Spring 2015

Dynamic re-optimization techniques for stream processing engines and object stores

Naresh Kumar Reddy Rapolu
Purdue University

Graduate School Form 30
Updated 1/15/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By NARESH KUMAR REDDY RAPOLU

Entitled

DYNAMIC RE-OPTIMIZATION TECHNIQUES FOR STREAM PROCESSING ENGINES AND OBJECT STORES

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

ANANTH GRAMA
Chair

SURESH JAGANNATHAN

PATRICK EUGSTER

SONIA FAHMY

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): ANANTH GRAMA

Approved by: WILLIAM GORMAN 4/14/2015

Head of the Departmental Graduate Program Date

DYNAMIC RE-OPTIMIZATION TECHNIQUES FOR STREAM PROCESSING ENGINES AND OBJECT STORES

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Naresh Kumar Reddy Rapolu

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2015

Purdue University

West Lafayette, Indiana

Dedicated to my family for their unconditional love, support and encouragement.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Ananth Grama, for his guidance and support. His *change-the-world* attitude towards research has motivated me to give that extra bit. The freedom he gave enabled my gradual growth from a student to a researcher. I'm deeply indebted to him for the belief he has in my abilities, inspite of all the set-backs along the way. Secondly, I would like to thank my mentor, Srimat Chakradhar, from NEC Laboratories. Apart from the internship opportunities, his constant feedback, monitoring, inspirational talks, focussed yet grounded nature, have set the tone for this disseration. I have been fortunate to have Suresh Jagannathan, Patrick Eugster and Sonia Fahmy on my PhD committee. Their insightful comments helped refine the dissertation.

I am indebted to the friendship and insight of my numerous friends at Purdue University including Karthik Kambatla, Adnan Hassan, KC Sivaramakrishnan, Sriharsha Gangam, Gowtham Kaki and many others. I would especially like to thank Jitendra Adapala. His belief in my abilities always surpassed my own assessment. Being my roommate and a close confidant, his impact on my life at Purdue is immeasurable. Towards the end, my philosophical interactions with Saurabh Misra, Siddharth Bhandari, Aakriti Jain, Gaurav Patankar and Akshay Ponda helped keep my sanity and focus intact.

My childhood friends (Nisha, Ramandeep, Munish, Sushanth and Joshua) have always been my pillars of strength, constantly re-assuring me that I have the potential to do good things in life. Friends I met later in life (Ramana Chakradhar, Sankalp Arrabolu, Shashank Chakelam, Anirudh Vemula, Kota Lakshminarayana, Debdutta Choudhary, Sandeep Chada and many more) have inspired me to strive hard to make the most of my talents.

Finally, I owe a debt of gratitude to my family, whose constant support and encouragement made all the difference in this long journey. Never did they doubt my completing this journey even when I doubted myself. Being a role-model all his life, my dad's pep-talks were the only thing that kept me moving.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

SPE	Stream Processing Engine
DAG	Directed Acyclic Graph of operators in a stream-processing engine, also referred to as topology
Entity Group	Pre-defined group of objects on which serializable transactions are supported
DT	Distributed Transaction on any set of objects in the object-store
LT	Local Transaction on a set of objects belonging to any single entity-group
MVCC	Multi Version Concurrency Control which uses version information on object to achieve serializability among transactions

ABSTRACT

Rapolu, Naresh K Reddy Ph.D., Purdue University, May 2015. Dynamic Re-optimization Techniques for Stream Processing Engines and Object Stores. Major Professor: Ananth Grama.

Large scale data storage and processing systems are strongly motivated by the need to store and analyze massive datasets. The complexity of a large class of these systems is rooted in their distributed nature, extreme scale, need for real-time response, and streaming nature. The use of these systems on multi-tenant, cloud environments with potential resource interference necessitates fine-grained monitoring and control. In this dissertation, we present efficient, dynamic techniques for re-optimizing stream-processing systems and transactional object-storage systems.

In the context of stream-processing systems, we present `VAYU`, a per-topology controller. `VAYU` uses novel methods and protocols for dynamic, network-aware tuple-routing in the dataflow. We show that the feedback-driven controller in `VAYU` helps achieve high pipeline throughput over long execution periods, as it dynamically detects and diagnoses any pipeline-bottlenecks. We present novel heuristics to optimize overlays for group communication operations in the streaming model.

In the context of object-storage systems, we present `M-Lock`, a novel *lock-localization* service for distributed transaction protocols on scale-out object stores to increase transaction throughput. Lock localization refers to dynamic migration and partitioning of locks across nodes in the scale-out store to reduce cross-partition acquisition of locks. The service leverages the observed object-access patterns to achieve lock-clustering and deliver high performance. We also present `TransMR`, a framework that uses distributed, transactional object stores to orchestrate and execute asynchronous components in amorphous data-parallel applications on scale-out architectures.

1 INTRODUCTION

Large scale storage and data-analysis systems have received significant interest and attention over the past decade. The term *BigData* is informally used to refer to this ecosystem. Several factors contribute to this growing interest. Large amounts of data is being generated by users on the web, as internet penetration increases through the use of mobile devices. Large datasets are also being collected by sensors and cameras deployed for traffic-monitoring in cities, peformance-monitoring in manufacturing facilities, anomaly-detection in nuclear-reactors, deep sky exploration, etc. Software systems are needed to store and analyze these datasets to gain actionable insights.

To reduce operational cost, large-scale storage and data processing is hosted on scale-out architectures, in which large number of machines based on off-the-shelf commodity hardware are used to build data-centers. In the context of cloud-based systems, software is hosted on virtual machines that potentially share the same physical machines. Tenant workloads using co-hosted virtual machines (sharing the physical machine) potentially experience resource (CPU/Network) interference. Software systems and programming models are being designed to efficiently execute traditional parallel and distributed computing workloads on such scale-out architectures.

1.1 Challenges Associated with Scalable Distributed Data Processing

Distributed software systems operating on scale-out architectures have the following requirements: i) on-demand scaling of computation and storage based on the application-needs; ii) coping with arbitrary computation or machine-failures, when operating on hundreds of machines; and iii) best-effort performance optimization in the presence of resource-interference, observed in multi-tenant, cloud-deployments.

MapReduce [1] (and its open-source version Hadoop) has become a popular programming model, as its runtime-engine aims to satisfy the above-mentioned requirements. However, the job execution latency in MapReduce-based systems is relatively high as they rely on conservative fault-tolerance methods – the intermediate state and job output is saved in a replicated distributed file system. Furthermore, when executing on a large-number of nodes, synchronization operations between processes become expensive. These operations are part of group-communication operations, barriers in bulk-synchronous-parallel (BSP) [2, 3] algorithms, etc. To reduce synchronization-overhead and decrease job latencies, apart from run-time optimizations, applications must exploit available algorithmic asynchrony. In particular, many machine learning algorithms [4–6] have shown to tolerate asynchrony to achieve high speedups, albeit with some loss in accuracy. However, orchestrating asynchronous application components at large-scale is challenging. Sophisticated frameworks are needed to ensure consistent application semantics while executing fault-prone, asynchronous computations at scale.

To process continuous streams of data at low latency, distributed stream processing engines, such as Borealis [7], TimeStream [8], Samza [9], and Storm [10] have been proposed. Stream engines code the application workflow as a directed acyclic graph (DAG) of operators, referred to as a topology. Tuples are processed in a pipelined fashion as they traverse through the topology. In this setting, even a single slow pipeline-stage affects the throughput of the entire pipeline. This problem is particularly pronounced when the system is executed in multi-tenant, cloud environments with potential resource-interference and node failures. To sustain high pipeline-throughput over long execution periods, it is necessary to dynamically detect and diagnose pipeline-bottlenecks.

Emerging online-learning applications have complex topologies and often use structured overlays for group-communication operations. For example, pipelined all-reduce overlay is frequently used to synchronize large state among operators. State-of-the-art schedulers take a static topology as input. However, for complex group communication operations such as all-reduce, the most efficient overlay structure depends on the network and compute resources allocated to the operators. For this reason, current schedulers are unable

to optimize complex communication structures, since they assume that the best topology is known a-priori. To detect and diagnose temporally varying pipeline-bottlenecks, stream-engines need a feedback-driven control loop. Furthermore, the diagnosis phase, which involves changing the topology routes, should cause least disruption to the tuple-throughput of the stream engine.

1.2 Challenges Associated with Scalable Storage Systems

Scale-out data stores, based on key-value abstractions are commonly used as backend or cloud storage for various applications. For example, Google's Bigtable [11] and Amazon's Dynamo [12] are used as database engines for various web services. However, these systems only offer single-key transactional guarantees. Systems like CloudDB [13, 14] and Megastore [15] were built to support OLTP applications on such stores. Megastore uses Bigtable [11] as the store, whereas CloudDB and Spire use HBase [16].

Scale-out stores typically rely on the key-value abstraction of an *object*. Several approaches have been proposed to support ACID transactions on scale-out object stores. A common strategy is to partition the store into *disjoint* groups of objects (also called entity-groups [15] or micro-shards [14]) and provide efficient transactional support *only* on objects in a single group.

Multi-version concurrency control (MVCC) is typically used for transactions within the entity-group (local-transactions or LT) [15, 17]. Transactions on objects in multiple entity-groups (distributed transactions or DT) use distributed-locking and two-phase commit [17]. However, as with all distributed protocols that use locks, the overhead of acquiring locks dominates the transaction latency.

Locks are typically implemented by augmenting data objects with an "isLocked" field. Using an atomic read-modify-write operation that is natively supported by the key-value stores, a data object can be locked by changing its "isLocked" field value. A transaction acquires the lock on the object before updating it. Although this approach is simple and efficient for modifying a single object, it is not suitable for multi-object distributed transac-

tions [17,18]. In a scale-out key-value store, data is routinely re-partitioned and re-assigned to other nodes to balance load across the cluster. Such movement of data scatters the locks on different nodes (lock dispersion), significantly increasing lock acquisition overhead. Moreover, re-partitioning of a set of data objects by the underlying key-value store is heavily influenced by their total size, which includes the size of all previous versions of the objects, the size of their indices, and the size of their caches. By grouping data objects and their locks, unnecessary movement of locks is triggered even though these locks do not contribute significantly to the size of the data objects. Furthermore, lock acquisition is sequential and synchronous (locks must be acquired one by one, in a pre-defined order, to prevent deadlocks), unlike data-updates and lock-releases, which can happen in parallel, and asynchronously.

To improve the efficiency of distributed transactions that use lock-based concurrency control protocols, lock-dispersion needs to be minimized. One potential method to reduce lock-dispersion is to minimize the dependence of lock-scaling from object-scaling through fine-grained dynamic tracking of lock-usage-patterns.

1.3 Problem Statement

This disseration aims to improve the efficiency of large-scale data-processing engines and large-scale distributed storage systems in multi-tenant, cloud environments. Specifically, we formulate the following problems, present hypotheses, and validate them.

- By exploiting algorithmic asynchrony, application performance can be significantly improved, mainly through reduction of synchronization overheads in distributed environments.
- Through the use of transactions on shared-state, large-scale data-processing systems can orchestrate asynchronous computations, enabling applications to exploit amorphous data-parallelism.

- Throughput of distributed transactions on large-scale key-value stores can be improved using dynamic lock-localization and lock-clustering techniques.

- Performance of large-scale stream processing systems can be significantly improved through fine-grained topology re-optimization by a feedback-driven, dynamic controller. By abstracting topologies with versioned route-maps, on-the-fly topology modification is enabled with least disruption to concurrent stream traffic.

- In the streaming model, group communication operations, such as *a*ll-reduce, can be significantly improved by using an overlay that minimizes the average completion time. Such a cost function considers the effect on concurrent, dependent stream traffic.

1.4 Contributions

The main contribution of this dissertation is to provide low-overhead, dynamic techniques to (i) improve throughput of stream-topologies that process continuous streaming data; (ii) improve efficiency of state-updates, which use lock-based distributed transactions, on key-value stores. The contributions include:

- *Efficient Execution of Asynchronous Algorithms in MapReduce*

  In Chapter 3, we present extensions to the MapReduce programming model to support asynchronous algorithms. We investigate the notion of partial synchronizations in iterative MapReduce applications to overcome global synchronization overheads. The proposed approach applies a locality-enhancing partition on the computation. Map-tasks execute local computations with (relatively) frequent local synchronizations, with less frequent global synchronizations. This approach yields significant performance gains in distributed environments, even though their serial operation counts are higher. In particular, (i) we motivate the need to extend MapReduce with constructs for asynchrony, (ii) we propose an API to facilitate partial synchronizations combined with eager scheduling and locality enhancing techniques, and (iii)

demonstrate performance improvements from our proposed extensions through a variety of applications from different domains.

- *Framework for Distributed Execution of Amorphous Data-parallel Applications*

  In Chapter 4, we present `TransMR`, a programming model and runtime system for distributed execution of amorphous data-parallel applications. `TransMR` extends the MapReduce programming model with constructs for transactional execution of computation units. MapReduce and related data-centric programming models have proven to be effective for a variety of large-scale distributed computations, in particular, those that manifest data parallelism. The fault-tolerance model underlying these programming environments relies on deterministic replay, which makes data-sharing (side-effects) across computations harder to support. This significantly limits the application scope of MapReduce and related models. In this dissertation: (i) we investigate data sharing (side-effects) in programming models operating on distributed key-value stores, specifically, the inconsistencies between the fault recovery mechanisms in execution and storage layers; (ii) we define semantics for a novel programming model, TransMR (Transactional MapReduce), which addresses these inconsistencies; and (iii) we demonstrate broad application scope and enhanced performance through data-sharing across computations for a prototype implementation of the proposed semantics.

- *Dynamic Lock Localization*

  In Chapter 5, we present *M-Lock*, a novel *lock localization* service for distributed transaction protocols (e.g., the protocol used by the Google App Engine [17]) on scale-out data stores to decrease the average latency of transactions. Lock localization refers to dynamic migration and partitioning of locks across various nodes in the scale-out store to reduce cross-partition acquisition of locks. When locks are localized (after separating from their data objects), fewer network round-trips and persistent-writes are necessary to acquire locks. In particular, (i) we propose new protocols to migrate, release and repatriate locks, (ii) we describe new policies

that control migration and repatriation of locks, (iii) we leverage application-specific affinity among data objects to achieve an application-specific localization of locks, and (iv) we present detailed experiments for the TPC-C benchmark.

- *Fine-grained Stream-topology Re-optimization*

  In Chapter 6, we present `VAYU`, a per-topology controller, which uses novel methods and protocols for dynamic, network-aware tuple-routing in the stream-topology. The controller relies on two novel techniques to achieve network-aware routing: (i) representing topology link structure using *route-maps*; (ii) consistent hashing for fine-grained key-space management and routing of tuples; feedback information about resource bottlenecks is translated to key-space mapping. We also present a light-weight, fault-tolerant protocol for atomic topology route-map update. By applying novel heuristics on the topology performance (feedback) metrics, the controller determines efficient route-maps, which encode tuple-routing information and also the topology link structure. These new route-maps are atomically injected into multiple operators, on-the-fly, using a light-weight, fault-tolerant protocol, for fast topology re-optimization.

- *Efficient Asynchronous Group Communication Overlays in the Streaming Model*

  In Chapter 7, we present novel overlay generation heuristics to improve efficiency of asynchronous group-communication operations in the streaming model. In particular, we focus on pipelined all-reduce overlays, typically used for model synchronization in online learning applications. We motivate the need for alternate cost functions, markedly different to the ones used in conventional messaging systems, to optimize overlays in the presence of concurrent stream traffic. We show that the heuristic-optimized overlays lead to significant performance gains even in the presence of complex network congestion patterns.

2 RELATED WORK

There are many previous studies that aim to improve efficiency of stream topologies and distributed transaction protocols. This chapter presents prior work relevant to the contributions in this dissertation *i.e.,* (i) On-the-fly re-optimization of stream topologies, (ii) Accelerating distributed transactions on scale-out key-value stores.

2.1 Programming Models for Data Parallel Computing

Several research efforts have targeted varions aspects of asynchronous algorithms. These include, novel asynchronous algorithms for different problems [2, 4–6], analysis of their convergence properties, and their execution on different platforms with associated performance gains. Recently, it has been shown that asynchronous algorithms for iterative numerical kernels significantly enhance performance on multicore processors [19]. In shared-memory systems, apart from the reduced synchronization costs, reduction in the off-chip memory bandwidth pressure due to increased data locality is a major factor for performance gains. Though the execution of asynchronous iterative algorithms on distributed environments has been proposed, constructs for asynchrony, impact on performance, and interactions with the API have not been well investigated. In this paper, we demonstrate the use of asynchronous algorithms in a distributed environment, prone to faults. With intuitive changes to the programming model of MapReduce, we show that data locality along with asynchrony can be safely exploited. Furthermore, the cost of synchronization (due to heavy network overheads) is significantly higher in a distributed setting compared to tightly-coupled parallel computers, leading to higher gains in performance and scalability.

Over the past few years, the MapReduce programming model has gained attention primarily because of its simple programming model and the wide range of underlying hardware environments. There have been efforts exploring both the systems aspects as well as

the application base for MapReduce. A number of efforts [20–22] target optimizations to the MapReduce runtime and scheduling systems. Proposals include dynamic resource allocation to fit job requirements and system capabilities to detect and eliminate bottlenecks within a job. Such improvements combined with our efficient application semantics, would significantly increase the scope and scalability of MapReduce applications. The simplicity of MapReduce programming model has also motivated its use in traditional shared memory systems [23].

A significant part of a typical Hadoop execution correponds to the underlying communication and I/O. This happens even though the MapReduce runtime attempts to reduce communication by trying to instantiate a task at the node or the rack where the data is present. Afrati et al. [1] study this important problem and propose alternate computational models for sorting applications to reduce communication between hosts in different racks. Our extended semantics deal with the same problem but, from an application's perspective, independent of the underlying hardware resources.

Recently, various forms of partial aggregations, similar to combiners in the MapReduce paper [24], have been shown to significantly reduce network overheads during global synchronization [25]. These efforts focus on different mathematical properties of aggregators (commutative and associative), which can be leveraged by the run-time to dynamically setup a pipelined tree-structured partial aggregation. These efforts do not address the problem of reducing the number of global synchronizations. In contrast, we focus on the algorithmic properties of the application to reduce the number of global synchronizations and its associated network overheads. By combining optimizations such as tree-structured partial aggregation, with capabilities of the proposed local reduce operations, we can reduce network overhead further.

The `TransMR` model supports transactional execution of distributed computations through the notion of a mutable shared state. Spark [26] and Piccolo [27] propose the use of shared state for distributed computations to achieve different goals. Spark uses read-only

[1]Foto N. Afrati and Jeffrey D. Ullman: A New Computation Model for Rack-based Computing. http://infolab.stanford.edu/ ullman/pub/mapred.pdf

shared data to build working-sets for concurrent map/reduce function invocations. Piccolo proposes the use of mutable in-memory tables to store data shared by concurrent threads. Piccolo's fault-tolerance and recovery model based on periodic, user-assisted checkpointing through distributed snapshots makes it hard to realize (efficient) transactional execution. Specifically, when any of the nodes fail, all of them have to be halted and rolled back to a consistent snapshot; unless checkpointing is executed at high frequency, it is hard to reason about the transactional behavior of processes and the consequent effect-propagation through shared state.

Google proposed Pregel [2] for large-scale graph-processing, based on the bulk synchronous parallel (BSP) programming model. Pregel (or BSP) does not support transactions and hence disallows speculative execution. Realizing Boruvka's MST application in Pregel, consequently, would involve algorithmically identifying and executing the non-conflicting operations at each stage. To avoid conflicting operations, the algorithm should be executed as a series of iterations (called steps in Pregel). Each iteration needs to compute the set of non-conflicting operations and execute them. Computing the set of non-conflicting operations is itself quite involved – making the program significantly more sophisticated.

In recent years, several systems have been proposed to increase the applicability of MapReduce. MapReduce Online [28] streams the data between map and reduce phases supporting pipelined execution, continuous queries, and online aggregation. Dryad [29] supports acyclic tasks and CIEL [30] adds support for dynamic task graphs particularly useful for dynamic-programming based applications. While these efforts have similar goals of increasing applicability, they do not address applications with multiple computational units accessing shared data-structures in a faulty environment.

2.2 Concurrency Control Protocols for Scale-out Key-value Stores

A number of recent systems support distributed transactions on key-value stores. They implement different concurrency control protocols depending on the consistency guarantee

and data-model supported by the system – Deuteronomy [31] employs two-phase locking for serializability; Megastore [15] employs optimistic concurrency, albeit for a hierarchical data-model; Percolator [18] uses optimistic concurrency with two-phase commit, but offers only snapshot isolation. Elastras [32] allows only per-partition transactions.

Deuteronomy [31, 33] was the first system to propose complete separation of the transaction manager (the mechanism to store locks) from the storage manager (the mechanism that stores data). In contrast, we argue for selective and adaptive separation of locks, depending on the application's needs, thereby benefiting both single-key and multi-key updates. Furthermore, Deuteronomy does not exploit lock localization on the distributed lock service, a critical component of our system.

CloudTPS [34] offers serializability through two phase commit and timestamp ordering across multiple transaction managers. However, their transaction managers (TMs) are statically defined, similar to H-WALs in our case. The techniques proposed in this paper can be used in such systems as CloudTPS to create on-the-fly transaction managers (TMs) that localize locks. G-Store [35] proposes forming key-groups for a long-standing application before it can execute transactions on the key-group. This technique reduces the distributed locking overhead, but the constraint that a key cannot belong to multiple key-groups severely limits the applicability of G-Store.

Recently, lock-free transactional support [36] and pre-ordered transaction execution [37] have been proposed on data stores. In contrast, we focus on lock-based concurrency control techniques, which are used in several systems.

Lock managers (Zookeeper [38]) have been proposed to store critical cluster management information such as node leases. In these systems, the entire lock-state is confined to a single-node, with synchronous replication for fault-tolerance. Furthermore, they are designed for read-heavy workloads. Due to these limitations, they cannot be used to handle write-locks for all data objects in the store. In M-Lock, the distributed lock managers are a part of the object-store and they benefit from the scalable and fault-tolerant properties of the object-store.

