of their streaming execution engines. Tensorflow [1] adopts dataflow graph to represent machine learning pipelines.

Some systems adopt the distributed shared memory (DSM) model. Piccolo [46] allows user-defined kernels to read and update distributed key-value tables in parallel. Parameter server systems, e.g., DistBelief [16], Project Adam [14], Parameter Server [31], Petuum [60,62], FlexPS [25], incorporate machine learning specialized optimizations such as bounded delay execution. DSM is flexible but the push/pull API is considered more low-level than the functional API in the dataflow model. Husky [66] adopts an object-oriented API to model different computational frameworks. GraphLab [34, 35] uses a data graph to represent computational structure and data dependencies for some machine learning problems. Graph processing frameworks [12, 15, 21, 22, 34, 36, 50, 51, 61, 64, 65, 74] usually expose vertex/edge/subgraph-centric programming models and incorporate graph specific optimizations.

Execution Model. Popular dataflow systems, e.g., Tez [52], DryadLINQ [68], and Spark [70], adopt a BSP execution model, in which a stage waits for its predecessors to finish. Specialized systems often adopt execution models tailored for their target workloads. GraphLab [34, 35] allows asynchronous vertex execution and uses distributed locking to resolve access conflict. Parameter server systems, e.g., Parameter Server [31], Petuum [60], adopt a bounded delay model (SSP) in which the progress differences among workers are bounded by a user-defined threshold. Maiter [73] and PowerGraph [21] support asynchronous execution optimized for graph workloads. In comparison, Tangram supports BSP, SSP and ASP, enabling it to efficiently process various types of workloads such as graph analytics, machine learning, etc.

Scheduling Model. Recent work [37, 43, 45, 53] observed that centralized scheduling is the bottleneck for scaling out when there are a large number of short-lived tasks. To reduce the control plane overhead, Drizzle [57] and Nimbus [37] cache the scheduling decision, while MonoSpark [42] and Canary [48] use local/distributed scheduler. Tangram avoids the centralized scheduling overhead by relying on the local controllers to schedule their own tasks. The global scheduler only launches plans and manages progress.

8 Conclusions

We proposed a programming model called MapUpdate to determine data mutability according to workloads, which not only brings good expressiveness but also enables a rich set of system features (e.g., asynchronous execution) and provides strong fault tolerance. We developed Tangram to support MapUpdate with novel designs such as partition-based progress control and context-aware failure recovery. We also incorporate optimization techniques such as process cache and partition migration. Our experiments show that Tangram is expressive and efficient, and achieves comparable performance with specialized systems for a wide variety of workloads. Our work demonstrates that we do not have to choose either mutable or immutable abstraction, but can embrace both of them in one unified framework to enjoy the best of both worlds.

Acknowledgments. We thank the reviewers and our shepherd Animesh Trivedi for their constructive comments that help improved the quality of the paper. This work was supported in part by ITF 6904945, and GRF 14208318 & 14222816.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.

[2] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[3] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, pages 185–198, 2013.

[4] G. Ananthanarayanan, M. C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. GRASS: trimming stragglers in approximation analytics. In *NSDI*, pages 289–302, 2014.

[5] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, pages 265–278, 2010.

[6] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative API for real-time applications in apache spark. In *SIGMOD*, pages 601–613, 2018.

[7] P. Boldi, M. Santini, and S. Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008.

[8] V. R. Borkar, Y. Bu, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Declarative systems for large-scale machine learning. *IEEE Data Eng. Bull.*, 35(2):24–32, 2012.

[9] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[10] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink$^{\text{TM}}$: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[11] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *SIGPLAN*, pages 363–375, 2010.

[12] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng. G-miner: an efficient task-oriented graph mining system. In *EuroSys*, pages 32:1–32:12, 2018.

[13] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: differentiated graph computation and partitioning on skewed graphs. In *EuroSys*, pages 1:1–1:15, 2015.

[14] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, pages 571–582, 2014.

[15] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[16] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, pages 1232–1240, 2012.

[17] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[18] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. C. Fox. Twister: a runtime for iterative mapreduce. In *HPDC*, pages 810–818, 2010.

[19] Enwiki Dump. https://dumps.wikimedia.org/enwiki/.

[20] Flink Parameter Server Limitations. https://github.com/gaborhermann/flink-parameter-server#limitations.

[21] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[22] 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 *OSDI*, pages 599–613, 2014.

[23] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Addressing the straggler problem for iterative convergent parallel ML. In *SoCC*, pages 98–111, 2016.