2.3 On-the-fly Stream Topology Re-optimization

Stream processing systems were initially designed to process continuous database queries on incoming event streams under low latency. These events could be generated by sensor networks in industrial process control, stock quotes, weather monitoring services, etc. Early stream processing systems include Aurora [39], Borealis [7] and System S [40] from IBM. Recently, several open-source stream processing systems have been developed. Examples include S4 [41] by Yahoo!, Storm [10] by Twitter, and Samza [9] by Linkedin. These systems have been designed specifically for analyzing clickstream data generated by logging online user activity on websites. Current generation stream-engines focus on enabling high-throughput stream processing along with new features such as: guaranteed message processing, fault-tolerant operator state management, and efficient scale-out mechanisms. Recently, several research efforts have investigated efficient mechanisms for fault-tolerant state management and operator scale-out [42, 43]. In contrast, we focus on heuristics and protocols for fast topology re-optimization, to maintain high stream-throughput at all times.

The TimeStream [8] system uses a technique called *resilient substituion* to scale-out arbitrary topologies. To scale a particular operator, the technique finds the smallest directed acyclic graph (DAG) containing the operator, and stops the inflow of tuples into this DAG. The DAG is then scaled-out by adding more nodes, and the flow is restarted. By providing generic mechanisms for scale-out, these techniques incur high overhead. In contrast, we propose light-weight protocols for load-balancing and route-modifications using global acking framework.

Workflow re-optimization has been proposed for batch processing systems such as MapReduce [44]. Their focus is on improving the job workflow based on statistics collected from previous job executions. Examples of workflow changes include: pushing filters upstream, changing the algorithms used for joins based on data-sizes, etc. In constrast, we focus on optimizing group communication topologies and fine-grained assignment of operators to resources.

Proposed schedulers for stream-engines, such as COLA [45], SODA [46], do not consider optimizing complex group communication topologies. Furthermore, their resoure-assignment framework is coarse-grained, where an entire operator-replica is considered for assignment. In contrast, by leveraging our *route-maps* representation of the topology, we separately optimize the group-communication operations irrespective of the original stream-topology. Furthermore, by using consistent-hashing for tuple-routing, we propose methods for fine-grained key-space (bucket) assignment.

Improving support for complex analytics applications in databases is an emerging field. Many new system designs have been recently proposed [47, 48]. In constrast, we focus on improving online analytics in the streaming model.

3 ASYNCHRONOUS ALGORITHMS IN MAPREDUCE

Motivated by the large amounts of data generated by web-based applications, scientific experiments, business transactions, etc., and the need to analyze this data in effective, efficient, and scalable ways, there has been significant recent activity in developing suitable programming models, runtime systems, and development tools. The distributed nature of data sources, coupled with rapid advances in networking and storage technologies naturally motivate abstractions for supporting large-scale distributed applications.

Asynchronous algorithms have been shown to enhance the scalability of a variety of algorithms in parallel environments. In particular, a number of unstructured graph problems have been shown to utilize asynchrony effectively to trade-off serial operation counts with communication costs. The increased communication costs in distributed settings further motivates the use of asynchronous algorithms. However, implementing asynchronous algorithms within traditional distributed computing frameworks presents challenges. These challenges, their solutions, and resulting performance gains form the focus of this chapter.

With a view to supporting large-scale distributed applications in unreliable wide-area environments, Dean and Ghemawat proposed a novel programming model based on `maps` and `reduces`, called MapReduce [24]. The inherent simplicity of this programming model, combined with underlying system support for scheduling, fault tolerance, and application development, make MapReduce an attractive platform for diverse data-intensive applications. Indeed, MapReduce has been used effectively in a wide variety of data processing applications. The large volumes of data processed at Google, Yahoo, and Facebook, stand testimony to the effectiveness and scalability of the MapReduce paradigm. The open-source implementation of MapReduce, Hadoop MapReduce [49] serves as a development testbed for a wide variety of distributed data-processing applications.

A majority of the applications currently executing in the MapReduce framework have a data-parallel, uniform access profile, which makes them ideally suited to `map` and `reduce`

abstractions. Recent research interest, however, has focused on more unstructured applications that do not lend themselves naturally to data-parallel formulations. Common examples of these include sparse unstructured graph operations (as encountered in diverse domains including social networks, financial transactions, and scientific datasets), discrete optimization and state-space search techniques (in business process optimization, planning), and discrete event modeling. For these applications, there are two major unresolved questions: (i) can the existing MapReduce framework effectively support such applications in a scalable manner? and (ii) what enhancements to the MapReduce framework would significantly enhance its performance and scalability without compromising desirable attributes of programmability and fault tolerance?

This chapter primarily focuses on the second question – namely, it seeks to extend the MapReduce semantics to support specific classes of unstructured applications on large-scale distributed environments. Recognizing that one of the key bottlenecks in supporting such applications is the global synchronization between the `map` and `reduce` phases, it introduces notions of partial synchronization and eager scheduling. The underlying insight is that for an important class of applications, algorithms exist that do not need global synchronization for correctness. Specifically, while global synchronizations optimize serial operation counts, violating these synchronizations merely increases operation counts without impacting correctness of the algorithm. Common examples of such algorithms include, computation of eigenvectors (pageranks) through (asynchronous) power methods, branch-and-bound based discrete optimization with lazy bound updates, computing all-pairs shortest paths in sparse graphs, constraint labeling and other heuristic state-space search algorithms. For such algorithms, a global synchronization can be replaced by concurrent partial synchronizations. However, these partial synchronizations must be augmented with suitable locality enhancing techniques to minimize their adverse effect on operation counts. These locality enhancing techniques typically take the form of min-cut graph partitioning and aggregation in graph analyses, periodic quality equalization in branch-and-bound, and other such operations that are well known in the parallel processing community. Replacing global synchronizations with partial synchronizations also allows us to schedule

subsequent `maps` in an eager fashion. This has the important effect of smoothing load imbalances associated with typical applications.

The techniques proposed in this chapter combine partial synchronizations, locality enhancement, and eager scheduling, along with algorithmic asynchrony to deliver distributed performance improvements of upto 800% (and beyond in some cases). Importantly, our proposed enhancements to programming semantics do not impact application programmability. We demonstrate all of our results on an Amazon EC2 8-node cluster, which involves real-world cloud latencies, in the context of *PageRank*, clustering (*K-Means*), and *Shortest Path* implementations. These applications are selected because of their ubiquitous interaction patterns, and are representative of a broad set of application classes.

The rest of the chapter is organized as follows: section 5.1 provides a more comprehensive background on MapReduce, Hadoop, and motivates the problem; section 3.2 provides an API to realize partial synchronizations; section 4.3 discusses our implementations of our proposed API, *PageRank*, *Shortest Path* and *K-Means* clustering in the context of the API and analyze the performance gains of our approach.

3.1 Background and Motivation

The primary design motivation for the functional MapReduce abstractions is to allow programmers to express simple concurrent computations, while hiding the cumbersome details of parallelization, fault-tolerance, data distribution, and load balancing in a single library [24]. The simplicity of the API makes programming relatively easy. Programs in MapReduce are expressed as `map` and `reduce` operations. The `map` phase takes in a list of key-value pairs and applies a programmer-specified function independently on each pair in the list. The `reduce` phase operates on a list indexed by a key, of all corresponding values, and applies the `reduce` function on the values; and outputs a list of key-value pairs. A phase involves distributed execution of tasks (application of the user-defined functions on part of the data). The `reduce` phase must wait for all the `map` tasks to complete, since it requires all the values correponding to each key. In order to reduce the network overhead, a

Combiner is often used to aggregate over keys from `map` tasks executing on the same node. Fault tolerance is achieved through deterministic-replay, i.e., scheduling failed computations on another running node. Most applications require iterations of MapReduce jobs. Once the `reduce` phase terminates, the next set of `map` tasks can be scheduled. As may be expected, for many applications, the dominant overhead in the program is associated with the global synchronizations between the `map` and `reduce` phases. When executed in wide-area distributed environments, these synchronizations often incur substantial latencies associated with underlying network and storage infrastructure.

To alleviate the overhead of global synchronization, we propose partial synchronizations (synchronization only across a subset of `maps`) that takes significantly less time, depending on where the maps execute. We observe that in many parallel algorithms, frequent partial synchronizations can be used to reduce the number of global synchronizations. The resulting algorithm(s) may be suboptimal in serial operation counts, but are be more efficient and scalable in a MapReduce framework. A particularly relevant class of algorithms where such tradeoffs are possible are iterative techniques applied to unstructured problems (where the underlying data access patterns are unstructured). This broad class of algorithms underlies applications ranging from *PageRank* to sparse solvers in scientific computing applications, and clustering algorithms. Our proposed API incorporates a two-level scheme to realize partial synchronization in MapReduce, described in detail in section 3.2.

We illustrate the concept using a simple example – consider *PageRank* computations over a network, where the rank of a node is determined by the rank of its neighbors. In the traditional MapReduce formulation, during each iteration, `map` involves each node pushing its *PageRank* to all its outlinks and `reduce` accumulates all neighbors' contributions to compute *PageRank* for the corresponding node. These iterations continue until the *PageRanks* converge. An alternate formulation would partition the graph; each `map` task now corresponds to the local *PageRank* computation of all nodes within the sub-graph (partition). For each of the internal nodes (nodes that have no edges leaving the partition), a partial reduction accurately computes the rank (assuming the neighbors' ranks were accurate to begin with). On the other hand, boundary nodes (nodes that have edges leading

to other partitions) require a global reduction to account for remote neighbors. It follows therefore that if the ranks of the boundary nodes were accurate, ranks of internal nodes can be computed simply through local iterations. Thus follows a two-level scheme, wherein partitions (`maps`) iterate on local data to convergence and then perform a global reduction. It is easy to see that this two-level scheme increases the serial operation count. Moreover, it increases the total number of synchronizations (partial + global) compared to the traditional formulation. However, and perhaps most importantly, it reduces the number of global reductions. Since this is the major overhead, the program has significantly better performance and scalability.

Indeed optimizations such as these have been explored in the context of traditional HPC platforms as well with some success. However, the difference in overhead between a partial and global synchronization in relation to the intervening useful computation is not as large for HPC platforms. Consequently, the performance improvement from algorithmic asynchrony is significantly amplified on distributed platforms. It also follows thereby that performance improvements from MapReduce deployments on wide-area platforms, as compared to single processor executions are not expected to be significant unless the problem is scaled significantly to amortize overheads. However, MapReduce formulations are motivated primarily by the distributed nature of underlying data and sources, as opposed to the need for parallel speedup. For this reason, performance comparisons must be with respect to traditional MapReduce formulations, as opposed to speedup and efficiency measures more often used in the parallel programming community. While our development efforts and validation results are in the context of *PageRank*, *K-Means* and *Shortest Path* algorithms, concepts of partial reductions combined with locality enhancing techniques and eager `map` scheduling apply to broad classes of iterative asynchronous algorithms.

We seek to answer the following key questions relating to the application scope and performance of MapReduce in the context of applications that tolerate algorithmic asynchrony: (i) what are suitable abstractions (MapReduce extensions) for distributed asynchronous algorithms? (ii) for an application class of interest, can the performance benefits of localization, partial synchronization, and eager scheduling of `maps` overcome the sub-

optimality in terms of serial operation counts, and (iii) can this framework be used to deliver scalable and high performance over wide-area distributed systems?

This chapter makes the following specific contributions —

- Motivates the use of MapReduce for implementing asynchronous algorithms in a distributed setting.
- Proposes partial synchronization and an associated API to alleviate the overhead due to the expensive global synchronization between `map` and `reduce` phases. Global synchronizations limit asynchrony.
- Demonstrates the use of partial synchronization and eager scheduling in combination with coarse-grained, locality enhancing techniques.
- Evaluates the applicability and performance improvements due to the aforementioned techniques on a variety of applications – *PageRank*, *Shortest Path*, and *K-Means*.

3.2 Proposed API

In this section, we present our API for the proposed partial synchronization and discuss its effectiveness. Our API is built on the rigorous semantics for iterative MapReduce, we propose in the associated technical report [50]. As mentioned earlier, our API for iterative MapReduce comprises a two-level scheme – local and global `map` and `reduce`. We refer to the regular MapReduce with global synchronizations as global MapReduce. A *global map* takes a partition as input, and involves invocation of *local map* and *local reduce* functions iteratively on the partition. The *local reduce* operation applies the specified reduction function to only those key-value pairs emanating from *local map* functions. Since partial synchronization suffices, *local map* operations corresponding to the next iteration can be eagerly scheduled. The *local map* and *local reduce* operations can use a thread-pool to extract further parallelism.

Often, the *local* and *global* `map`/`reduce` operations are functionally the same and differ only in the data they are applied on. Given a regular MapReduce implementation, it is fairly straight-forward to generate the *local map* and *local reduce* functions from the semantics explained in the technical report [50], thus not increasing the programming complexity. In the traditional MapReduce API, the user provides `map` and `reduce` functions along with the functions to split and format the input data. To generate the *local map* and *local reduce* functions, the user must provide functions for termination of global and local MapReduce iterations, and functions to convert data into the formats required by the *local map* and *local reduce* functions.

However, to accommodate greater flexibility, we propose use of four functions — `gmap`, `greduce`, `lmap` and `lreduce`; `gmap` invoking `lmap` and `lreduce` functions as described in section 4.3. Functions *Emit()* and *EmitIntermediate()* support data-flow in traditional MapReduce. We introduce their local equivalents — *EmitLocal()* and *EmitLocalIntermediate()*. Function `lreduce` operates on the data emitted through *EmitLocalIntermediate()*. At the end of local iterations, the output through *EmitLocal()* is sent to the *global reduce*; otherwise, *local map* receives it as input. Section 4.3 describes our implemenation of the API and our implementations of *PageRank*, *Shortest Path*, and *K-Means* using the proposed API; demonstrating its ease of use and effectiveness in improving the performance of applications using asynchronous algorithms.

3.3 Implementation and Evaluation

In this section, we describe our implementation of the API and the performance benefits from the proposed techniques of locality-enhanced partitioning, partial synchronization, and eager scheduling. We consider three applications — *PageRank*, *Shortest Path*, and *K-Means* to compare general MapReduce implementations with their modified implementations.

Our experiments were run on an 8-node Amazon EC2 cluster of large instances. This reflects the characteristics of a typical cloud environment. Also, it allows us to monitor

Table 3.1: Measurement Testbed and Software

Amazon EC2	4 64 bit EC2 Compute Units
8 Large Instances	7.5 GB RAM, 2 x 420 GB storage
Software	Hadoop 0.20.1, Java 1.6
Heap space	4 GB per slave

the utilization and execution of `map` and `reduce` tasks. Table 3.1 presents the physical resources, software, and restrictions on the cluster.

3.3.1 API Implementation

As in regular MapReduce, our execution also involves `map` and `reduce` phases; each phase executing tasks one one each machine. Each `map`/`reduce` task involves the application of `gmap`/`greduce` functions to corresponding data. Within the `gmap` function we execute *local MapReduce* iterations.

Figure 3.1 descibes our construction of `gmap` from the user-defined functions — *lmap* and *lreduce*. The argument to gmap is a <key, value> list(*xs*), on which the *local* MapReduce operates. `lmap` takes an element of *xs* as input, and emits its output by invoking *EmitLocalIntermediate()*. `lreduce` operates on this local intermediate data. A hashtable is used to store the intermediate and final results of the *local* MapReduce. Upon local convergence, `gmap` outputs the contents of this hashtable. `greduce` acts as the normal *global reduce* on `gmap`'s output. Such an implementation allows the use of other optimizations like Combiners in conjunction.

The rest of the section describes benchmark applications, their nominal and eager (partial synchronization with eager scheduling) implementations, and corresponding performance gains. We discuss *PageRank* in detail to illustrate our approach; *Shortest Path* and *K-Means* are discussed briefly in the interest of space.

```
gmap(xs : X list) {

  while(no-local-convergence-intimated) {

    lmap(x); // emits lkey, lval

    lreduce();
  }

  for each value in lreduce-output{
      EmitIntermediate(key, value);
  }
}
```

Figure 3.1.: Construction of `gmap` from `lmap` and `lreduce`

3.3.2 PageRank

The *PageRank* of a node is the scaled sum of the *PageRanks* of all of its neighbors, given by the following expression:

$$PR_d = (1 - \chi) + \chi * \sum_{(s,d)\epsilon E} s.pagerank/s.outlinks \tag{3.1}$$

where χ is the damping factor, $s.pagerank$ and $s.outlinks$ correspond to the *PageRank* and the out-degree of the source node, respectively.

The asynchronous *PageRank* algorithm involves an iterative two step method. In the first step, the *PageRank* of each node is sent to all its outlinks. In the second step, the *PageRanks* received at each node are aggregated to compute the new *PageRank*. The *PageRanks* change in each iteration, and eventually converge to the final *PageRanks*. For nominal as well as eager implementations, we use a graph represented as adjacency lists as input. All

nodes have an initial *PageRank* of 1. We define convergence by a bound on the norm of difference (infinite norm of 10^{-5} in our case).

General *PageRank*

The general MapReduce implementation of *PageRank* iterates over a `map` task that emits the *PageRanks* of all the source nodes to the corresponding destinations in the graph, and a `reduce` task that accumulates *PageRank* contributions from various sources to a single destination. In the actual implementation, the `map` function emits tuples of the type $< d_n, p_n >$, where d_n is the destination-node, and p_n is the *PageRank* contributed to this destination node by the source. The `reduce` task operates on a destination node, gathering the *PageRanks* from the incoming source nodes and computes a new *PageRank*. After every iteration, the nodes have renewed *PageRanks* which propagate through the graph in subsequent iterations until they converge. One can observe that a small change in the *PageRank* of a single node is broadcast to all the nodes in the graph in successive iterations of MapReduce, incurring a potentially significant cost.

Our baseline for performance comparison is a MapReduce implementation for which `maps` operate on complete partitions, as opposed to single node adjacency lists. We use this as a baseline because the performance of this formulation was noted to be on par or better than the adjacency-list formulation. For this reason, our baseline provides a more competitive implementation.

Eager *PageRank*

We begin our description of Eager *PageRank* with an intuitive illustration of how the underlying algorithm accommodates asynchrony. In a graph with specific structure (say, a power-law type distribution), one may assume that each hub is surrounded by a large number of spokes, and that inter-hub edges are comparatively infrequent. This allows us to relax strict synchronization on inter-hub edges until the subgraph in the proximity of a hub has relatively self-consistent *PageRanks*. Disregarding the inter-hub edges does not lead

to algorithmic inconsistency since, after few local iterations of MapReduce calculating the *PageRanks* in the subgraph, there is a global synchronization (following a *global map*), leading to a dissemination of the *PageRanks* in this subgraph to other subgraphs via inter-hub edges. This propagation imposes consistency on the global state. Consequently, we update only the (neighboring) nodes in the smaller subgraph. We achieve this by a set of iterations of *local* MapReduce as described in the API implementation. This method leads to improved efficiency if each `map` operates on a hub or a group of topologically localized nodes. Such topology is inherent in the way we collect data as it is crawler-induced. One can also use one-time graph partitioning using tools like Metis [1]. We use Metis since our test data set is not partitioned a-priori.

In the Eager *PageRank* implementation, the `map` task operates on a subgraph. *Local* MapReduce, within the *global map*, computes the *PageRank* of the constituent nodes in the subgraph. Hence, we run the *local* MapReduce to convergence. Instead of waiting for all the other *global map* tasks operating on different subgraphs, we eagerly schedule the next *local map* and *local reduce* iterations on the individual subgraph inside a single *global map* task. Upon local convergence on the subgraphs, we synchronize globally, so that all nodes can propagate their computed *PageRanks* to other subgraphs. This iteration over *global* MapReduce runs to convergence. Such an Eager *PageRank* incurs more computational cost, since local reductions may proceed with imprecise values of global *PageRanks*. However, the *PageRank* of any node propagated during the *global reduce* is representative, in a way, of the subgraph it belongs to. Thus, one may observe that the *local* and *global reduce* functions are functionally identical. As the subgraphs (partitions) have approximately the same number of edges, we expect similar number of local iterations in each *global map*. However, if the convergence rates are very different, the global synchronization requires waiting for all partitions to converge.

Note that in Eager *PageRank*, *local reduce* waits on a *local* synchronization barrier, while the *local maps* can be implemented using a thread pool on a single host in a cluster.

[1] METIS. http://glaros.dtc.umn.edu/gkhome/views/metis

The local synchronization does not incur any inter-host communication delays. This makes associated overheads considerably lower than the *global* overheads.

Input data

Table 3.2: *PageRank* Input Graph Properties

Input graphs	**Graph A**	**Graph B**
Nodes	280,000	100,000
Edges	3 million	3 million
Damping factor	0.85	0.85

Table 3.2 describes the two graphs used as input for our experiments on *PageRank*, both conforming to power law distributions. Graph A has 280K nodes and about 3 million edges. Graph B has 100K nodes and about 3 million edges. We use preferential attachment [51] to generate the graphs using igraph [2]. The algorithm used to create the synthetic graphs is described below, along with its justification.

Preferential attachment based graph generation Test graphs are generated by adding vertices one at a time — connecting them to *numConn* vertices already in the network, chosen uniformly at random. For each of these *numConn* vertices, *numIn* and *numOut* of its inlinks and outlinks are chosen uniformly at random and connected to the joining vertex. This is done for all of the newly connected nodes to the incoming vertex. This method of creating a graph is closely related to the evolution of online communities, social networks, the web, *etc.* This procedure increases the probability of highly reputed nodes getting linked to new nodes, since they have greater likelihood of being in an inlink from other randomly chosen sites.

Figures 3.2 and 3.3 show the distribution of inlinks for the two input graphs. The best line fit gives the power-law exponent for the two graphs showing their conformity with a

[2]The Igraph Library. http://igraph.sourceforge.net/

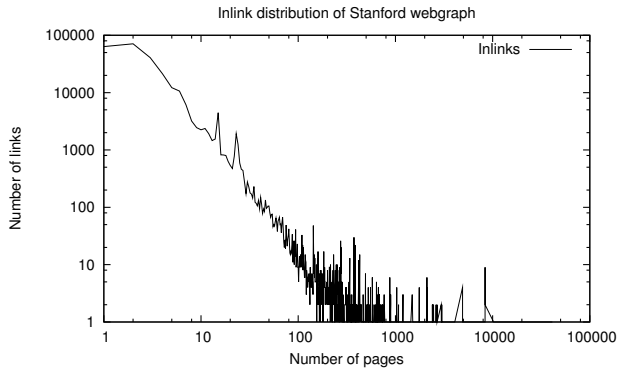

Figure 3.2.: Inlink distribution of Graph A

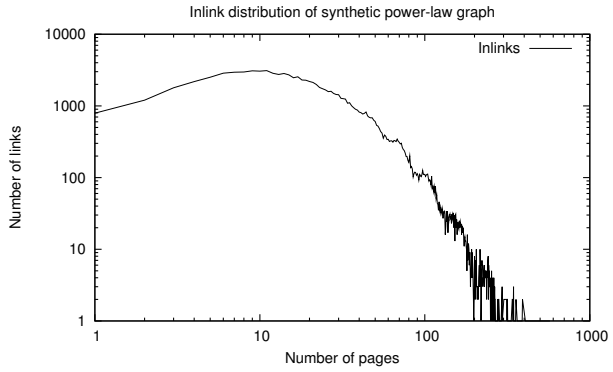

Figure 3.3.: Inlink distribution of Graph B

hubs-and-spokes model. Very few nodes have a very high inlink values, emphasizing our point that very few nodes require frequent global synchronization. More often than not, even these nodes (hubs) mostly have spokes as their neighbors.

Crawlers inherently induce locality in the graphs as they crawl neighborhoods before crawling remote sites. We partition graphs using Metis. A good partitioning algorithm that minimizes edge-cuts has the desired effect of reducing global synchronizations as well. Metis, though not optimized for power-law graphs, does a decent partitioning of the graph. This partitioning is performed off-line (only once) and takes about 5 seconds which is negligible compared to the runtime of *PageRank*, and hence is not included in the reported numbers.

Results

To demonstrate the dependence of performance on global synchronizations, we vary the number of iterations of the algorithm by altering the number of partitions the graph is split into. Fewer partitions result in a smaller number of large subgraphs. Each `map` task does more work and would normally result in fewer global iterations in the relaxed case. The fundamental observation here is that it takes fewer iterations to converge for a graph having already converged subgraphs. The trends are more pronounced when the graph follows the power-law distribution more closely. In either case, the total number of iterations are fewer than in the general case. For Eager *PageRank*, if the number of partitions is decreased to one, the entire graph is given to one *global map* and its local MapReduce would compute the final *PageRanks* of all the nodes. If the partition size is one, each partition gets a single adjacency list, Eager *PageRank* becomes regular *PageRank*, because each `map` task operates on a single node.

Figures 3.4 3.5 show the number of iterations taken by the relaxed and general implementations of *PageRank* on input graphs A and B that we use for input, as we vary the number of partitions. The number of iterations does not change in the general case, since each iteration performs the same work irrespective of the number of partitions and partition sizes.

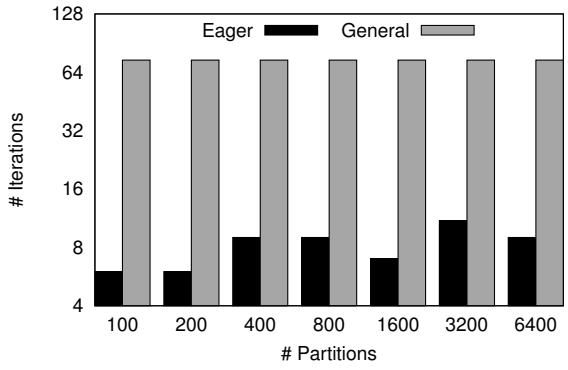

Figure 3.4.: *PageRank*: Number of iterations to converge (on y-axis) for different number of partitions (on x-axis) for Graph A

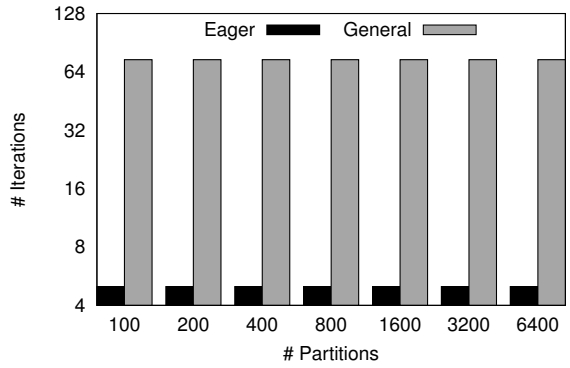

Figure 3.5.: *PageRank*: Number of iterations to converge (on y-axis) for different number of partitions (on x-axis) for Graph B

The results for Eager *PageRank* are consistent with our expectation. The number of global iterations is low for fewer partitions. However, it is not strictly monotonic since partitioning into different number of partitions results in varying number of inter-component edges.

The time to solution depends strongly on the number of iterations but is not completely determined by it. It is true that the global synchronization costs would decrease when we reduce the number of partitions significantly; however, the work to be done by each `map` task increases significantly. This increase potentially results in increased cost of computation, more so than the benefit of decreased communication. Hence, there exists an optimal number of partitions (not too small or not too big) for which we observe best performance.

Figures 3.6 and 3.7 show the run-times for the eager and general implementations of *PageRank* on graphs A and B with varying number of partitions. These figures highlight significant performance gains from the relaxed case over the general case for both graphs. On an average, we observe 8 X improvement in running times.

3.3.3 Shortest Path

Shortest Path algorithms are used to compute the shortest paths and distances between nodes in directed graphs. The graphs are often large and distributed (for example, networks

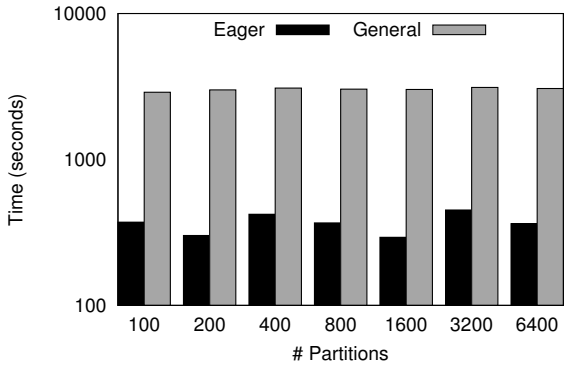

Figure 3.6.: *PageRank*: Time to converge (on y-axis) for various number of partitions (on x-axis) for Graph A

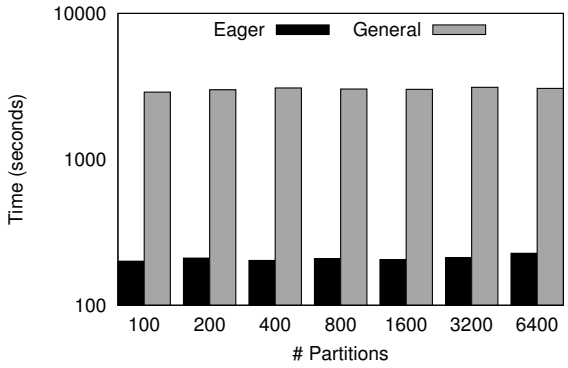

Figure 3.7.: *PageRank*: Time to converge (on y-axis) for various number of partitions (on x-axis) for Graph B

of financial transactions, citation graphs) and require computation of results in reasonable (interactive) times. For our evaluation, we consider *Single Source Shortest Path* algorithm in which we find the shortest distances to every node in the graph from a single source. *All-Pairs Shortest Path* has a related structure, and a similar approach can be used.

Distributed implementation of the commonly used Dijkstra's algorithm for *Single Source Shortest Path* allows asynchrony. The algorithm maintains the shortest known distance of each node in the graph from the source (initialized to zero for the source and infinity for the rest of the nodes). Shortest distances are updated for each node as and

when a new path to the node is discovered. After a few iterations, all paths to all nodes in the graph are discovered, and hence the shortest distances converge. Distributed implementations of the algorithm allow partitioning of the graph into subgraphs, and computing shortest distances of nodes using the paths within the subgraph asynchronously. Once all the paths in the subgraph are considered, a global synchronization is required to account for the edges across subgraphs.

Implementation

In the general implemenation of *Single Source Shortest Path* in MapReduce, each `map` operates on one node (would take its adjacency list as input); and for every destination node, emits the sum of the shortest distance to the node and the weight of the edge in consideration. This is the shortest distance to the destination node on a known path through the source. Each `reduce` phase operates on one node (receives weights of paths through multiple nodes as input); finds the minimum of the different paths to find the shortest path until that iteration. Convergence takes a number of iterations — the shortest distances of nodes from the source would not change. Again for the base case (like in *PageRank*), we take a partition as input instead of a single node's adjacency list, without any loss in performance.

In the eager implemenation of *Single Source Shortest Path*, each `map` takes a subgraph as input; and through iterations of *local map* and *local reduce* functions, computes the shortest distances of nodes in the subgraph from the source through other nodes in the same subgraph. A *global reduce* ensues upon convergence of all local MapReduce operations. Since most real-world maps are heavy-tailed, edges across partitions are rare and hence we expect a decrease in the number of global iterations, with bulk of the work performed in the local iterations.

Results

We evaluate *Single Source Shortest Path* on graph A used in the evaluation of *PageRank*. We assign random weights to the edges in the graphs.

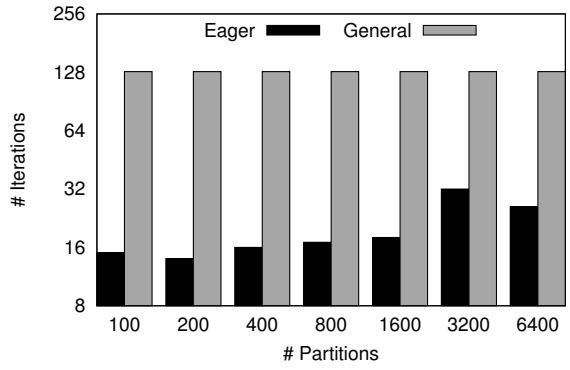

Figure 3.8.: *Single Source Shortest Path*: Number of iterations to converge (on y-axis) for different number of partitions (on x-axis) for Graph A

Figure 3.8 shows the number of global iterations (synchronizations) *Single Source Shortest Path* takes to converge for varying number of partitions in graph A. Clearly, the eager implemenation requires fewer global iterations for fewer partitions. Again, the iteration count is not strictly monotonic, due to differences in partitioning. The number of global iterations remains the same.

Figure 3.9 shows the convergence time for *Single Source Shortest Path* for varying number of partitions in graph A. As observed in *PageRank*, though the running time depends on the number of iterations, it is not entirely determined by it. As in the previous case, we observe significant performance improvements amounting to 8x speed-up over the general implementation.

3.3.4 K-Means

K-Means is a commonly-used technique for unsupervised clustering. Implementation of the algorithm in the MapReduce framework is straightforward as shown in [23, 52].

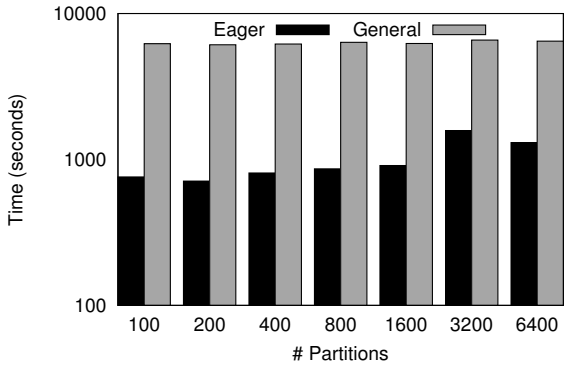

Figure 3.9.: *Single Source Shortest Path*: Time to converge (on y-axis) for various number of partitions (on x-axis) for Graph A

Briefly, in the `map` phase, every point chooses its closest cluster centroid and in the `reduce` phase, every centroid is updated to be the mean of all the points that chose the particular centroid. The iterations of `map` and `reduce` phases continue until the centroid movement is below a given threshold. Euclidean distance metric is usually used to calculate the centroid movement.

In Eager *K-Means*, each *global map* handles a unique subset of the input points. The *local map* and *reduce* iterations inside the *global map*, cluster the given subset of the points using the common input-cluster centroids. Once the local iterations converge, the *global map* emits the input-cluster centroids and their associated updated-centroids. The *global reduce* calculates the final-centroids, which is the mean of all updated-centroids corresponding to a single input-cluster-centroid. The final-centroids form the input-cluster centroids for the next iteration. These iterations continue until the input-cluster centroids converge.

The algorithm used in the relaxed approach to *K-Means* is similar to the one recently proposed by Tom-Yov and Slonim [53] for pairwise clustering. An important observation from their results is that the input to the *global map* should not be the same subset of the input points in every iteration. Every few iterations, the input points need to be partitioned differently across *global maps* so as to avoid the algorithm's move towards local optima.

Also, the convergence condition includes detection of oscillations along with the Euclidean metric.

We use the *K-Means* implementation in the normal MapReduce framework from the Apache Mahout project [54]. Sampled US Census data of 1990 from the UCI Machine Learning repository [55] is used as the clustering data for comparison between the general and eager approaches. The sample size is around 200K points each with 68 dimensions. For both General and Eager *K-Means*, initial centroids are chosen at random for the sake of generality. Algorithms such as canopy clustering can be used to identify initial centroids for faster execution and better quality of final clusters.

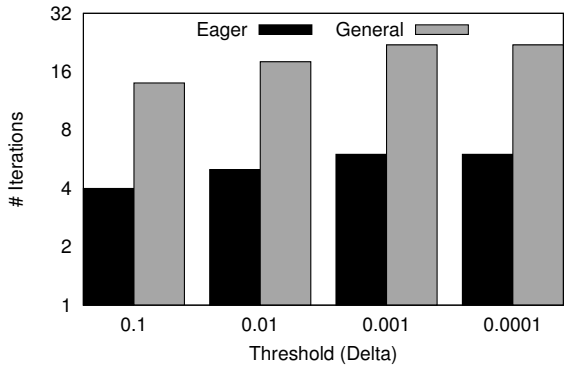

Figure 3.10.: Iterations-to-converge for varying thresholds

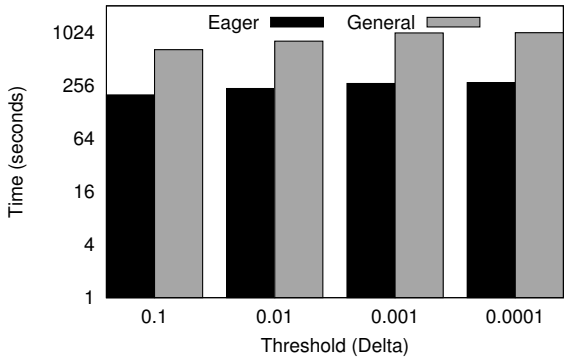

Figure 3.11.: Time-to-converge for varying thresholds

Figure 3.10 shows the number of iterations required to converge for different thresholds of convergence, with a fixed number of partitions (52). It is evident that it takes more iterations to converge for smaller threshold values. However, Eager *K-Means* converges in less than one-third of the global iterations taken by general *K-Means*. Figure 3.11 shows the time taken to converge for different thresholds. As expected, the time to converge is proportional to the number of iterations. It takes longer to converge for smaller threshold values. Partial synchronizations lead to a performance improvement of about 3.5 x on average compared to general *K-Means*.

3.3.5 Broader Applicability

While we present results for only three applications, our approach is applicable to a broad set of applications that admit asynchronous algorithms. These applications include — *all-pairs Shortest Path*, network flow and coding, neural-nets, linear and non-linear solvers, and constraint matching.

3.4 Discussion

We now discuss some important aspects of our results — primarily, (i) does our proposed approach generalize beyond small classes of applications? (ii) what impact does it have on the overall programmability? and (iii) how does it interact with other aspects, such as fault tolerance and scalability, of the underlying system?

Generality of Proposed Extensions Our partial synchronization techniques can be generalized to broad classes of applications. *PageRank*, which relies on an asynchronous mat-vec, is representative of eigenvalue solvers (computing eigenvectors using the power method of repeated multiplications by a unitary matrix). Asynchronous mat-vecs form the core of iterative linear system solvers. *Shortest Path* represents a class of applications over sparse graphs that includes Minimum Spanning Trees, Transitive Closure, and Connected Components. Graph alignment through random-walks and isoranks can be directly cast

into our framework. A wide range of applications that rely on the spectra of a graph can be computed using this algorithmic template. Our methods direcly apply to neural-nets, network flow, and coding problems, *etc.* Asynchronous *K-Means* clustering immediately validates utility of our approach in various clustering and data-mining applications. The goal of this chapter is to examine tradeoffs of serial operation counts and distributed performance. These tradeoffs manifest themselves in wide application classes.

Programming Complexity While allowing partial synchronizations and relaxed global synchronizations requires slightly more programming effort than traditional MapReduce, we argue that the programming complexity is not substantial. This is manifested in the simplicity of the semantics used in the chapter to describe it. Our implementations of the benchmark problems did not require modifications of over tens of lines of MapReduce code.

Other Optimizations Few optimizations have been proposed for MapReduce for specific cases. Partial synchronization techniques do not interfere with these optimizations. *eg.,* Combiners are used to aggregate intermediate data corresponding to one key on a node so as to reduce the network traffic. Though it might seem our approach might interfere with the use of combiners, combiners are applied to the output of *global map* operations, and hence *local reduce* (part of the `map`) has no bearing on it.

Fault-tolerance While our approach relies on existing MapReduce mechanisms for fault-tolerance, in the event of failure(s), our recovery times may be slightly longer, since each `map` task is coarser and re-execution would take longer. However, all of our results are reported on a production cloud environment, with real-life transient failures. This leads us to believe that the overhead is not significant.

Scalability In general, it is difficult to estimate the resources available to, and used by a program execution in a cloud. We draw our assertions of scalability of our formulation by infering resource availability from the number of graph partitions. Please note that these

inferences are meant to be qualitative in nature, and not based on raw data (since the data is not made available by design).

In order to get a quantitative understanding of our scalability, we ran a few experiments on the 460-node cluster provided by the IBM-Google consortium as part of the CluE NSF program. Typically, clusters run of the order of 10 `map` tasks per node. Since each `map` handles one complete partition of the graph, for very large numbers of partitions(*eg.,* 6400), we potentially use the entire 460 nodes in the Google cluster for the `map` phase. Such high node utilization incurs heavy network delays during copying and merging before the `reduce` phase, leading to increased synchronization overheads. By showing significant performance improvements even in a setting of such large scale, our approach demonstrates scalability.

3.5 Future Work

The myriad trade-offs associated with diverse overheads on different platforms pose intriguing challenges. We identify some of these challenges as they relate to our proposed solutions:

Generality of semantic extensions. We have demonstrated the use of partial synchronization and eager scheduling in the context of few applications. While we have argued in favor of their broader applicability, these claims must be quantitatively established. Currently, partial synchronization is restricted to a `map` and the granularity is determined by the input to the `map`. Taking the configuration of the system into account, one may support a hierarchy of synchronizations. Furthermore, several task-parallel applications with complex interactions are not naturally suited to traditional MapReduce formulations. Do the proposed set of semantic extensions apply to such applications?

Optimal granularity for maps. As shown in our work, as well as the results of others, the performance of a MapReduce program is a sensitive function of map granularity. An automated technique, based on execution traces and sampling [56] can potentially deliver

these performance increments without burdening the programmer with locality enhancing aggregations.

System-level enhancements. Often times, when executing iterative MapReduce programs, the output of one iteration is needed in the next iteration. Currently, the output from a reduction is written to the (distributed) file system (DFS) and must be accessed from the DFS by the next set of maps. This involves significant overhead. Using online data structures (for example, Bigtable) provides credible alternatives, however, issues of fault tolerance must be resolved.

Implications for tightly coupled systems. MapReduce has been shown to be an effective alternative to conventional threading APIs even on shared memory systems [23] for specific applications. It is important to note that shared-memory MapReduce has a much larger design space because of higher control over the spawned map and reduce tasks (thread pools). A number of performance optimizations are possible here – ranging from pipelining iterates of map and reduce operations, to reordering map and reduce operations in order to aggregate maps and enhance computation-communication characteristics. Other optimizations involve speculative execution of maps, relying on promises as outputs of maps as opposed to the values themselves. This rich space of optimizations bears significant research investigation.

.

3.6 Chapter Summary

In this chapter, we motivate MapReduce as a platform for distributed execution of asynchronous algorithms. We propose partial synchronization techniques to alleviate global synchronization overheads. We demonstrate that when combined with locality enhancing techniques and algorithmic asynchrony, these extensions are capable of yielding significant performance improvements. We demonstrate our results in the context of the problem of computing *PageRanks* on a web graph, find the *Shortest Path* to any node from a source,

and *K-Means* clustering on US census data. Our results strongly motivate the use of partial synchronizations for broad application classes. Finally, these enhancements in performance do not adversely impact the programmability and fault-tolerance features of the underlying MapReduce framework.

4 TRANS-MR: DATA-CENTRIC PROGRAMMING BEYOND DATA PARALLELISM

Data-centric programming models like MapReduce [1] and Dryad [29] have received considerable attention over the past few years. The success of these models can be attributed to the simplicity of the underlying programming models, support for fault tolerance, and scalable performance. MapReduce adopts deterministic replay for fault-tolerance — compute elements that fail are simply re-executed. In the absence of side-effects, re-executed compute elements produce the same outputs, thus providing clearly specified semantics.

Fault-tolerance through deterministic replay, however, does not work in the presence of side-effects (e.g., writes to persistent storage or communication over the network) or non-deterministic operations (e.g., using a random number generator). Consider a map function writing to the underlying distributed file system. If this instance is replayed (in case of a fault), the re-execution is oblivious of the previous write and hence rewrites the data. Both of these writes are, however, visible to external processes leading to non-deterministic behavior. For this reason, side-effects are not well-supported within the MapReduce framework.