[24] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, 2013.

[25] Y. Huang, T. Jin, Y. Wu, Z. Cai, X. Yan, F. Yang, J. Li, Y. Guo, and J. Cheng. Flexps: Flexible parallelism control in parameter server architecture. *PVLDB*, 11(5):566–579, 2018.

[26] IndexedRDD for Apache Spark. `https://github.com/amplab/spark-indexedrdd`.

[27] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[28] R. Jagerman, C. Eickhoff, and M. de Rijke. Computing web-scale topic models using an asynchronous parameter server. In *SIGIR*, pages 1337–1340, 2017.

[29] J. Jiang, B. Cui, C. Zhang, and L. Yu. Heterogeneity-aware distributed parameter servers. In *SIGMOD*, pages 463–478, 2017.

[30] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SoCC*, pages 6:1–6:15, 2014.

[31] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[32] libhdfs3. `https://github.com/Pivotal-DataFabric/attic-libhdfs3`.

[33] G. Loosli, S. Canu, and L. Bottou. Training invariant support vector machines using selective sampling. In L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, editors, *Large Scale Kernel Machines*, pages 301–320. MIT Press, Cambridge, MA., 2007.

[34] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, pages 340–349, 2010.

[35] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[36] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[37] O. Mashayekhi, H. Qu, C. Shah, and P. Levis. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *ATC*, pages 513–526, 2017.

[38] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging AI applications. In *OSDI*, pages 561–577, 2018.

[39] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455, 2013.

[40] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: A universal execution engine for distributed data-flow computing. In *NSDI*, 2011.

[41] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. M. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramclouds: scalable high-performance storage entirely in DRAM. *Operating Systems Review*, 43(4):92–105, 2009.

[42] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *SOSP*, pages 184–200, 2017.

[43] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The case for tiny tasks in compute clusters. In *HotOS*, 2013.

[44] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun. Making sense of performance in data analytics frameworks. In *NSDI*, pages 293–307, 2015.

[45] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *SOSP*, pages 69–84, 2013.

[46] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, pages 293–306, 2010.

[47] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou. Real-time constrained cycle detection in large dynamic graphs. *PVLDB*, 11(12):1876–1888, 2018.

[48] H. Qu, O. Mashayekhi, C. Shah, and P. Levis. Decoupling the control plane from program control flow for flexibility and performance in cloud computing. In *EuroSys*, pages 1:1–1:13, 2018.

[49] B. Recht, C. Re, S. J. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[50] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: scale-out graph processing from secondary storage. In *SOSP*, pages 410–424, 2015.

[51] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

[52] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. C. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *SIGMOD*, pages 1357–1369, 2015.

[53] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *EuroSys*, pages 351–364, 2013.

[54] D. Sculley. Web-scale k-means clustering. In *WWW*, pages 1177–1178, 2010.

[55] Standby Masters with ZooKeeper. https://spark.apache.org/docs/latest/spark-standalone.html#standby-masters-with-zookeeper.

[56] C. Teflioudi, F. Makari, and R. Gemulla. Distributed matrix completion. In *ICDM*, pages 655–664, 2012.

[57] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, pages 374–389, 2017.

[58] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[59] S. Webb, J. Caverlee, and C. Pu. Introducing the webb spam corpus: Using email spam to identify web spam automatically. In *CEAS*, 2006.

[60] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *SoCC*, pages 381–394, 2015.

[61] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. Gram: scaling graph computation to the trillions. In *SoCC*, pages 408–421, 2015.

[62] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. In *SIGKDD*, pages 1335–1344, 2015.

[63] Yahoo! Webscoope. http://webscope.sandbox.yahoo.com/.

[64] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

[65] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW*, pages 1307–1317, 2015.

[66] F. Yang, J. Li, and J. Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *PVLDB*, 9(5):420–431, 2016.

[67] H. Yang, A. Dasdan, R. Hsiao, and D. S. P. Jr. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*, pages 1029–1040, 2007.

[68] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[69] H. Yun, H. Yu, C. Hsieh, S. V. N. Vishwanathan, and I. S. Dhillon. NOMAD: nonlocking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *PVLDB*, 7(11):975–986, 2014.

[70] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[71] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, pages 29–42, 2008.

[72] ZeroMQ. http://zeromq.org/.

[73] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. *IEEE Trans. Parallel Distrib. Syst.*, 25(8):2091–2100, 2014.

[74] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, pages 301–316, 2016.