The application scope of MapReduce, and related models can be extended significantly by allowing communication/ data-sharing across computations. Data-sharing through side-effects on shared address space (e.g., a shared disk-resident key-value store) enables speculation and task-parallelism in applications. Consider an illustrative example of finding the minimal spanning tree of a large graph using Boruvka's algorithm. Each iteration (operating on distinct nodes) coalesces a node and its closest neighbor. Iterations in which node-coalescing does not cause conflicts, can be executed in parallel. However, these conflicts can be detected only at runtime, since it depends on the input graph. This form of parallelism is known as speculative-parallelism or amorphous data-parallelism [58]. Exploiting this form of parallelism requires communication across computations to detect

and resolve potential conflicts. Further, communication through mutable shared-data helps develop scalable online and streaming applications, such as online aggregation, which need immediate change-propagation.

Towards this goal, we propose effective mechanisms for supporting side-effects over a shared address space. As a model, we use a distributed key-value store (Bigtable [11]) as the underlying storage for MapReduce — the input, output, and side-effects are stored in this fault-tolerant key-value store. Bridging the disparate fault-tolerance mechanisms adopted by the storage (persistence through replication) and computation (deterministic replay) layers presents significant technical challenges relating to definition of semantics, efficient implementations, and application integration.

In this chapter, we propose semantics for transactional execution of computations (map/reduce functions) over distributed key value stores, using primitives adapted from Software Transactional Memory (STM) literature. By restricting side-effects only to the key-value store, we derive effective mechanisms for avoiding the consistency problems associated with deterministic replay. In our model, results of one computation (writes to the global key-value store) become atomically visible to other computations, and to other concurrent jobs, upon successful completion of the computation. Though we discuss our semantics in the context of MapReduce and HBase, our proposed semantics apply more generally to all data-centric models over shared address spaces. We support our claims of performance and enhanced application scope in the context of diverse speculative-parallel applications such as Boruvka's minimum spanning tree algorithm and maximum flow calculation using Push-Relabel algorithm. Note that these algorithms cannot be expressed in the current MapReduce framework.

4.1 TransMR Programming Model

The `TransMR` (Transactional MapReduce) programming model defines the semantics for transactional execution of computations over shared address spaces. The system architecture, shown in Figure 4.1, describes interactions between various components. The

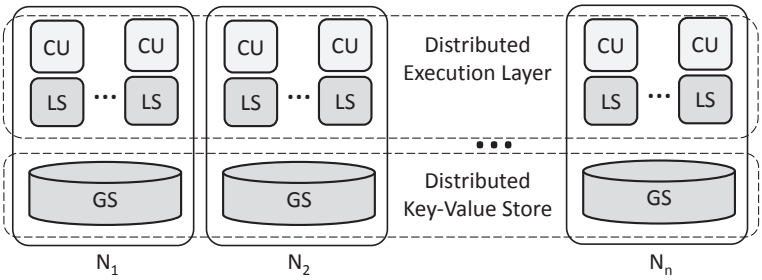

Figure 4.1.: System architecture

computation and storage layers span a cluster of nodes. We propose the use of distributed key-value store for the shared global store (GS). The contents of the global store (GS) are accessible to all computation units (CU), albeit through a private local store (LS). Reads/writes from within a CU are served from/to its local store (write buffer). If the local store does not have the data corresponding to a `read`, it fetches the (key, value) pair from the global store. All writes are buffered in LS.

Upon execution of a computation unit CU_i, its write buffer (present in its local store) is validated against the global store for any concurrent accesses of the same data by other CUs. In the absence of such conflicts, the buffered writes are safely written to the global store. However, in case of conflicts, the computation unit (CU_i) is re-executed. Software Transactional memory (STM) systems achieve this behavior by defining transactional execution scope through TM_{BEGIN} and TM_{END} statements. In the `TransMR` model, each map/reduce function is treated as a computation unit, resulting in their transactional execution. The model supports serializability as the consistency guarantee during validation and commit of conflicting CU transactions.

4.1.1 Semantics

Figure 4.2 describes the syntax of our proposed model. Each computation unit has a private local store(Σ) in addition to the shared global store(Γ). We define lookup functions (mapping keys to values), σ and γ, for local and global store lookups, respectively. For

$$
\begin{aligned}
LocalStore &:= \{\Sigma_1, ..., \Sigma_m\} && (4.1)\\
GlobalStore &:= \{\Gamma\} && (4.2)\\
\sigma \in \Sigma &= L \rightarrow Z && (4.3)\\
\gamma \in \Gamma &= L \rightarrow Z && (4.4)\\
Fn &:= \{f_m, f_r\} && (4.5)\\
f \in Fn &:= Atomic\{Op^*\} && (4.6)\\
Op &:= Get\ \ k | Put\ (k, v) | Other && (4.7)\\
b \in Boolean &:= \{True, False\} && (4.8)\\
k, v \in Values &:= \{b, UnObservable\} && (4.9)\\
l &:= [v_1, ..., v_n] && (4.10)
\end{aligned}
$$

Figure 4.2.: Transactional MapReduce: Syntax

each computation unit, σ is empty to begin with; subsequent reads/ writes add mappings. A computation unit is defined as a sequence of operations — read/ write from/ to the store (Get /Put) or a thread local operation (Other) with no side-effects.

The operational semantics, shown in Figure 4.3, capture the behavior of the model. The semantics use **map**, **fold**, **if-then-else** constructs, which carry their usual functional definitions. A Transactional MapReduce job (**TMR**) takes an input list, along with map/reduce functions. The job involves applying the computation units (map/reduce functions) on appropriate elements in the input list *atomically*. The possible constituent operations (Get, Put, and Other) are executed in the context of both local and global stores. A `Put`(k, v) operation modifies the local store adding the new key-value pair, (k, v), to the map. A `Get`(k) operation first copies the value from global store to local store if it does not already exist ($k \notin$ *domain*(σ)), and subsequently returns the value. Upon successful completion of all operations in the computation, the local store is copied to the global store atomically through a *two-phase commit protocol*. The functional definitions of **map** and **fold** capture

$$l, \sigma \implies \sigma(l) \qquad \text{(LOCAL)}$$

$$l, \gamma \implies \gamma(l) \qquad \text{(GLOBAL)}$$

$$\frac{\textbf{map}\ f_m\ \bar{l}, \gamma \implies \bar{l}'', \gamma'' \qquad \textbf{fold}\ f_r\ \bar{l}'', \gamma'' \implies \bar{l}', \gamma'}{\textbf{TMR}\ f_m\ f_r\ \bar{l}, \gamma \implies \bar{l}', \gamma'} \qquad \text{(TMR)}$$

$$\frac{\begin{array}{c}\textbf{if}\ (k \notin \mathit{domain}(\sigma))\ \textbf{then}\ \sigma' = \sigma[k \mapsto \gamma(k)] \\ \textbf{else}\ \sigma' = \sigma \\ k, \sigma' \implies v\end{array}}{\mathit{Get}\ k, \sigma, \gamma \implies v, \sigma', \gamma} \qquad \text{(GET)}$$

$$\frac{\sigma' = \sigma[k \mapsto v]}{\mathit{Put}\ (k, v), \sigma, \gamma \implies \mathit{True}, \sigma', \gamma} \qquad \text{(PUT)}$$

$$\frac{}{\mathit{Other}, \sigma, \gamma \implies \mathit{UnObservable}, \sigma, \gamma} \qquad \text{(OTHER)}$$

$$\frac{\begin{array}{c}\mathit{Op}_1, \sigma, \gamma \implies v_1, \sigma'_1, \gamma \\ \mathit{Op}_2, \sigma_1, \gamma \implies v_2, \sigma'_2, \gamma \\ \ldots \\ \mathit{Op}_n, \sigma_{n-1}, \gamma \implies v_n, \sigma'_n, \gamma \\ \forall k_i \in \mathit{domain}(\sigma) \qquad m = |\sigma|, \\ \gamma' = \gamma[k_1 \mapsto \sigma(k_1), ..., k_i \mapsto \sigma(k_i), ...k_m \mapsto \sigma(k_m)]\end{array}}{\mathit{Atomic}(\ \mathit{Op}_1,\ \mathit{Op}_2, ...,\ \mathit{Op}_n), \gamma \implies v_n, \gamma'} \qquad \text{(FN)}$$

Figure 4.3.: Transactional MapReduce: Operational semantics

the serialized application of functions to list items. In practice, validation protocols are used to achieve this serialization. Our implementation for MapReduce over Bigtable uses optimistic concurrency control to guarantee serializability among concurrent transactional executions of computation units.

4.2 Design of TransMR Framework

The transactional semantics mentioned above are general enough to be realized using conventional MapReduce-based execution environments (Hadoop [49], Dryad, Pig, *etc.*) operating on typical key-value stores (HBase [59]), Cassandra, *etc.*). This section discusses various design considerations and our implementation of the proposed programming model. The `TransMR` framework uses Hadoop and HBase as the execution and storage engines, respectively. The framework treats map/reduce functions as computation units (CU) executing over the global store, HBase. The map and reduce functions are executed transactionally and upon successful completion, their outputs are stored in HBase. These outputs are visible to other map/ reduce computations of the same MapReduce job, and also to other jobs. As long as the key of a map function output forms the key of HBase table, all values with the same key are versioned using timestamps and stored together. They are implicitly sorted using insertion sort. Thus, a reduce function can directly read its input from HBase, through a scan of all the versioned values for any particular key, avoiding the expensive shuffle phase in Hadoop.

4.2.1 Concurrency Control

The validation-and-commit phase of the CU transaction uses optimistic concurrency control [60]. At the start of its validate-and-commit phase, a transaction increments atomic counters on those GS nodes hosting keys involved in that transaction. The read/write sets of a transaction T_i , are validated against the sets of those transactions that committed their writes, between the start of T_i and the time it increased the atomic counter; the start of T_i is noted by saving the state of the atomic counters at the beginning of the transaction.

In our implementation, the choice of optimistic concurrency control (optimistic reads and write-buffering), as opposed to pessimistic locking, must be noted. This choice is motivated by the nature of clients in data-centric models. Typically, a client can execute at any node (potentially, the slowest) in a heterogeneous distributed environment. Furthermore, the duration of a transaction may be potentially long. In such a scenario, pessimistic lock-

ing of rows prevents parallel execution of other transactions with data dependencies. In case of crash failures, these transactions must wait for the system to release all the locks held by the failed transaction. Thus, in fault-prone environments, pessimistic locking could impact the performance significantly. In optimistic concurrency control, reads do not require locks (eager reads) and writes are buffered (lazy writes). During commits, only those concurrent transactions that have conflicts are considered [61]. As no locks are acquired, the possibility of a deadlock is avoided.

4.2.2 Fault Tolerance Model and its Implications on CAP

The client (the process executing the computation units) may be fault-prone and also fault-tolerant in itself. This fault-prone nature is directly implied by the general characteristics of MapReduce based execution environments — run on commodity clusters or virtual machines in the cloud and susceptible to hardware/software faults. The client's fault-tolerant nature implies that, even if the client fails during its execution, its replay mechanism makes the client recover and process all its records. Since the availability of the client is itself in question, expecting high availability from the storage servers is unreasonable. Further, MapReduce based applications demand strict consistency of data to ensure correctness of the algorithm's execution. The above two considerations motivate our choice of Consistency(C) over Availability(A), while accounting for Network Partitions(P) inside a datacenter, during execution of distributed transactions (*i.e.,* choosing C and P in the CAP Theorem [62]). For the duration of the network partition, the system does not allow affected distributed transactions to succeed. Thus, by weakening availability, the framework assures strict consistency of data. In the wake of crash failures, the leases held by compute nodes and storage nodes time-out, leading to replay-based recovery measures for compute nodes and replica-based recovery for storage nodes.

4.2.3 Prototype Implementation

The prototype was implemented by modifying and integrating various parts of Hadoop and transactional HBase. It primarily involved integrating their disparate fault-tolerance mechanisms during execution followed by validation of the CU transaction. Consider the following known corner case in the two-phase-commit protocol: a participant crashes after sending its own vote, but before receiving the commit/abort decision from the coordinator. Upon recovery, the participant faces ambiguity over committing the logged read/write sets of the successful validation phase. In our prototype, to resolve this ambiguity, the transaction manager(coordinator) writes its decision to Abort or Commit in the *Global CU log* (a table in HBase), before sending the decision to the nodes involved in the commit. The entry in the *Global CU log* can be identified by a unique *transaction-id*. The recovering storage node uses this id to look up the final decision and complete the write-ahead log, which is later used to regain the consistent state of the failed node. Further, the entry in the *Global CU log* is duplicated to be accessible by using the unique *computation-unit-id*; this helps Hadoop verify the successful execution of a map/reduce function and thereby avert re-execution of it, due to faults or speculative execution. Thus, the *Global CU log* forms a key element in dealing with arbitrary failures of computation and storage, by integrating their disparate fault-tolerance mechanisms.

4.3 Evaluation

The `TransMR` programming model allows speculative parallel execution of tasks with potential data dependencies. We demonstrate results on two such applications in this section — Boruvka's minimum spanning tree algorithm, and Preflow Push-Relabel maximum flow computations. We run our experiments on 16 Amazon EC2[1] extra large instances (`c1.xlarge`; each instance has 8 cores and 7 GB RAM).

[1] Amazon EC2. http://aws.amazon.com/ec2/

4.3.1 Boruvka's Minimum Spanning Tree (MST)

The sequential version of Boruvka's MST algorithm iterates over nodes in the graph. Each iteration — operating on a node (u) — involves finding the node v closest to u's component, adding the edge between these two nodes to the minimal spanning tree, and coalescing u and v. The process is initiated with as many components as nodes (each node forming a component); every iteration coalesces two components. The resulting component gives the minimal spanning tree of the input graph.

Parallelizing these iterations involves detecting runtime conflicts in case two distinct nodes (u_1, u_2) attempt to coalesce the same node v. Such a formulation is infeasible in traditional MapReduce. In the the `TransMR` formulation of Borvuka's algorithm, we store the input graph as well as coalescing information, as different column families in HBase. Each row corresponds to one graph node, the adjacency list, and the *node-id* of its parent in the component tree. Each map function, with a single row being its input, parses the adjacency list of a node u, and the adjacency lists of other nodes in its component (obtained by traversing its component tree) to find the closest node v. It then coalesces u's component tree with v's component tree by making one the parent of the other. The algorithm does not need a reduce phase. Instantiations of the same map function on different nodes in the graph might conflict when they both try to coalesce the same component; in this case, the consistency guarantee — serializability among conflicting instantiations — provided by the runtime, is necessary and sufficient for the correctness of algorithm's execution. From a programmer's perspective, the algorithm fits within the regular MapReduce programming model, except that the system needs to handle runtime data-dependencies; the `TransMR` programming model provides this guarantee to the programmer.

For evaluation, we run Boruvka's algorithm on a 100 thousand node graph with an average degree of 50 generated using the forest fire model of iGraph[2]. The sequential implementation is the same program run on a single node without any speculative parallelism (all maps executed sequentially). Figure 4.4 plots the average execution time and the num-

[2]iGraph. http://igraph.sourceforge.net

ber of aborts due to conflicts against the number of machines used. Due to the large number of vertices, the average number of conflicts detected amount to less than 0.5 percent of total executions. We observe upto 3.73 times speedup on 16 nodes. In the initial stages of the algorithm, almost half of the nodes can coalesce with their nearest neighbors without conflicts, leading to abundant parallelism. The available parallelism reduces significantly as the computation progresses. Considering the algorithm's inherent sequential nature due to dependencies, the observed performance gains are significant.

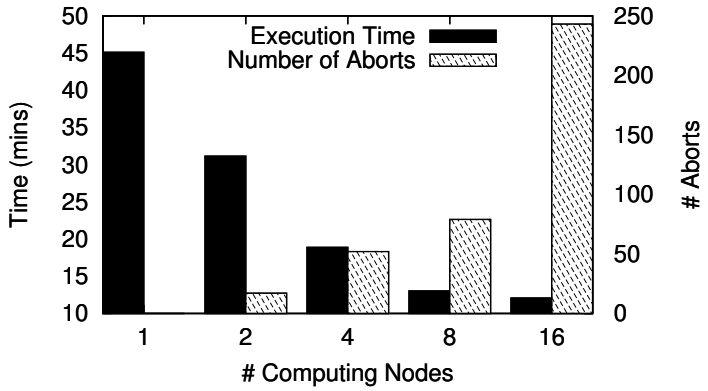

Figure 4.4.: Application performance: Boruvka's MST

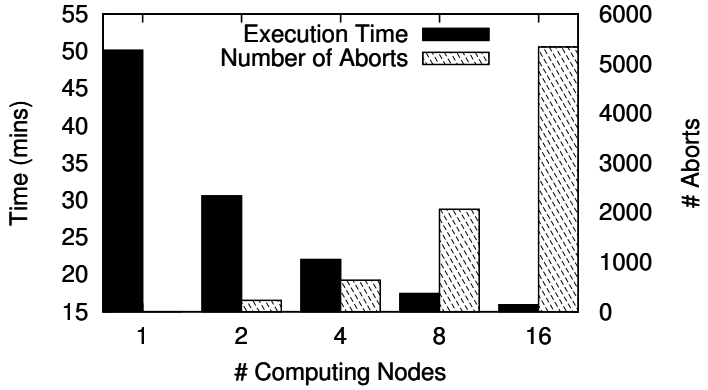

Figure 4.5.: Application performance: Preflow push-relabel

4.3.2 Preflow Push-Relabel

The Preflow Push-Relabel algorithm computes the maximum flow possible through a flow network. The algorithm maintains a preflow — a flow function with the possibility of excess at the vertices — terminating when there is no positive excess. The *Push* operation increases the flow on a residual edge, and a height function on the vertices identifies the residual edges that can be pushed. When there are no more *Push* operations to be executed, a *Relabel* operation increases the height of the vertices, which have excess preflows. This sequence of operations continues until there are no more excesses on any of the vertices other than the source. It is evident that the same operation *Push* or *Relabel* cannot be applied to neighboring nodes concurrently. Conflicting executions must be detected at run-time, and hence traditional MapReduce cannot exploit this parallelism. A trivial concurrent implementation is to lock the entire neighborhood of a node before operating on it. This involves significant serialization overhead. An alternate approach is to speculatively execute the operations on all the nodes; detect and resolve conflicts at runtime by serializing their execution. The latter approach is adopted by the `TransMR` programming model.

In the `TransMR` formulation, each map function operates on one node whose adjacency list is stored as one row in an HBase table. Depending on the neighborhood constraints, the function executes a *Push* or a *Relabel* operation on the node. A *Relabel* operation simply involves increasing the height of the node, if it is less than all the neighboring nodes. In a *Push* operation, a residual edge is chosen from the node's adjacency list and its capacity is updated. Data corresponding to the other vertex connected by the edge, its values of excess, and residual capacity are updated, and both of the updated nodes (rows) are atomically committed. During the transactional commit, concurrent map-transactions are checked for reads or writes to the two rows being updated. If a conflict is detected, the transaction which is later in the commit-pending-queue is aborted and the corresponding map function is re-executed from the beginning; this ensures serializability. The programmer merely specifies concurrent transactions (maps) and not consider conflict detection or resolution,

thus adding *no additional complexity* to programs. The job is iteratively executed until there are no feasible *Push* or *Relabel* operations.

The input flow network is generated using the Washington network generator[3]. The network is a 1000 x 1000 grid with the source connected to all the nodes in the first column and the sink connected to all the nodes in the last column. Every node in a column randomly connects to three other nodes in the next column. The edge weights are randomly generated. The sequential implementation is the same program run on a single node without any speculative parallelism — a single map task executing all the map functions sequentially. Note that the algorithm can only be executed sequentially in the regular MapReduce setup, without any transactional support. Figure 4.5 shows the average times and associated aborts over a window of 40 iterations of *Push* or *Relabel* operations on the feasible nodes. On the 16 node cluster, the number of aborts (re-executions) amount to about 4% of the total executions. Further, we observe 4.5x speedup on 16 nodes. As before, speculative execution enables a meaningful performance gain, as compared to the baseline case where no parallelism could be exploited.

4.4 Discussion

4.4.1 Applicability

While our evaluation describes only two applications, the `TransMR` framework is applicable to *all* applications exhibiting speculative-parallelism [58]. Furthermore, applications suited to transactional memory systems (concurrent threads modifying a shared data-structure) and pipelined workflows, such as those present in the STAMP benchmark suite [63], can be easily formulated in the `TransMR` programming model. The model also suits producer-consumer based online applications needing immediate access to mutable shared data. The model trivially allows regular data-parallel applications; however, the involved setup costs for transactional support might lead to minor overheads. By implicitly executing each computation transactionally instead of explicit scope definitions (*begin,*

[3]Washington max flow network generator. http://www.avglab.com/andrew/CATS/maxflow/synthetic.htm

end statements), TransMR model offers increased applicability without increasing the programming complexity. As in any data-centric programming model, the programmer only needs to specify the operation on the specific data-element without being concerned about its runtime interaction with other operations.

4.4.2 Performance Improvement

Distributed transactions constitute the primary overhead in the TransMR model. It should be noted that the number of keys involved in a distributed transaction is typically small, because the read-write sets of computations (map/reduce functions), where the keys come from, are small. The performance of TransMR framework can be significantly improved by using locality-enhancing storage schemes, leading to localization of distributed transactions. To further mitigate the overhead of distributed transactions, application-specific optimizations such as relaxing consistency guarantees from serializability to snapshot isolation, as used in Percolator [18] or reducing the transaction scope to a subset of data-items, as used in Megastore [15], can be employed. While realizing these optimizations constitutes our future work, the primary goal of this chapter is to advocate the transactional programming model and its benefits.

4.5 Chapter Summary

In this chapter, we propose TransMR programming model to enable data-sharing in data-centric programming models for enhanced applicability. We define the semantics for transactional execution of MapReduce computations over shared address space. Through a prototype implementation of the proposed semantics, we demonstrate the applicability of the TransMR programming model in the context of applications exhibiting speculative parallelism.

5 M-LOCK: ACCELERATING DISTRIBUTED TRANSACTIONS ON KEY-VALUE STORES THROUGH DYNAMIC LOCK LOCALIZATION

Scale-out data stores, based on key-value abstractions are commonly used as backend or cloud storage for various applications. Systems like CloudDB [13, 14] and Megastore [15] were built to support OLTP applications on such stores. Megastore uses Bigtable [11] as the store, whereas CloudDB uses HBase [59]. Distributed graph databases like Trinity [64], HyperGraphDB [65] and Titan [66] also host large social network graphs in these stores.

Scale-out stores typically rely on the key-value abstraction of an *object*. Several approaches have been proposed to support ACID transactions on scale-out object stores. A common strategy is to partition the store into *disjoint* groups of objects (also called entity-groups [15] or micro-shards [14]) and provide efficient transactional support *only* on objects in a single group.

Multi-version concurrency control (MVCC) is typically used for transactions within the entity-group (local-transactions or LT) [15, 17]. Transactions on objects in multiple entity-groups (distributed transactions or DT) use distributed-locking and two-phase commit [17]. However, as with all distributed protocols that use locks, the overhead of acquiring locks dominates the transaction latency.

Locks are typically implemented by augmenting data objects with an "isLocked" field. Using the atomic read-modify-write operation that is natively supported by the key-value stores, a data object can be locked by changing its "isLocked" field value. A transaction acquires the lock on the object before updating it. Although this approach is simple and efficient for modifying a single object, it is not suitable for multi-object distributed transactions [17,18]. In a scale-out key-value store, data is routinely re-partitioned and re-assigned to other nodes to balance load across the cluster. Such movement of data scatters the locks on different nodes (lock dispersion), significantly increasing lock acquisition overhead. Moreover, re-partitioning of a set of data objects by the underlying key-value store is heav-

ily influenced by their total size, which includes the size of all previous versions of the objects, the size of their indices, and the size of their caches. By grouping data objects and their locks, unnecessary movement of locks is triggered even though these locks do not contribute significantly to the size of the data objects.

Furthermore, lock acquisition is sequential and synchronous (locks must be acquired one by one, in a pre-defined order, to prevent deadlocks), unlike data-updates and lock-releases, which can happen in parallel, and asynchronously.

To overcome the ill-effects of lock dispersion, we propose *M-Lock*, a novel *lock localization* service for distributed transaction protocols (e.g., the protocol used by the Google App Engine [17]) on scale-out data stores to decrease the average latency of transactions. Lock localization refers to *dynamic migration and partitioning* of locks across various nodes in the scale-out store to *reduce cross-partition* acquisition of locks. When locks are localized (after separating from their data objects), fewer network round-trips and persistent-writes are necessary to acquire locks.

Separating locks from associated data, however, affects the latency of local transactions operating on a single entity-group, since they may need to synchronize with the separated, remote locks. M-Lock balances this trade-off using online models that measure relative lock-usage.

This chapter makes the following specific contributions: (a) we propose *new protocols to migrate, release and repatriate* locks, (b) we describe *new policies* that control migration and repatriation of locks, (c) we leverage application-specific affinity among data objects to achieve an *application-specific localization* of locks, and (d) we present detailed experiments on a 30-node cluster for the TPC-C benchmark. Our experiments show that M-Lock improves the average transaction throughput by more than 25% on range-partitioned data, and by 6% to 14% even when we use application-knowledge to manually control data-placement.

5.1 Background

We briefly review a distributed transaction (DT) protocol that has been implemented on top of the Google App Engine (GAE) [17]. The protocol assumes that (a) the objects in the store are partitioned *a-priori* into entity-groups, and (b) there is a layer of software (local transaction or LT layer) that executes ACID transactions on objects in a single entity-group; for example, GAE uses MVCC for the LT layer.

The DT protocol uses write locks to control concurrent access to data-objects. A write lock is itself an object in the store. A lock-object controls access to one data-object, and is in the same entity-group as the data-object it protects. The DT protocol uses the LT layer for all reads or writes to application and lock-objects in an entity-group. We summarize the major steps, and highlight how the DT protocol leverages the LT layer [17].

1. *Reads:* Objects are read directly from the key-value store. These are the *initial* reads. Their keys form the *read-set*. The protocol records the version numbers for the read objects.

2. *Intercept Writes:* Object writes from the transaction are intercepted. Their keys form the *write-set*.

3. *Write to shadow objects:* Object writes are recorded in temporary objects, called the *shadow* objects, until it is safe to reflect the updates in the store. An object and its shadow are placed in the same group. Keys of the shadows form the *shadow-write-set*.

4. *Persist meta-data*: The read-set, the write-set and the shadow-write-set are written directly to the store as a transaction-specific object, called the *metadata* object. This information is used to roll forward the transaction in case of failure.

5. *Acquire locks:* Write locks are acquired for objects in the write-set, one at a time and in separate LTs, respecting the global ordering of the locks. Every lock is acquired in an LT because an atomic read-check-modify of the lock object is required to acquire

the lock. If a write lock cannot be acquired, then the protocol rolls forward the DT that is holding the lock, and the current transaction tries to acquire the lock again.

6. *Check validity of reads:* By now, all locks have been acquired. Using one LT per entity-group, reads are validated (by comparing the version numbers of the initial reads with the reads inside the LT). If the version information has changed for any object, then the transaction is aborted (the object is now stale).

7. *Commit:* Shadows are copied into their application objects, write locks are released, and the shadow-objects are deleted.

A special case is worth noting. If a transaction reads and writes objects in only one entity-group, then the entire transaction is attempted within a single LT. There is no need to write shadows, persist meta-data, or acquire locks. The LT checks for the absence of write locks on reads and writes. If write locks are found on objects being written, then the protocol terminates the LT and reverts to the usual distributed transaction (DT) protocol. Otherwise, the LT validates the reads, and then writes the objects. Due to this optimization, a transaction on a single entity group (*independent* LT) is faster than a DT.

5.2 Motivation

Latency of a distributed transaction depends on several factors. Most steps of the distributed protocol are scalable. For example, requests for multiple reads in the *Reads* step can be issued and executed in parallel by key-value stores. Similarly, we can parallelize the *Writes to shadows*, *Validation of reads* and the *Commit* steps. In contrast, the *Acquire locks* step can only acquire locks *one by one*, in accordance with a global ordering of locks. There are two factors that influence the latency of this step:

1) *Network round-trips:* If locks reside on different server nodes, then multiple network round-trips are necessary to acquire all the locks. A decrease in the network overhead due to lock acquisitions will decrease the overall latency of a distributed transaction, and improve throughput.

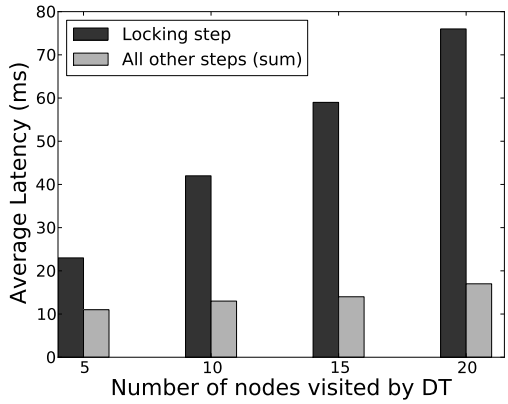

Figure 5.1.: Latency of the locking step versus other steps

2) *Persistent writes:* The DT protocol uses one LT to acquire a lock. Irrespective of the concurrency protocol that is used in the LT layer, we incur a persistent write for LT commit. The concurrency protocol logs all the object updates in the LT in a single Write-Ahead-Log(WAL) object, and the protocol persists this WAL-object. Updates logged in this WAL-object are later applied to the actual objects in the key-value store. Since the locks are acquired one by one, the LTs are also issued one by one, and persistent writes of the WAL-objects cannot be overlapped. Depending on the data replication strategy of the key-value store, a single persistent write can involve multiple network round-trips. For example, in HBase, if the replication factor is set as 3, then every persistent write is synchronously replicated onto 3 different machines. Our experiments show that the persistent write overhead during the acquisition of locks is comparable to, if not more than, the network overhead due to locks being scattered on different server nodes.

Figure 5.1 shows that the latency due to locking-related network round-trips and persistent writes is significantly higher than the latency of all other steps in the TPC-C NewOrder transaction [67].

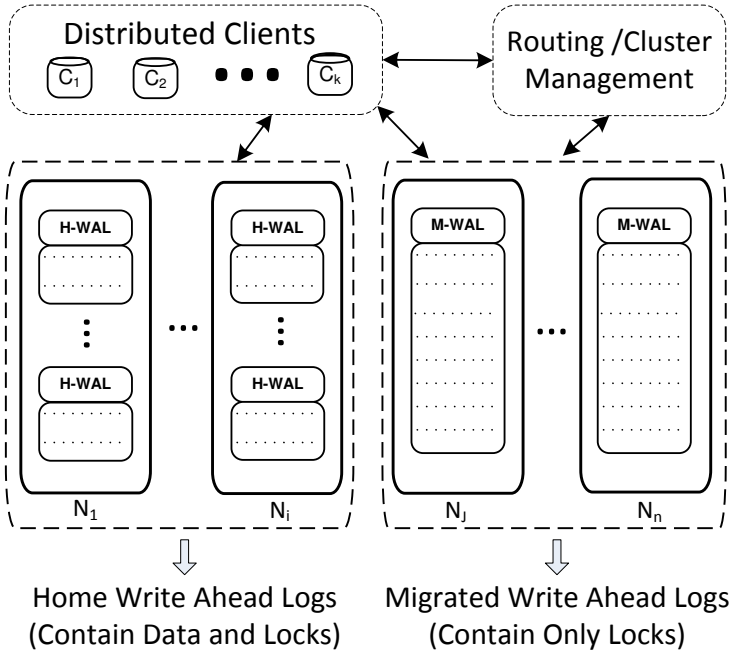

Figure 5.2.: Architecture of M-Lock

5.3 Overview of M-Lock

Figure 5.2 shows the architecture of M-Lock. Each entity-group has data-objects, lock-objects, and a Write-Ahead-Log (WAL) object that is used by the LT layer for atomic commit of data or lock-objects in that group. We refer to this WAL as the Home-WAL(H-WAL). Note that we use the term H-WAL to also refer to the entity-group that the H-WAL object belongs to. The M-Lock service opportunistically moves one or more lock-objects in an entity group to a new entity-group that only has lock-objects. The new group also has an object called the Migrated-WAL (M-WAL) that is used for atomic commit of the lock-objects in that group. Again, we use the term M-WAL to also refer to the new entity-group. Both H-WALs and M-WALs reside on the same store. M-Lock service isolates M-WALs from H-WALs, in a best-effort manner, by placing them on different nodes. Isolation reduces the interference between data and locks with respect to request load and memory usage. Furthermore, the M-Lock service attempts to place locks that frequently co-occur in

transactions in the same M-WAL. As described in Section 5.5, M-Lock also balances the trade-off between the latencies of distributed and independent local transactions.

5.4 Dynamic Lock Protocols

We describe several new protocols that ensure consistent movement of lock-objects between WALs. Policies that determine when and where to move the lock-objects are described in Section 5.5.

5.4.1 Lock Migration Protocol

Migration of a lock-object from a H-WAL to an M-WAL is non-trivial because it involves updates to objects in two different entity-groups. In essence, lock migration is itself a distributed transaction on two different WALs, and our migration protocol must ensure consistency of the transfer of locks. Note that lock migration in M-Lock is a best-effort endeavor: even if some locks cannot be migrated to the M-WAL, the DT will correctly acquire and release its locks. Given a H-WAL and a M-WAL, our lock migration protocol consists of the following steps:

1. If the lock has already migrated, then we inform the application about the new location of the lock, and abort the migration protocol. Also, if the lock is not free, we abort the migration protocol. Only if the lock is free and is in the H-WAL do we execute the next step.

2. We create a new lock object in the M-WAL, and mark the lock as available. However, no other transaction can access this lock until we record the key of the new lock-object in the H-WAL.

3. We record the new location of the lock in the H-WAL, only after checking that (a) the lock in the H-WAL is still free, and (b) the lock has not been migrated by another transaction. These checks are necessary because the lock may have been acquired or migrated by another transaction during the creation of the new lock-object in the

M-WAL. Note that these checks, as well as any possible writes involve only one WAL.

4. If either check fails in the previous step, we delete the newly created lock-object from the M-WAL, and abort the migration protocol. Creation and deletion of the newly created lock-object in the M-WAL cannot result in inconsistent operations (reads/writes) because transactions can access the M-WAL only through the H-WAL, and we never updated the H-WAL.

All reads and writes of the lock-objects in M-WALs or H-WALs are performed using the LT layer. The correctness of our protocol depends on being able to do the operations in Step 3, atomically: confirming that the lock-object is in the H-WAL and that it is free, and then placing the location (detour) of the migrated lock-object in the H-WAL. Since these operations only involve one H-WAL, they can happen atomically in an LT, and other transactions will see the lock-object either in an M-WAL or a H-WAL.

5.4.2 Lock Release Protocol

In the *commit* step of the distributed transaction protocol, release of migrated locks merits careful consideration. When there is no lock migration, an update to an object and the release of its lock happen atomically (in an LT) because the object and its lock are in the same H-WAL. However, if a lock has migrated, then the release of the lock happens at its M-WAL, while the write to the object happens at the H-WAL. Furthermore, the update of the object in the H-WAL must complete before the release of the lock. In essence, lock release is also a DT on two different WALs, and our release protocol must ensure consistency.

To accomplish inter-WAL causal operations, we augment a WAL with a persistent queue of *causal asynchronous updates*. Requests for update of objects under this WAL are written to the WAL as usual. Requests for updates of objects under other WALs are placed in the queue. First, updates to the objects under the WAL are committed. Sub-

sequently, a background thread asynchronously executes requests to update objects under other WALs.

Given a H-WAL and an M-WAL, our lock release protocol is as follows:

1. Update the object protected by the lock, and enqueue a request in the causal asynchronous update queue of the H-WAL to release the lock in the M-WAL.

2. Execute requests in the queue, asynchronously, using a background thread.

The asynchronous update queue ensures the consistency and causality of the following two operations – we update the object before we release the lock on the object.

5.4.3 Visiting Appropriate WALs When Validating Reads

In the read step, along with the data object, the client reads lock-object information at H-WAL. If the client finds migration information (M-WAL location) on the lock-object, then in the *validate reads* step, it visits the M-WAL directly to check for potential locks and stale reads. Note that commit function updates the transaction timestamp on both the lock objects (at H-WAL and M-WAL).

5.4.4 Lock Repatriation Protocol

This protocol will undo the migration of a lock. To ensure consistency, we must execute the two operations – deletion of the lock in the M-WAL and the reset of migration information in the H-WAL – as an atomic operation. In essence, lock repatriation is also a distributed transaction on two different WALs. Given a H-WAL and a M-WAL, repatriation consists of the following steps:

1. If the lock is in a released state, then delete the lock from the M-WAL. Otherwise, we abort repatriation.

2. Enqueue a request in the causal asynchronous update queue of the M-WAL (Migrated WAL). This request will reset the migration information for the lock in the H-WAL (Home WAL).

The asynchronous update queue ensures that we delete the lock from the M-WAL before we reset the migration information in the H-WAL. It is possible that a transaction observes the migration of a lock from the H-WAL and re-directs itself to the M-WAL only to find that the migrated lock-object in the M-WAL has been deleted. This is analogous to a cache-miss. In such cases, the DT protocol simply aborts and retries the entire transaction.

5.4.5 Performance Implications

Lock Migration If we add an explicit lock migration step to the distributed protocol, we increase the transaction latency. The challenge is to mask the latency of lock migration. Fortunately, locks are acquired on the objects in the write-set, which is known before the *Write to Shadows* and *Persist Metadata* steps of the distributed protocol. Therefore, if lock migration can be executed concurrently with these two steps, then the increase in transaction latency is mitigated. Migration of individual locks can happen in parallel, and the migration time is independent of the number of locks. Only the transaction that migrates the lock may see an increase in latency, while all subsequent transactions that use the lock will not incur the cost of lock migration.

Lock release After the update of an object, there is a delay in the release of its lock in the M-WAL. This delay does not add to the latency of the transaction. However, the asynchronous release of locks can affect concurrency.

Lock repatriation There can be a delay between the deletion of the lock in the M-WAL, and the subsequent reset of migration information in the H-WAL. Therefore, a transaction can read migration information in the H-WAL, but find that the lock does not exist in the M-WAL. We abort and restart such transactions. To control such aborts, we suppress

repatriation until the demand for the lock declines. We use a threshold time parameter (*trelease*) that configures the minimum time to wait after the last lock release.

Fault tolerance Fault-tolerance properties of the base DT protocol are similar to those described in [17]. For the additional protocols, the safety property relies on the fact that the locks reside either in M-WAL or H-WAL (asynchronous queues included), but not in any other state. The protocols also ensure consistent, decentralized routing information on both the lock objects (at H-WAL and M-WAL). This routing information is also cached at client-side to route subsequent transactions. In the event of a network partition, the DT protocol simply halts until the partitions reconnect. The DT protocol ensures strict consistency (C) and partition-tolerance (P) while forgoing availability (A), in the CAP theorem.

5.5 Dynamic clustering of locks

Lock localization is achieved when locks are clustered on M-WALs so that overheads of network round-trips and persistent writes are reduced for the entire set of transactions. One way to achieve this is to infer lock affinity by carefully examining the lock access patterns of transactions. A subset of locks that appear together in several transactions form a cluster, and multiple lock clusters may be evident in a given set of transactions. In our experiments, we use Schism [68] to generate clusters of locks. Schism models locks as nodes in a graph and a transaction as a clique on its corresponding locks. A graph partitioner [69] is used to partition the graph into multiple sub-partitions. Each sub-partition is given a numeric *lock cluster tag*. This process of collecting transaction logs and modelling access patterns runs offline on a separate cluster node. *Lock cluster tags* are stored in a specific system-table, in a compressed format. Clients read this information when they are first spawned and use it to determine the M-WAL on which a lock is to be migrated.

The aim of lock-localization is to place locks belonging to a lock cluster in a single M-WAL. Accordingly, M-Lock assumes that the application specifies a *function* that maps locks to lock clusters. M-Lock uses this *function* to assign locks to M-WALs. We note that our approach is similar to locality-sensitive-hashing [70], where similar data-points

(vectors in euclidean space) are hashed to common buckets. Schism generates this map from locks to lock cluster tags using graph partitioning. However, applications can use other static knowledge about the workload to generate the function. For example, frequent players of a game, or team members who collaboratively edit documents, can be grouped statically to create the map. Note that the mapping *function* could even be approximate. Since M-WALs only have lock-objects, their number is significantly less than H-WALs, thereby achieving lock-localization.

We map lock-objects in H-WALs to M-WALs, as follows. Given a lock-object l, we use the application-specific map function to determine the lock cluster l_c. We then construct the key for the migrated lock-object as the concatenation of the lock cluster(l_c), the lock (l) and the unique identifier (trx-id) of the transaction that is moving the lock: l_c@l@trx-id. We append the unique transaction identifier to avoid conflicts when two transactions attempt to concurrently migrate the same lock-object.

We note two interesting consequences of our key naming scheme. First, if the underlying key-value store is lexicographically range-partitioned, then most if not all of the migrated locks of a lock cluster are in the same M-WAL. If all the locks of a transaction are in a single M-WAL, then we incur the overhead of only one network round-trip, and one persistent write. Second, the global ordering of migrated locks (i.e., the lexicographical ordering of keys of the migrated lock-objects) is different from the global ordering of home locks (i.e., lexicographical ordering of the keys of the lock-objects in the H-WALs, which also happens to be the default global lock ordering for all the transactions). As a result, if a transaction acquires locks as per the global ordering of the home locks, then the transaction may have to visit an M-WAL multiple times to acquire locks in the M-WAL. For example, consider a transaction that acquires three locks $l1, l2$ and $l3$, in order. Assume that $l1$ and $l3$ belong to the same lock cluster, and that these locks have been migrated to the same M-WAL. Assume that the lock $l2$ has been migrated to a different M-WAL. During lock acquisition, we first acquire $l1$ from its M-WAL. Then, we acquire $l2$ from a different M-WAL, and we re-visit the M-WAL that had $l1$ to acquire $l3$!

Re-visiting an M-WAL implies additional network round-trips and persistent writes. To avoid re-visiting an M-WAL, we acquire migrated locks in order, and we acquire home locks in order. Note that it does not matter whether we acquire migrated locks before the home locks or vice-versa, since both approaches will result in a global, total order of migrated and home locks. In our experiments, we acquire migrated locks before home locks. In our example, we acquire locks $l1$ and $l3$ by visiting their M-WAL once, and then acquire $l2$ from a different M-WAL. Note that a transaction simply aborts if it does not find a lock at the WAL it was routed to. This ensures that all clients observe a consistent global order of locks in H-WALs and M-WALs, which prevents deadlocks.

5.5.1 Design Considerations for M-WAL

In our design, we have one M-WAL per node. In the context of HBase, an M-WAL corresponds to a region. The actual number of regions (M-WALs) needed to host all migrated locks is dynamically determined by the load-balancer. If the load-balancer detects high average frequency or queueing delay of lock requests for current region placement, it splits a region (M-WAL) into two and moves the splits to different nodes. To maintain clustering, region-splitter should split a region at the boundary of a lock-cluster. An M-WAL has locks from multiple H-WALs. Consequently, each M-WAL has a large number of lock-objects and could potentially receive a large number of concurrent requests for locks.

To efficiently handle these requests, our LT layer for M-WALs is based on *fine-grained object latches*. This is similar to the use of latches in single-node relational databases. Furthermore, M-WALs are designed assuming that the concurrency control follows the simple deadlock-prevention mechanism based on ordered locking.

On the other hand, the LT layer for H-WALs uses multi-version concurrency control (MVCC) [15]. Such a simple lock-free mechanism is sufficient for object requests without heavy contention.

5.5.2 Balancing Latency Trade-off Between DT and LT

We refer to clustering of lock-objects into H-WALs as the *home-clustering*, and clustering of migrated lock-objects into M-WALs as the *migrated-clustering*. Obviously, transactions that are limited to a single H-WAL (referred to as independent LTs, to differentiate them from the LTs that are part of a distributed transaction), benefit from the home-clustering of locks.

Before separation of locks from data, independent LTs commit data in H-WAL by checking for the absence of write-locks on the write-set and ensuring that reads are not stale. This execution involves only H-WAL. However, after lock separation, independent LTs become full-fledged distributed transactions as they need to synchronize between H-WAL and corresponding migrated locks in M-WAL. The additional network round-trips for lock acquisition and version check at M-WALs leads to extra latency. Even if we disable lock-migration for independent LTs, there are other distributed transactions that may be using the same set of locks, and they could have migrated the locks to M-WALs.

A distributed transaction, on the other hand, benefits from migrated-clustering, while the home-clustering increases the latency of such transactions. Therefore, in steady state, lock-localization must balance the trade-off between *home-clustering* and *migrated-clustering* to benefit the latency of independent LTs and DTs, respectively. To automatically adapt to the mix and frequency of local and distributed transactions, we define a lock-weight w_i for every lock-object i, which represents the running average of DT proportion in the transaction mix. It is calculated at the end of prescribed intervals, defined as time period during which a fixed number, p, transactions (LT or DT) requested the lock-object. The weight after interval n is defined as follows:

$$w_i(n) = f * w_i(n-1) + (1-f) * \left(\frac{c_{DT}(n)}{c_{DT}(n) + c_{LT}(n)}\right) \quad (5.1)$$

Here, $w_i(n-1)$ is the lock-weight calculated after $(n-1)^{\text{st}}$ interval, f is the fractional importance given to previous weight, $c_{DT}(n)$ is the count of distributed transactions (DTs) encountered in the n^{th} interval, and $c_{LT}(n)$ is the count of independent local transactions (LTs) encountered in the n^{th} interval.

Whenever a DT or an independent LT acquires the lock-object i, we increase the respective count, c_{DT} or c_{LT}. At the end of the interval, the lock-weight is updated and counts nullified. If the lock is mostly used by DTs for several intervals, the lock-weight increases, signifying its inclination for *migrated-clustering*. On the other hand, if LTs use the lock frequently, the lock-weight decreases showing inclination for *home-clustering*.

5.5.3 Lock Migration and Repatriation Policy

In M-Lock, locks are selected for migration or repatriation based on their lock-weights.

Migration policy: If the lock-weight is greater than a pre-set threshold (w_{high}), then we migrate the lock. The default threshold is 0.7, meaning that migration should take place if more than 70% of transactions using the lock are DTs.

Repatriation Policy: If the lock-weight of a migrated lock is less than a pre-set threshold (w_{low}), then we repatriate the lock. The default threshold is 0.3, meaning that lock should be repatriated if more than 70% of transactions using the lock are independent LTs. We also make sure that there are no transactions that are waiting for the lock, and a pre-set amount of time has elapsed since the last release of the lock. These conditions regulate the probability of a transaction-abort due to a missing lock.

The fraction f specifies the rate at which the lock-weight adjusts to transaction proportions in the past interval. If it is set to a low value, transaction bursts (LT or DT) would significantly alter the lock-weight. This is unfavorable for the system, as it could trigger frequent lock movement. On the other hand, a high value would measure the proportion over several intervals and trigger movement only when there is sufficient evidence of fluctuation in lock usage.

5.6 Evaluation

We conduct all our experiments on a 30-node cluster, where each node has a 2.4 GHz quad-core Xeon processor with 8GB of RAM. HBase [59], setup on 20 nodes, was the key-

value store. The remaining 10 nodes host Zookeeper and several single-threaded clients that issue distributed and independent local transactions.

We use TPC-C [67] benchmark to evaluate the benefits of M-Lock. The New-Order transaction, which simulates the steps involved in accepting a purchase order from a customer, forms 90% of the benchmark. There are 15 warehouses (`WAREHOUSE` table). Each warehouse serves customers (`CUSTOMER` table) who are themselves assigned to districts (`DISTRICT` table). A purchase order is a set of item-amount pairs, and the order information is stored in the `ORDERS` and `ORDER-LINE` tables. Stock information of the items is stored in the `STOCK` table.

Every relational-table is stored as a HTable in HBase, with each relational record forming a HBase row. Each row of the `DISTRICT` or the `STOCK` table is an entity-group. Every purchase order is assigned a unique number. This number corresponds to specific rows in the `ORDERS`, `ORDER-LINE`, and `NEW-ORDER` tables, and all these rows form an entity-group. All tables in HBase are individually partitioned into 20 regions, and placed on 20 region-servers. When M-Lock is used, H-WALs are placed on 15 region-servers, whereas all M-WALs are stored on a separate set of 5 region-servers.

A customer places an order for k items. By default, an item is served by the customer's home warehouse. However, with a probability p, an item will be purchased from a remote warehouse. By changing the values of k and p, we vary the number of locking-related local transactions (LT) in a DT. The default values are $k = 15$ and $p = 0.01$. Every New-Order DT requires $k + 2$ LTs to acquire locks: k for `STOCK` updates, 1 each for `DISTRICT` and `ORDER` updates.

5.6.1 Impact of M-Lock

For all the experiments in this section, we use only transactions that update objects in multiple entity-groups. In the next section, we study the effect of including independent LTs.

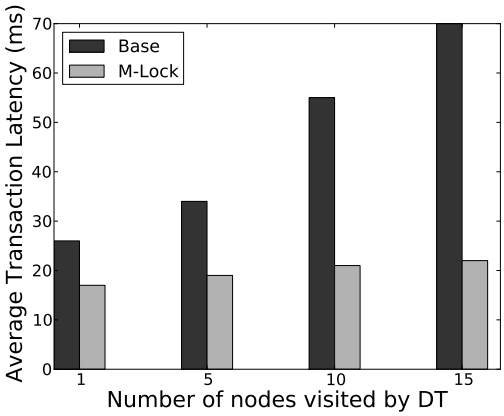

Figure 5.3.: Effect of LT commit and network overhead

Fewer network round-trips to acquire locks

In this experiment, we analyze the benefit of lock-localization at low lock and resource contention, which is a common-case in cloud based key-value stores. To measure the reduction in LT commit and network overhead, we preset the transactions to visit 1, 5, 10 and 15 nodes to acquire locks. We let $p = 0.01$ and $k = 15$.

Figure 5.3 shows the average transaction latency with and without lock-localization. In the base case, every DT uses $15 + 2 = 17$ LTs to acquire locks, irrespective of the number of nodes it visits. As expected, when we increase the number of nodes (X-axis), the network overhead adds to the LT commit overhead, resulting in higher locking-latency, consequently increasing the overall transaction latency. By using M-Lock, these transactions require only 1.1 LTs and 1.1 network round-trips on the average to acquire locks.

In TPC-C, the probability p of a remote warehouse purchase is low (0.01 by default). M-Lock migrates all the locks in a warehouse to a single M-WAL, and most transactions acquire all their locks by visiting only one M-WAL. This is the reason for the low number of LTs (1.1) and network round-trips (1.1) to acquire locks.

Fewer persistent writes to acquire locks

In Figure5.3, the first data point (value on X-axis is 1) shows scenario where DT visits only one node in the base case; this isolates the effect of persistent writes during locking. M-Lock reduces LTs for acquiring locks from 17 to 1.1, by using coarse-grained M-WALs, thereby significantly reducing transaction latency.

Higher transaction throughput

We evaluate the impact of M-Lock on the transaction throughput when there is high contention for locks and storage node resources. In the TPC-C benchmark, a database with m warehouses and d districts per warehouse, allows a maximum of $d * m$ concurrent transactions. This is because every New-Order transaction must obtain a unique order-number from the district of the customer. Once a district is locked, no other transaction can access this district, and there are only $d * m$ unique districts in the database. In our experiments, we set $d = 50$ and $m = 15$, and gradually increased number of customers to observe throughput under contention.

As lock contention increases, transactions either wait for conflicting locks or abort when optimistic concurrency control detects stale reads. The wait time or the probability of abort decreases with a decrease in the average lock-holding times. Since the lock-holding time depends on the total latency of the transaction holding the contended locks, throughput improves when we decrease the average total latency of a transaction. Therefore, by using M-Lock, we can improve the transaction throughput even when there is high contention.

We consider two very different data partitioning schemes to study the effect of M-Lock on the transaction throughput when the contention for locks is high.

Range-partitioning We partition tables using HBase's lexicographical range partitioner. Data placement (and data movement) is left to the key-value store. DTs encountered 7.1 network hops, on average, to acquire locks. Figure 5.4 shows that as we increase the number of customers, the transaction throughput rapidly increases. However, at around

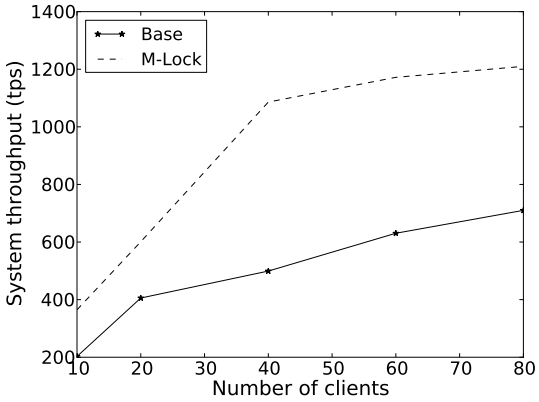

Figure 5.4.: Transaction throughput for range partitioning

80 customers, we observe that due to high lock contention the transaction throughput only improves slowly. However, compared to the base case where M-Lock is not used, we observe that lock localization improves the transaction throughput by more than 25% when the lock contention is high.

Manual partitioning For this experiment, we manually controlled the data placement. This is the case when a database administrator partitions the database with some knowledge about the application. For example, we partition the `STOCK` table such that each node only contains records related to a single warehouse, because most items in a NewOrder transaction are delivered from a single warehouse. Figure 5.5 shows the transaction throughput as we increase the number of customers. We also vary p to assume values between 0.01 and 0.2. At low values of p, persistent write (i.e., LT commit) overhead is dominant in the base case. As p increases, the base case experiences increasing network-related overheads. By using M-Lock, we see a modest increase in throughput (6% to 14%) at different contention levels. This is because the data partitioning in the base case is also quite sophisticated to avoid network overheads.

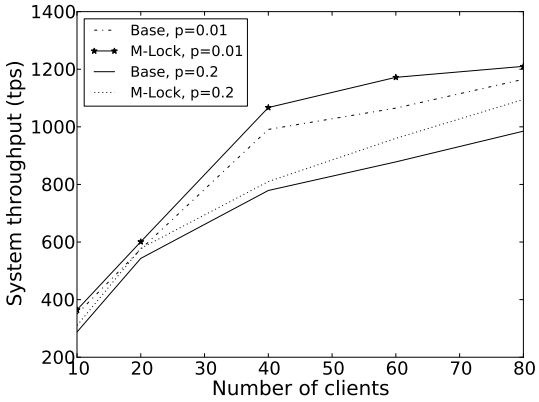

Figure 5.5.: Transaction throughput for manual partitioning

5.6.2 Mix of Single and Multiple Entity-group Transactions

To simulate transactions that only update objects in a single entity-group, we choose a random item and decrease its availability in the `STOCK` table by one. For experiments in this section, we set the high lock-weight threshold (w_{high}) for all locks as 0.7, and low lock-weight threshold (w_{low}) as 0.3. Therefore, we allow lock migration on an object, if more than 70% of the transactions accessing it are DTs.

When 80% of the transactions update objects in multiple entity-groups (DTs) (i.e., 20% of the the transactions only update objects in a single entity-group – independent LTs), as expected, we observe that migration of locks reduces the average latency of DTs. However, the 20% of the transactions that updated only objects in a single entity-group suffered an average latency increase of 42%, due to remote lock acquisitions from M-WALs. Due to the low LT proportion, overall system throughput still improves. Note that the cost of migration is imposed only once when the protocol is triggered. Subsequent transactions do not suffer any delay due to the protocol. When 80% of the transactions only update objects in a single entity-group, as expected, lock-migrations were extremely rare, and the throughput of the baseline and M-Lock were observed to be similar. M-Lock does not introduce any significant overhead when it is not being invoked.

It is important to note that the protocols should not be operated with a lock-weight thresholds (w_{high} or w_{low}) close to 0.5. This could lead to frequent migrations and repatriations as the system tries to choose a balance point in the trade-off between placing the locks at H-WALs or M-WALs. For example, a burst of DTs for a short duration could increase the lock-weights to a value above 0.5, thereby triggering migrations. A subsequent burst of LTs could reduce the low-weight to a value below 0.5 and trigger repatriations. Setting the thresholds far apart, along with the use of running average to calculate lock-weights, will curb the frequent movement of locks. Furthermore, protocols will be triggered only when sufficient evidence about change in transaction access patterns is observed in the past several intervals.

5.6.3 Latency of Lock Migration

Since locks can be migrated in parallel, latency for migration is roughly independent of the number of locks being migrated. Our experiments show that more than 80% of the lock migration latency is easily overlapped with the *Write to shadows* and *Persist Metadata* steps of the DT protocol, when migration is performed by the transaction. Furthermore, migrations and repatriations are triggered gradually over-time when individual lock-weights exceed set thresholds.

5.7 Discussion

Our evaluation shows that M-Lock is beneficial when workload contains frequent distributed transactions executing on arbitrary entity-groups spread across multiple nodes. Its benefits are limited if the transaction access patterns are known apriori and the data can be statically partitioned to suit them. In such cases, M-Lock can be disabled by the administrator leading to no extra overhead to the baseline system. More importantly, M-Lock enables the programmer to *dynamically* impose an extra level of clustering (lock-clustering), on top of the data-clustering imposed during creation of entity-groups. Compared to data objects, lock objects are light-weight in terms of size and memory usage. Consequently, lock ob-

jects are more suitable for dynamic movement between nodes and localization on a small set of nodes.

M-Lock is also useful to balance the latency trade-off, when there is sufficient skew in the LT-DT transaction proportions. Since M-Lock uses an online model to measure the skew, it is important for the observed skew to persist for a certain duration. If the workload is composed of only local transactions (LTs) on single entity-groups, M-Lock can be simply disabled, reverting the system to its baseline behavior. The model can be extended with other factors such as temporal affinities of locks to specific application layer processes.

Policies for migration and repatriation can be extended to consider the observed latencies of distributed transactions, through a feedback loop. This would further limit the lock movement to only those cases when localization is needed – e.g., when transactions violate SLOs (service-level-objectives), or when other system bottlenecks reduce overall throughput.

5.8 Chapter Summary

We presented M-Lock, a new lock-localization system that selectively localizes locks, to reduce the locking overheads of distributed transactions. We described new protocols for consistent migration and repatriation of locks, and new policies that guide the localization. M-Lock is shown to improve distributed transaction throughput on range-partitioned stores.

6 VAYU: ACCELERATING STREAM PROCESSING APPLICATIONS THROUGH DYNAMIC NETWORK-AWARE TOPOLOGY RE-OPTIMIZATION

Stream processing engines (SE) such as Borealis [7], Samza [9], and Storm [10] are commonly used to process continuous streams of data originating from distributed sensors, user-activity logs, and database transactions. Stream processing applications include online learning of complex machine learning models over streaming data [71–73]. A diverse set of applications have been successfully developed, including click-stream analysis [41], tracking malicious activity (spam classification, intrusion detection), real-time analysis of micro-blogs and tweets [74], and ad-click mining [75].

Stream engines code the application workflow as a directed acyclic graph (DAG) of operators, referred to as a *topology*. Tuples are processed in a pipelined fashion as they traverse through the topology. To achieve exactly-once processing of all tuples and to avoid buffer-overflows in the pipeline, stream engines adopt coarse-grained fault-tolerance and flow-control mechanisms. In this setting, even a single slow pipeline-stage affects the throughput of the entire pipeline. To sustain high pipeline-throughput over long execution periods, it is necessary to dynamically detect and diagnose any pipeline-bottlenecks.

With the emergence of cloud-computing solutions, stream engines are often deployed on cloud-based virtual machines. Instead of stand-alone deployments, they co-exist alongside other compute and storage systems, such as MapReduce for batch-processing and key-value stores for data. The orchestration of cluster resources among these systems is handled by a global cluster scheduler such as Mesos [76]. In such deployments, stream engines experience network heterogeneity due to several factors.

Co-hosted VM: Two virtual machines (VM) hosted on the same physical machine interfere in their network usage. The bandwidth observed by one VM depends on the traffic flowing into/ from the other VM.

Co-hosted Framework: Cluster managers such as Mesos may allot compute slots to two different frameworks – batch-processing framework such as MapReduce, and a stream-processing framework – on the same physical machine or VM. This leads to network interference and un-even bandwidth availability to the different frameworks.

Co-hosted Topology: Even if only a stream processing system occupies all slots on the entire physical machine, multiple stream-topologies may be scheduled on these slots leading to network interference.

Furthermore, as SEs are deployed for long time periods, workload variations are commonly observed (e.g., activity of energy-measuring sensors exhibits temporal variation), leading to temporal skew in CPU utilization. Thus, for efficient stream processing, stream engines must effectively diagnose any pipeline bottlenecks induced by heterogeniety in compute and network resources.

To detect and diagnose temporally varying pipeline-bottlenecks, SEs need a feedback-driven control loop. Furthermore, the diagnosis phase, which involves changing the topology routes, should cause least disruption to the tuple-throughput of the stream engine. Resilient substitution [8] has been proposed recently as a general technique to scale or re-assign topology operators. This is an expensive operation, since it involves multiple steps: stopping the stream, spawning new operators, copying the necessary state from old operator, refreshing network connections, and re-starting the stream. However, frequent invocation of such high overhead methods to react to dynamic bottlenecks is infeasible.

To allievate these problems, we propose `VAYU`, a per-topology controller, which uses novel methods and protocols for dynamic, network-aware routing and on-the-fly topology modifications. We rely on three novel techniques to achieve network-aware routing: (i) representing topology link structure using *route-maps*; (ii) consistent hashing for fine-grained key-space management and routing of tuples; feedback information about resource bottlenecks is translated to key-space mapping; and (iii) a light-weight, fault-tolerant protocol for atomic route-map update. By applying novel heuristics on the topology performance (feedback) metrics, the controller determines efficient route-maps, which encode tuple-routing information and also the topology link structure. These new *route-maps* are atomically in-

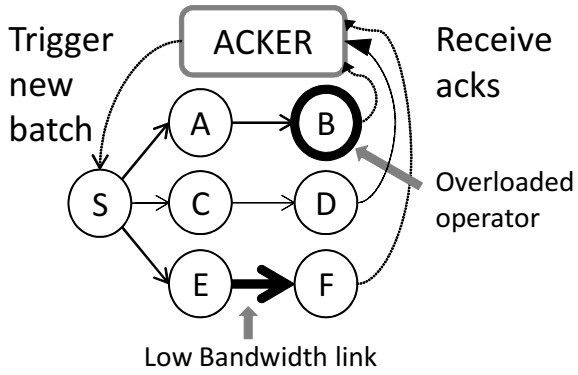

Figure 6.1.: Flow control and fault tolerance mechanism in Storm

jected into multiple operators, on-the-fly, using a light-weight, fault-tolerant protocol, for fast topology re-optimization.

We implement our algorithms and protocols in Storm [10]. In the context of three real applications, we demonstrate that VAYU achieves significant performance improvements, 20% to 200% depending on the bottleneck. Furthermore, we show that our improvements are robust to highly dynamic network state, as well as complex congestion patterns.

6.1 Motivation and Overview

We start by describing a low-overhead method for flow-control and fault-tolerance employed by Storm. We then discuss how pipeline-bottlenecks can severely affect throughput, particularly with the mechanisms employed by Storm. Finally, we mention the shortcomings of traditional schedulers in solving this problem. We use these arguments to motivate our proposed solution.

6.1.1 Flow Control and Fault Tolerance Mechanism in Storm

In Storm, a topology is a graph whose nodes are operators (spouts and bolts) and edges are virtual connections among operators. For simplicity, assume that every topology has a source operator (also called a spout). Figure 6.1 shows an example topology. Tuples

enter the topology through the spout. For every tuple (tup_s) that enters the topology, Storm keeps track of all the tuples that are emitted by bolts as an after-effect of observing tup_s. The emitted tuples are called descendents of tup_s. When a bolt receives tup_s and emits tup_d, it acknowledges by sending *ack* tuples, one each for tup_s and tup_d, to a system managed *Acker* thread. The *Acker* thread calculates the XOR of all identifiers (extracted from *ack* tuples) that it receives. If the topology is a directed acyclic graph (DAG), a tuple's descendents form a bounded tree. In such cases, the *Acker* thread receives the identifier of every tuple twice, and XOR of all these tuple identifiers will result in a zero. Upon observing a zero, the *Acker* thread marks the source tuple tup_s as successfully processed, and a new source tuple is allowed to enter the topology, leading to end-to-end flow control. If the *Acker* thread does not receive acknowledgements for all descendents within a user-defined threshold time interval, then it forces the source tuple (tup_s) to re-enter the topology, leading to re-processing of the entire descendent tree. This method guarantees atleast-once processing of all tuples.

Building upon the basic technique, more stringent guarantees such as exactly-once processing of tuples can be achieved [10]. To this end, tuples are split into batches with monotonically increasing identifiers. Once the *Acker* thread receives acknowledgements for all descendents created by tuples in a batch, it forces the operators to commit the state created by that batch. During commit, the state is tagged with the identifier of the last observed batch. When a batch is replayed, owing to a fault in the system, the committed batch-identifiers are used to ensure that state is not updated twice by tuples of the replayed-batch. More details about the mechanism can be found in [10].

Drawbacks of the flow-control mechanism

The fault-tolerance mechanism described above has low overhead, since it needs constant space per descendent tree (one variable for XORing all tuple-identifiers in a batch), irrespective of the number of tuples in the tree. However, the flow control aspect of the method is highly sensitive to rate of *ack* emission by different operators processing tuples

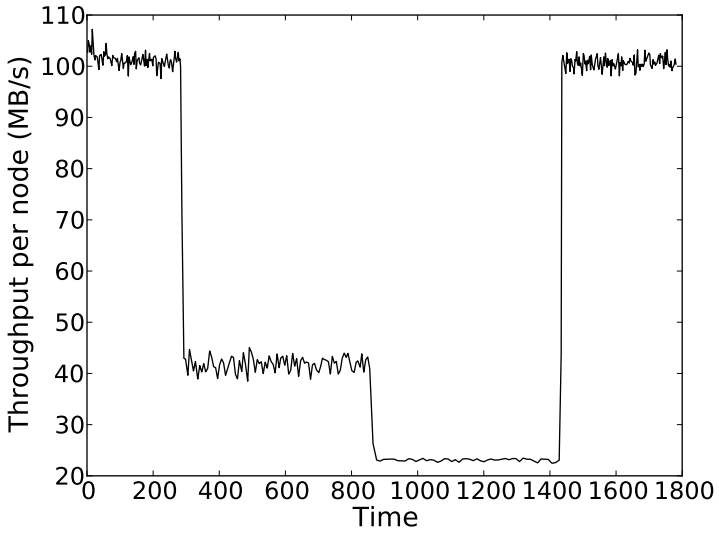

Figure 6.2.: Effect of choking a single random node. Bandwidth choked to 400 Mb/s at 300 sec mark, and to 200 Mb/s at 850 sec, and completely unchoked at 1400 sec mark

in the batch. Consider the topology shown in figure 6.1. Assume that the link between operators E and F has low bandwidth. This leads to low rate of tuple movement in S-E-F path of the pipeline. Consequently, the rate of *ack* emission by F is low. The *Acker* thread emits new batches into the system only after reception of acks from old-batches. Furthermore, a constant number of batches traverse the topology at any time. This method of matching sending-rate to the ack-rate is called *ack-clocking* in computer networks [77]. Due to the low rate of ack emission by F, *ack-clocking* ensures that tuples traverse S-E-F path with the rate dictated by the slowest segment (low-bandwidth link). This rate-matching avoids queue overflows and packet drops. However, since the *Acker* thread applies ack-clocking to entire batches, overall batch-emission rate decreases. Consequently, throughput of other pipeline-paths (S-C-D), with potentially high-capacity, also decreases. This leads to low-utilization of resources, and low system throughput. Similar throughput decrease can be observed due to excessive processing delays at an overloaded operator (operator B in figure 6.1). Note that this phenomenon occurs even if the batch contains only one tuple, but is replicated and forwarded onto different topology-paths. Figure 6.2 shows the throughput

decrease observed in a *tweet hashtag counting* application, when the receiver bandwidth of a randomly chosen node is dynamically varied. In the topology, one set of operators read tweets from external files and passes them to another set of counter-operators. All operators are placed on 16 nodes, ensuring load-balance. The topology throughput drops by more than 50% when bandwidth is reduced from 940 Mb/s (gigabit network) to 400 Mb/s. Also observe that the original throughput is restored when the link bandwidth is restored. This demonstrates how one choked link can cause throughput decrease in the entire topology, due to the coarse-grained *ack-clocking* mechanism.

Maintaining high throughput in the presence of conservative flow-control mechanisms requires dynamic, network-aware re-routing of data to balance load and increase resource (compute and network) utilization.

6.1.2 Overview of Proposed System

In this chapter, we propose light-weight methods for network-aware routing, which is a combination of compute/ network load-balancing, along with efficient topology re-optimization. Every operator chooses the destination operator for its outgoing tuples based on *route-maps*. Route-maps contain information on the type and proportion of traffic for each destination operator. The per-topology controller periodically collects metrics from the system. Based on the observed bottlenecks, the controller computes new route-maps that minimize the maximum network and CPU utilization (Section 6.2). The resultant route-maps are installed in a consistent manner on a running cluster, using a light-weight *atomic route-update* protocol (Section 6.3). The key insight in the solution is to allow adaptive tuple routing at the sender operators, by sending them feedback information about CPU and network conditions of downstream nodes.

6.2 Dynamic Network Aware Stream Routing

In a stream-topology, tuples are communicated among operators running on compute nodes. We refer to *grouping* as the pattern of tuple-routing by a set of senders (upstream

operators) to a set of receivers (downstream operators). A topology is expressed as a series of groupings between operators. The following are two most common groupings observed in topologies: (i) *shuffle grouping:* upstream operators route each tuple to a random downstream operator; and (ii) *fields grouping:* all tuples with the same values for a given set of tuple-fields – also called a *key* – are routed to the same downstream operator. Fields-grouping is used for aggregation or reduction of tuples with the same key.

6.2.1 Factors Affecting Grouping Throughput

Several factors affect the grouping throughput, measured as the tuple processing rate of downstream operators.

Network bandwidth skew: The available network bandwidth at downstream operators may be skewed. If the stream is network bound, the queueing delay at the operator with lesser bandwidth would significantly affect the grouping-throughput.

Per-key tuple-count skew: In *fields grouping*, there may be a skew in the number of tuples per key – some keys may be heavy-hitters. For example, in a sensor-processing application, some sensors could be more active than others. Consequently, even if keys are randomly hashed with a good hash function, the downstream operator receiving the tuples bearing the heavy-hitter key becomes a bottleneck.

Current systems, such as Storm, use modulo-based hashing to realize fields grouping. A tuple with key k is sent to the operator with index (i), where $i = hash(k)\,\%\,m$, where $hash$ denotes the hash function and m is the number of downstream operators. If the number of distinct keys is large, this method leads to good load-balancing among downstream operators. However, in the context of many real applications, the method suffers from several drawbacks:

Inability to accommodate downstream skew: As discussed, *network bandwidth skew* and *per-key tuple-count skew* affect the grouping-throughput. These criteria need to be incorporated in the routing strategy of upstream operators for efficient load-balancing among downstream operators. Modulo-based hashing methods do not allow this flexibility.

Large overhead while scaling: Consider the process of adding a new downstream operator. With modulo based hashing, many keys would be re-mapped to different downstream operators. If downstream operators contain state, key-remapping entails high-overhead state movement between operators.

Lack of specificity: It is not possible to assign a particular heavy-hitter key to a specific operator with more resources.

6.2.2 Consistent Hashing

To avoid the drawbacks of modulo-based hashing, we use a variant of consistent hashing [78]. Consistent hashing was first used in distributed hash table (DHT) implementations to accommodate frequent node additions and removals in peer-to-peer systems. In our case, we use consistent hashing primarily to encode the fine-grained information about changing network-capacities and workload imbalance among downstream operators.

In consistent hashing, keys are hashed into a range, say, -2^{31} to $2^{31} - 1$, using a hash function. The range is divided into p contiguous partitions, termed as *buckets*. Assuming there are m downstream operators, initially, each downstream operator is assigned p/m buckets, chosen randomly without replacement. This random assignment leads to load-balance among downstream operators if the bucket count (p) is large and if all downstream operators have equal CPU and network capacities.

6.2.3 Fine-grained Resource Assignment

Random assignment of buckets to operators is not sufficient to account for fine-grained *network bandwidth skew* and *per-key tuple-count skew*, in fields-grouping. To this end, the controller periodically collects the following statistic for all buckets in all operators: *per-bucket per-batch tuple-count*, equal to count of tuples received by the bucket in the last batch. It also collects the following system metrics: i) CPU capacities of nodes (measured as millions of instructions per second (mips)); ii) network bandwidths (in and out) of all node-pairs; and iii) system throughput. The controller uses these metrics to increase the

pipeline throughput by appropriately assigning the buckets to operators, which are hosted on physical nodes.

Since it is difficult to model the throughput of a complex pipeline, state-of-the-art schedulers aim to load-balance the nodes while decreasing the amount of network traffic [45, 46]. In the same spirit, our controller first balances compute requirements of all pipeline stages by proportionately increasing the cpu-weight (share of total compute-capacity) of cpu-constrained stages. Later, it assigns buckets to operators so as to minimize the maximum CPU and network utilization. For simplicity, assume every node has a replica of all different operator-types. In this setting, a bucket can be assigned to its operator-replica on any node. The resource assignment problem, formulated as an integer programming problem, is an extension of the one used by the COLA [45] scheduler.

Resource assignment problem formulation

For both shuffle and fields grouping, each key-space partition forms a bucket. Let B denote the set of all buckets. If $b \in B$ is a bucket, $D(b)$ denotes the computation-rate (*mips*) used to process data received by b. Let N denote the set of nodes. For a node $n \in N$, $C(n)$ denotes the computing capacity (*mips*) of the node. For any assignment of buckets to nodes, we set the decision variable $x_{b,n}$ to 1 if bucket b is assigned to an operator-replica on node n, otherwise it is set to 0. Let $S(n)$ denote the sum of compute-rates of all buckets assigned to n. In terms of the decision variables, $S(n) = \sum_b x_{b,n} * D(b)$. For buckets $b_1, b_2 \in B$, let $F(b_1, b_2)$ denote the rate of data-flow between the two buckets in the stream-topology. Using the decision variable, for nodes $u, v \in N$, the rate of data flowing from u to v, can be defined as $F(u, v) = \sum_{b_1, b_2 \in B} x_{b1,u} * x_{b2,v} * F(b_1, b_2)$. Let $R(u, v)$ denote the actual bandwidth between the two nodes.

Controller assigns buckets to nodes (specified by variables $x_{b,n}$) based on the solution of the following integer program:

$$\begin{aligned} \text{minimize} \quad & w_{cu} * \overbrace{\max_{n} S(n)/C(n)}^{\text{CU}} \\ & + w_{nu} * \underbrace{\max_{u,v} F(u,v)/R(u,v)}_{\text{NU}} \\ \text{subject to} \quad & x_{b,n} \in \{0,1\} \; \forall \; b,n; \quad \textstyle\sum_n x_{b,n} = 1 \; \forall \; b \end{aligned} \tag{6.1}$$

The objective of the problem is to minimize a weighted function of two quantities: i) CU: maximum observed CPU utilization; and ii) NU: maximum observed network utilization. The controller also has another competing goal of reducing the total inter-node traffic ($\sum_{u,v \in N} F(u,v)$). Similar to COLA [45], we use a combination of a graph partitioner and load-balancing heuristics to obtain a feasible solution. For large problem sizes (bucket-count is large), graph partitioners are computationally expensive. In such cases, we implement a recently proposed *re-streaming algorithm* for multi-constraint graph partitioning [79], which is shown to be competitive with offline-graph partitioners while using limited resources.

The following are key differences of our formulation with respect to the COLA scheduler: i) modelling traffic as data received by fine-grained key-space buckets instead of coarse-grained operators. This allows fine-grained mapping of buckets to nodes, leading to balanced-load even in the presence of per-key tuple count skew; ii) balancing utilization of all network links is added to the objective function. This is important in environments where network capacity can have significant variations.

Accuracy of network bandwidth estimates

Measuring available network bandwidth between nodes in a cluster already running a stream-processing engine is a challenging problem for two reasons: i) the data-sent by measuring tools, such as IPerf [80], interferes with the stream-traffic, thereby returning noisy bandwidth estimates; ii) due to the interference from measurement-traffic, the stream

throughput drops during the measurement-period. To avoid these overheads, the controller mainly uses bandwidth estimates inferred from the rate of tuple-acks emitted by different nodes in the past time-windows. It uses measuring tools for metrics are in-sufficient. Also note that the assignment problem only requires relative bandwidths, instead of accurate values. To avoid frequent bucket re-assignment, controller invokes the control loop only when hysteresis-applied system throughput (rate of tuple-ingestion by the spout) in the recent time windows drops by a threshold percentage (default: 10%) when compared to the long term average. Furthermore, the controller first checks for CPU load imbalance. Once the controller discards the per-key tuple-count skew as the reason for throughput-decrease, it checks for the network bandwidth skew. These policies decrease the usage-frequency of noisy bandwidth estimates, and also helps in setting appropriate weights in the multi-constraint load-balancing heuristics.

The output of the resource assignment problem is a mapping of buckets to operators, referred to as *route-maps*. Using these route-maps, in modified shuffle-grouping, upstream operators choose a random downstream bucket, instead of a downstream operator, when routing tuples.

6.3 Consistent On-The-Fly Topology Modification

In previous sections, we described methods for dynamically changing key-space assignment and overlay topologies. Another novelty in `VAYU` lies in the method for updating routing information at operators to reflect the changes proposed by the controller.

Route Maps as Topology Route Specification: Each operator maintains a *route-map* that specifies the routes on which output messages should be sent after processing incoming messages. A sample route-map entry to specify *fields or shuffle grouping* would be: *send-to*: [10, 344] : [1], [5677, 34345]: [2]. It denotes the assignment of key-space partitions to downstream operators.

To change a running topology on-the-fly, route-maps of all involved operators must be updated in an atomic manner – i.e., all nodes must switch to the new route-maps at the same

time. For example, if upstream operators route tuples based on different *route-maps*, two tuples with same keys may not reach the same downstream operators, thereby violating the semantics of fields-grouping. Similarly, overlay modification needs to be atomic. If only a sub-set of the nodes have received new route-maps and the other nodes are using the old route-maps, then the resultant topology may not satisfy reduction/aggregation semantics.

To achieve atomic route-map update, in this chapter we propose a light-weight protocol for modifying topology route-maps in an atomic manner. We focus on updating the route-maps with least possible interruption to the existing stream traffic. We first describe the relevant runtime properties and constraints of Storm. Next, we describe the proposed protocol and sketch a proof of its correctness.

6.3.1 System Properties

As described in chapter 6.1.1, Storm uses a global acknowledgement mechanism to deal with both network-error and operator-failures during tuple traversal.

Operators in a Storm topology are fail-fast. That is, unlike database nodes that log all their actions into a write-ahead-log, storm operators do not log all input tuples and their corresponding outputs. Only designated operators containing state can checkpoint their local state at batch-boundaries. This design helps in quick re-spawning of a failed operator on another node without the overhead of processing any undo or redo logs. In case of operator failure, the global fault-tolerance mechanism ensures re-delivery of unprocessed tuples. This fail-fast design, unfortunately, does not permit the use of traditional atomic commit protocols such as 2-phase or 3-phase commit protocols, which rely on local write-ahead-logs for participant recovery.

6.3.2 Atomic Route Map Update Protocol

We propose a six-step protocol to assure atomic route-map update. As shown in Figure 6.3, for every topology, there exist two components: a *controller* (part of the scheduler) and a *spout*; both these components have corresponding kafka queues, which form their

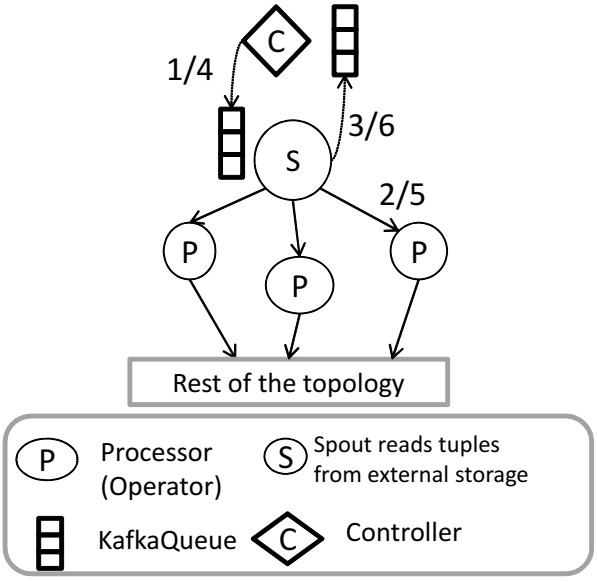

Figure 6.3.: Atomic route-map update protocol

message sources. The spout has the following functionality: (i) it truncates the stream into batches and demarcates them by appending *start-batch* and *end-batch* tuples to the stream at batch-boundaries; and (ii) it emits *tick-tuples* to trigger time-based windowed reductions. Truncating the stream into batches permits the use of global fault-tolerance mechanisms. Note that *start-batch* and *end-batch* tuples traverse the entire topology DAG starting from the spout. Tick-tuples are used to periodically trigger all-reduce (or windowed aggregation operations) on all operators.

To update route-maps, the controller creates new route-maps for each involved operator, tags the new maps with a version number (which increases monotonically), and executes the following six-step protocol.

1) Controller first stores the new route-maps in its local state, durably stored in zookeeper. Later, it sends the new route-maps message, tagged by a version-id, to the spout (S), by placing it in the latter's kafka queue.

2) Spout reads the new route-maps from its kafka queue, appends an *install-routes* command to the message, and sends it to all the involved operators by piggybacking on the next *start-batch* tuple. The spout waits for acknowledgements from involved operators;

this happens through Storm's *Acker* interface. On reception of route-maps, operators do not immediately switch to the new route-maps; they simply append it to their list of route-maps.

3) After receiving acknowledgements from all operators, the spout sends a *routes-installed* confirmation message to the controller by placing it in the latter's kafka queue.

4) On reception of the *routes-installed* message, the controller durably stores the new topology-route-maps in its local-state. As the controller is part of the scheduler, this local-state is stored in zookeeper. The controller now sends a *activate-new-routes* message to the spout by placing it in the latter's kafka queue.

5) On receiving the *activate-new-routes* message, the spout first appends the received message onto the next *start-batch* tuple. The controller then waits for the successful commit of all currently executing batches before sending the piggybacked *start-batch* tuple. Since all operators start using the new route-maps for the same batch, semantics of grouping and reduction among operators is consistent.

6) Once the spout receives all acknowledgements for the *start-batch* tuple containing *activate-new-routes* message, it sends an *activated-new-routes* message to the controller, by placing it in the latter's kafka queue. When the controller receives the message, it marks the successful completion of the protocol.

6.3.3 Correctness of the Protocol

We prove the correctness of the protocol by showing that it does not violate the following safety properties.

No duplicate state changes: Two operators must not update the state for the same key-space bucket in a batch. Our protocol ensures this property since every batch adopts a single route-map version, and all operators operating on a batch follow the same route-map.

Access to complete state: Once an operator assumes ownership of a key-space bucket, it must have access to all the state previously generated for that bucket. This property enables owner operator to process all queries involving keys in its assigned bucket. The protocol satisfies this property as new routes are activated only after ensuring that all previ-

ous batches have committed. Therefore, the new owner of the bucket can fetch any needed state from the persistent store while answering queries.

6.3.4 Protocol Fault Tolerance

This section describes the mechanisms used by the protocol to handle tuple and operator failures at various steps.

Route Installation Phase

The route installation phase is marked complete only after the controller read the *installed-routes* message from the spout and subsequently stored all the versioned route-maps in zookeeper. If installation fails before this point, controller times-out and retries the installation phase. The controller ensures that the protocol does not move to the *activate-new-routes* phase until the previous *route-installation* phase successfully completes. This ordering ensures that all operators are aware of the new routes before any of them starts to send messages along new routes.

Three types of faults are possible: spout failure, operator failure, or tuple loss due to network failure. If spout fails, then controller times-out and retries the phase. In the event of operator failure or tuple loss, the spout times-out in receiving the acknowledgements, and re-tries the installation phase by piggybacking on the next *start-batch* tuple.

Route Activation Phase

Since route-activation is piggybacked on *start-batch* tuples, any operator failure manifests as a regular topology failure. The system reacts to operator failure in two ways: (i) the controller re-spawns the operator with all the latest route-maps information. Since controller is in *route-activation-phase* it is guaranteed that all other live operators have the latest route-maps; and (ii) if the spout emitting the stream tuples times-out, it re-emits the batch tuples. Since, every *start-batch* tuple carries the *route-map* version number that the

operators are required to follow, each operator will follow the correct route-map during next aggregation/ reduction.

Controller Failure

The controller always logs its topology modification related actions in zookeeper. This is done to make it fail-fast. Thus, re-spawning the failed controller is a sufficient fault-tolerance mechanism for the protocol to progress. The re-spawned controller will work-off the state stored in zookeeper. Furthermore, it reads the pending message from its kafka queue. For this reason, it can never miss any messages. Since communication between the spout and the controller always takes place through durable kafka queues, message loss in that communication channel is not possible.

6.3.5 Need for Two Phases

The first phase (route-map installation) is used to ensure two conditions: (i) all involved operators have sufficient resources (memory capacity, connections to new, scaled-out operators, etc.) to exchange buckets as dictated by new route-maps; and (ii) all operators have the new route-maps. Once these two conditions are satisfied, the second phase (route activation phase) can atomically switch to new routes without any system-level interruptions.

Owing to its two-phase nature, the protocol can be extended to scale-out a topology, on-the-fly, through the addition of new nodes.

6.4 Experimental Evaluation

In this section, we present a comprehensive evaluation of the techniques presented in this chapter, namely, routing for dealing with overloaded network-links/cpu; and topology-modification protocol. The goal is to demonstrate the effectiveness and robustness of the proposed techniques. We conduct all experiments on a 30-node cluster. Each node has

a 2.4 GHz quad-core Xeon processor with 8GB of RAM, connected via gigabit ethernet links.

6.4.1 Static versus Dynamic Topologies

Applications

To compare the performance of static and dynamic (network-aware routing) topologies, we implemented three representative streaming applications from different domains: (i) hashtag counting on tweets; (ii) a malicious url detection algorithm [81], representative of an online learning application; and (iii) stream analytics on sensor measurements from DEBS 2014 grand challenge [82].

In hashing counting application, tweets are emitted to random counter-operators which maintain a count of unique hashtags observed in tweets. The application mainly uses shuffle-grouping. The sensor analytics application [82] addresses analysis of energy consumption measurements from sensors deployed in a smart grid. The *load prediction* query forecasts the energy demand of a smart plug in the near future based on measurements sent by its sensor in a past time-window. Another query detects *outliers* among the smart plugs based on their past usage. Both queries use fields-grouping for time-windowed processing of measurements from a particular sensor. Two issues manifest in a stream-engine that processes measurements from a large number of sensors. The rate of emitted measurements across different sensors is prone to skew: temporal skew occurs when sensors in one continent do not emit observations (during night-time) while sensors in other continents are active; spatial skew occurs when some sensors emit frequent measurements due to heavy usage. Fine-grained tracking of bucket (key-space) load is key to handle throughput loss due to skew.

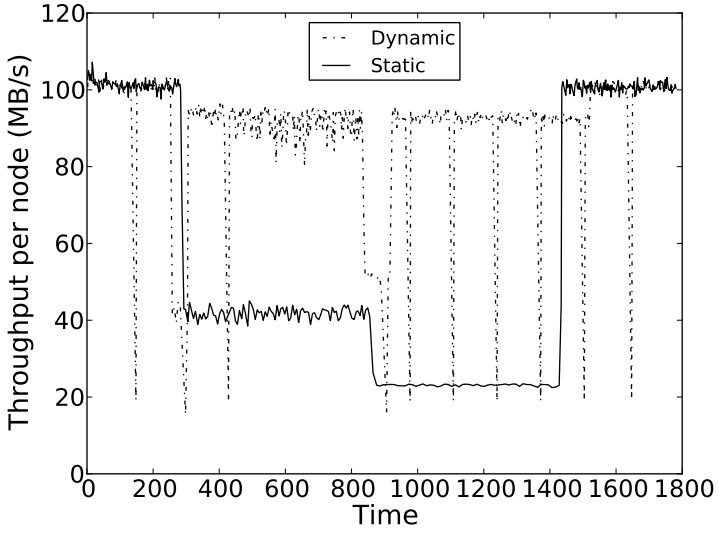

Figure 6.4.: Effect of choking a single random node. Bandwidth choked to 400 Mb/s at 300 sec mark, and to 200 Mb/s at 850 sec mark, and completely unchoked at 1400 sec mark

Effect of Link Congestion on Static and Dynamic Topologies

To test the effect of dynamic network-aware routing, we randomly choose certain nodes hosting counter-operators (in hashtag counting app) and decrease their in-bandwidth using *traffic control (TC)* and *intermediate functional block (IFB)* tools in linux. The controller detects the choked receiver via the metrics interface. Subsequently, the controller creates new-route maps and installs them in the topology.

Impact of Dynamic congestion

In this set of experiments, we investigate the response behavior of the controller when one link-state is dynamically varied. Here, a random learner is chosen and its in-bandwidth choked according to the following pattern: at the 300 sec mark, the in-bandwidth is choked to 400 Mb/s; at 850 sec mark, it is choked to 200 Mb/s; at 1400 sec mark, it is unchoked to its full gigabit bandwidth. Figure 6.4 shows the throughput-per-node (volume of input

tweets processed per node) as a function of time, for the *hashtag counting* application. Results for the other two applications showed a similar pattern.

For static overlays, once choking sets in, throughput drops significantly (almost 60% for 400 Mb/s and 80% for 200 Mb/s), even though only a single link is choked. In the case of dynamic overlays, once the controller detects throughput loss and new route-maps are incorporated into the topology, we observe a substantial increase in overall throughput. After 400 Mb/s choke, throughput returns to 95 MB/s from 45 MB/s, corresponding roughly to 200% increase in throughput. For 200 Mb/s choke, performance increases by almost 300% when choked link utilization is reduced. Our experiments demonstrate that network-aware routing can be used to recover a substantial part of this lost performance. Note the downward spikes (intermittent loss of throughput), in the dynamic case. They coincide with the times when the controller triggers either network bandwidth measurements or new route-map installation, after observing changes in system throughput. Activation of new route-map requires flushing of current batches to avoid semantic inconsistencies. The loss in throughput for a brief time window leads to more accurate load-balancing and consequent increase in throughput over the long-term. Futhermore, due to hysteresis in measurements, the controller takes a while to react to loss in system throughput. The reaction time of the controller can be tuned by adjusting the hysteresis parameters.

It is important to note that while performance improvement from dynamic, network-aware routing are substantial, it could not completely regain the lost throughput. This is due to two reasons: i) the measure of available-bandwidth at learners is not very-accurate; and ii) the stream-traffic removed from the choked nodes is now processed by the other nodes, along with their own traffic, thereby increasing the total batch-processing delay.

Impact of Complex Congestion Patterns in Link State

To realize complex congestion patterns typically observed in cloud settings when the system runs for long time periods, we choke nodes based on sampling a distribution. Figure 6.5 shows the performance of static and dynamic topologies under multi-node complex

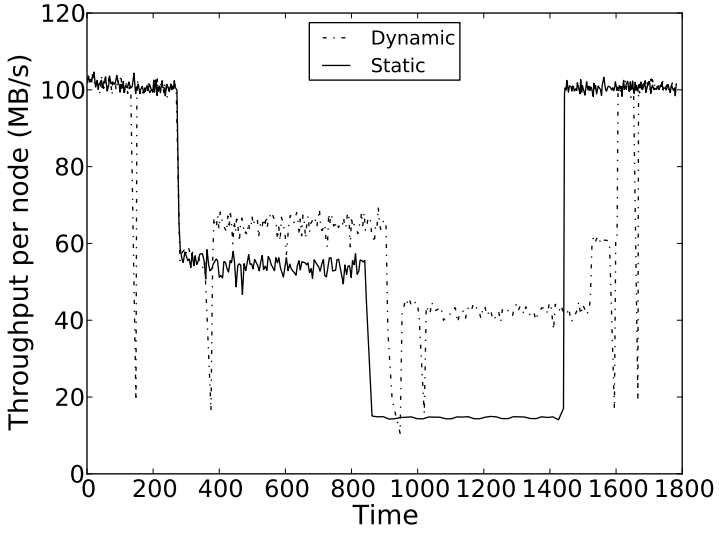

Figure 6.5.: Effect of choking multiple nodes (complex congestion). At 300 sec mark, bandwidths are sampled from Normal (mean=700Mb/s, sd=200Mb/s). At 800 sec mark, bandwidths follow Normal (mean=400Mb/s, sd=300Mb/s). At 1450 sec mark, all nodes are unchoked

congestion pattern. At 300 sec mark, in-bandwidths are sampled from a *normal* distribution (mean=700Mb/s, sd=200Mb/s). At 800 sec mark, bandwidths follow normal distribution (mean=400Mb/s, sd=300Mb/s). At 1450 sec mark, all nodes are unchoked. As is evident from the results, when the standard-deviation is large (300Mb/s), the improvement in throughput is large (more than 50%). This arises due to the need for accurate load-balancing between slow and fast links. When standard deviation is small, the difference between in-bandwidths is not significant. Consequently, the need for dynamic load-balancing diminishes and the performance gains are commensurately lower.

Impact of skew in per-key tuple count

To test the impact of CPU load imbalance created by per-key tuple count, we use the sensor-analytics application, which relies on fields-grouping for analyzing the pattern of

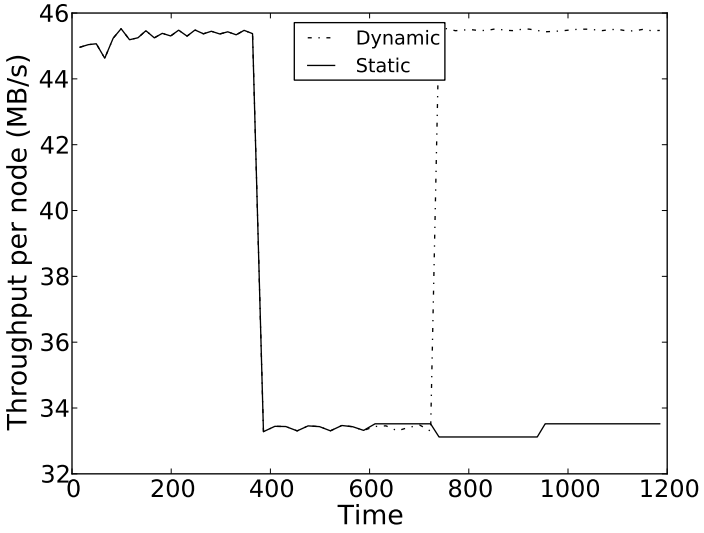

Figure 6.6.: Impact of operator load skew. Initially all sensors emit as same rate (5 tuples/s). At 360 sec mark, sensor emission rates follow Normal (mean=5 tuples/s, sd=5 tuples/s)

individual sensor measurements over a time-window. We create skew among sensors by sampling their measurement emission rates from a normal distribution.

Figure 6.6 shows the throughput increase due to fine-grained balancing of tuple-skew by the controller. At the 250 sec mark, the rate of sensor-measurement emission is sampled from a normal distribution(mean=5 tuples/s, sd=5 tuples/s). The small number of nodes hosting the sensors with high rate of emission, process more data, adding extra delay to the batch-pipeline. The controller detects the reduced throughput and triggers load-balancing of the fine-grained key-space buckets, leading to sensor re-assignment. Throughput improves by more than 20% after bucket reassignment.

The above results show the benefit of fine-grained tracking of per-sensor activity via key-space bucket monitoring. Consistent hashing enables fine-grained key-space partitioning, which is needed to track, diagnose and rectify skewed sensor activity. Selective bucket re-assignment leads to selective-sensor allocation to nodes.

6.5 Discussion

Using large number of buckets leads to fine-grained tracking of key-space activity. However, it also increases overhead of metrics collection. Thus, we plan to extend VAYU with methods for dynamic merging of contiguous buckets when they exhibit similar activity, and methods for splitting buckets when finer-grained tracking is needed.

Current implementation of VAYU does not support dynamic physical resource scaling through node additions. However, the two phase nature of route-map update protocol, provides clear interfaces for such extensions. In particular, the route-installation phase (first-phase) can be used to establish connections with new nodes. Due to the strict ordering between phases, hot-swapping of route-maps in second-phase is guaranteed to maintain correct operation semantics.

6.6 Chapter Summary

Dynamic compute and network overheads can significantly impact the performance of streaming systems. In this chapter, we present efficient techniques for dynamic topology re-optimization, through the use of a feedback-driven control loop. We present a novel technique for network-aware tuple routing using consistent hashing. that improves stream flow throughput in the presence of a number of runtime overheads. To enable fast topology re-optimization with least system-disruption, we present a light-weight, fault-tolerant protocol. All of the proposed techniques are implemented in a real system and comprehensively validated on three real applications. We demonstrate significant improvement in performance (20% to 200%), while dealing with various compute and network bottlenecks. We show that our performance improvements are robust to dynamic changes, as well as complex congestion patterns. Given the importance of stream processing systems and the ubiquity of dynamic network state in cloud environments, our results represent a significant and practical improvement.

7 RE-OPTIMIZING ASYNCHRONOUS GROUP COMMUNICATION OVERLAYS

Emerging online-learning applications [83, 84] have complex topologies and often use structured overlays for group-communication operations. In an online learning application, learner operators process their respective partitions of the input stream (also called an example stream), and update their individual models. Concurrently, learners also synchronize their models periodically using group communication primitives – typically an all-reduce [83]. When compared to traditional streaming workloads, online learning workloads have the following distinct characteristics: (i) models may be large (tens to hundreds of megabytes), which leads to large state transfers between operators; (ii) complex, pipelined group communication (all-reduce) topologies are needed to synchronize potentially large state among all learners. Dynamic orchestration of complex topologies to maintain high throughput in the presence of bottlenecks requires novel techniques. In this chapter, we propose an algorithm for determining spanning tree overlays for pipelined all-reduce operations, which explicitly accounts for dynamic network-state.

Schedulers built for traditional SEs [45, 46] do not effectively handle new challenges posed by deployment and usage requirements. First, schedulers take a static topology as input. However, for complex group communication operations such as all-reduce, the most efficient overlay structure depends on the network and compute resources allocated to the learner-operators. For this reason, current schedulers are unable to optimize complex communication structures, since they assume that the best topology is known a-priori.

In an online learning application, operators that train the model using training-examples are termed *learners*. For accurate model training, the model (in our case of a stochastic gradient descent, also called a weight vector) is periodically synchronized among learners, using an all-reduce operation. Static binary trees are among the most commonly used overlays in systems [83, 84] for pipelined all-reduce group communication. Each learner divides its weight vector into slices. Each slice traverses up the tree during reduction and

down the tree during broadcast. The tree structure is effective for pipelining slices of a large model or for sending complete models in quick succession, as the network links in a tree allow un-congested traffic flow. However, the throughput of a pipelined tree is heavily influenced by the slowest link. In streaming systems, since model synchronization traffic flows alongside the regular example traffic (input tuples used to train models), the available bandwidth on different links may vary significantly. In this scenario, the tree overlay must be dynamically optimized to suitably use links with higher available bandwidth. Furthermore, in a tree, different nodes (learners) emit their model and receive the reduced model at different times. In this scenario, tree overlays must ensure that low-bandwidth links do not significantly impact model synchronization times for all of the nodes. With these high-level goals, we first define a few terms and formally state the problem. We then describe our method for dynamic computation of efficient overlays.

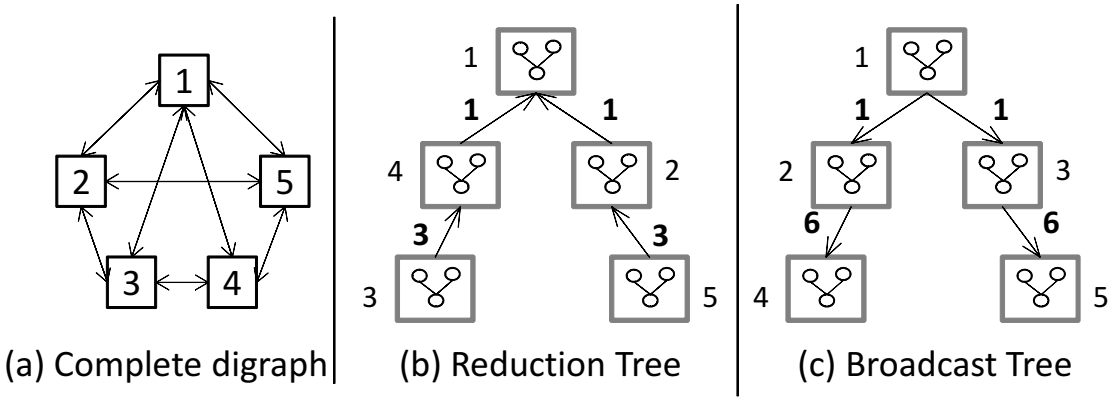

Figure 7.1.: Sample reduction and broadcast trees generated by MWD heuristic

7.1 Problem Formulation

Let G be a complete directed graph, where nodes denote machines hosting learner operators and edges represent potential overlay links. The directed edge-weight between nodes s and d is $W_{s,d}$, which is the time taken to transmit a byte of data from s to d (inverse

of link bandwidth). Figure 7.1 shows a sample reduction tree and a sample broadcast tree generated from the same graph.

Let $t_s(i)$ denote the start-time for model synchronization at node i. This is the time when node i attempts to reduce its first model-slice with the corresponding model-slices received from its children. Node i subsequently sends the reduced model-slice to its parent in reduction tree. Let $t_e(i)$ denote the end-time for model synchronization at node i. This is the time when node i receives the last reduced model-slice from its parent in broadcast tree. Let $t_{ms}(i)$ denote the model synchronization time for node i, defined as $t_e(i) - t_s(i)$.

Problem Statement: Generate a spanning tree for pipelined all-reduce that minimizes the average model synchronization time, over all nodes.

$$\text{minimize} \quad (\frac{1}{|N|}) * \sum_{i \in N} \; t_{ms}(i) \tag{7.1}$$

In message passing (MPI) systems, overlays are chosen to minimize the maximum completion time of a group communication operation. In contrast, we focus on optimizing the average completion time. This is because, in streaming systems, group communication operations do not follow barriers; they are triggered periodically irrespective of the completion of the previous operations. Furthermore, in online learning applications, average completion time is also an indirect measure of model mixing-rate.

We extend the *Min Weighted Degree Tree* (MWD) heuristic [85] to generate both pipelined broadcast and pipelined reduction trees.

7.2 Pipelined All-reduce Overlay Generation

To compute a broadcast spanning tree topology, MWD (Algorithm 1) works as follows: Initially the spanning tree contains only the root node. In each iteration, the algorithm adds the least weighted out-edge (intuitively, the fastest out-link), say (u, v), to the spanning tree. The edge-weights of all other outgoing-edges from node u that are not already in the spanning tree, are incremented by the time node u spends in broadcasting to the previously chosen children. This time is equal to sum of edge-weights to current children ($\sum_{v' \in children(u)}(W_{u,v'})$). The algorithm continues to select edges until all nodes are in-

cluded in the spanning tree. When selecting edges, nodes in the spanning tree that have less than a preset threshold k edges are given preference. This parameter can be tuned to generate trees with different branching factors.

To generate spanning tree for all-reduce, we use the MWD heuristic (Algorithm 1) to first find a reduction tree, starting from a given root (v). Since it is a reduction tree, edges coming into the spanning tree (in-edges) are considered when choosing the min-edge. After the reduction spanning tree is generated, its edges are removed from the graph. The algorithm is now run on the residual graph to generate a broadcast tree with the same root node (v). The average synchronization path length, over all nodes, is calculated using the generated reduction and broadcast trees. The algorithm repeats the above steps, each time with a different node as the root of spanning tree. The final chosen root node is the one with the least average synchronization path length, and the final reduction and broadcast trees are the ones generated by the chosen root node. Inside each node, one learner operator is chosen as *leader* and all other learners are connected to it. The leader reduces the model slices generated by node-local learners before sending them to other nodes. It also performs node-local broadcast.

Using Fibonacci heaps for edge-set implementation, the algorithm takes $\mathcal{O}(|E| + |V| \log |E|)$ to generate reduction and broadcast trees for a chosen root, where E is the set of edges and V is the set of nodes. The algorithm is invoked $|V|$ times, one for each chosen root node.

Algorithm MWD has a number of desirable properties. The greedy heuristic builds pipelined spanning trees with low weighted out-degree (sum of out-edge weights) of any node in the broadcast tree; correspondingly low weighted in-degree of any node in reduction tree. This strategy minimizes the choking effect of any one stage in the pipeline.

Before generating the broadcast tree, the algorithm removes the edges used for reduction tree. This eliminates the possibility of a single link being used for both reduction and broadcast. The final chosen root is one that reduces the average model synchronization time of all nodes. Intuitively, the heuristic pushes the congested links closer to the leaves than the root, because a congested link close to the root will lie in the synchronization path

of a large number of nodes, thereby increasing their synchronization time. In Figure 7.1 node 5 has low in-bandwidth. Therefore, the heuristic places it among leaves, avoiding its choked bandwidth from affecting the entire pipeline. Furthermore, if node 5 was made the root, it would receive the model-traffic from two children. This extra traffic further decreases the bandwidth for the concurrent training-example traffic, thereby increasing the choking-effect of that node.

A linear tree, generated by setting k to 1, has better pipeline bandwidth than binary tree (k set to 2). Assuming n nodes in total, in linear tree, $n-1$ nodes are involved in reduction. Even if one of these nodes has less in-bandwidth, it chokes the entire pipeline. Furthermore, for small-sized models, or when the change in model since last synchronization is small, linear tree performs poorly as it takes time proportional to $\mathcal{O}(n)$. On the other hand, binary trees have comparatively less pipeline bandwidth, but offer two benefits: i) since only $n/2$ internal nodes are involved in reductions, ill-effects of choked nodes (upto $n/2$) can be localized by placing them among the leaves; ii) binary trees perform well even for small models due to $\mathcal{O}(\log n)$ height. Therefore, `VAYU` chooses binary trees by default. Ternary tree (k is 3) can be used when there is considerable skew in node capacities, since only $n/3$ nodes form internal nodes. This chapter optimizes spanning trees for all-reduce operation. Furthermore, we assume that the physical network allows all-to-all communication between nodes. We plan to investigate heuristics for optimizing other overlays such as hypercubes for pipelined all-reduce.

7.3 On-the-fly Topology Modification

Route Maps as Topology Route Specification: Each operator maintains a *route-map* that specifies the routes on which output messages should be sent after processing incoming messages. For instance, to specify an all-reduce topology, each route-map entry would be of the form: *receive-from* : [1, 2, 3], *send-to* : [4, 5, 6] . This implies, the operator should wait for input message reception from operators with ids: 1, 2 and 3, before sending the output message to operators with ids: 4, 5 and 6. To change a running topology on-the-fly,

Algorithm 1 Min Weighted Degree Tree

procedure MIN-WEIGHTED-DEGREE-TREE($V, v_s, E, T, b, TreeType$)
  $TreeEdges \leftarrow \emptyset$
  $TreeVertices \leftarrow \{v_s\}$
  for each $e = (u, v) \in E$ **do**
    $cost(u, v) \leftarrow T_{u,v}$
  end for
  while $TreeVertices \neq V$ **do**
    if $TreeType = BroadcastTree$ **then**
      $ReadyVertices \leftarrow \{u \in TreeVertices \mid |children(u)| < b\}$
      $ReadyVertices \leftarrow \{u \in ReadyVertices \mid$ DISTANCE-FROM-ROOT(u) is the least$\}$
      $link(u, v) \leftarrow \{u \in ReadyVertices, v \notin TreeVertices \mid cost(u, v)$ is the least$\}$
      $TreeVertices \leftarrow TreeVertices \cup \{v\}$
      $TreeEdges \leftarrow TreeEdges \cup \{(u, v)\}$
      for each edge $(u, w) \notin TreeEdges$ **do**
        $cost(u, w) \leftarrow \sum_{v' \in children(u)} (T_{u,v'})$
      end for
    end if
    if $TreeType = ReductionTree$ **then**
      $ReadyVertices \leftarrow \{u \in TreeVertices \mid |children(u)| < b\}$
      $ReadyVertices \leftarrow \{u \in ReadyVertices \mid$ DISTANCE-FROM-ROOT(u) is the least$\}$
      $link(u, v) \leftarrow \{v \in ReadyVertices, u \notin TreeVertices \mid cost(u, v)$ is the le ast$\}$
      $TreeVertices \leftarrow TreeVertices \cup \{u\}$
      $TreeEdges \leftarrow TreeEdges \cup \{(u, v)\}$
      for each edge $(w, v) \notin TreeEdges$ **do**
        $cost(w, v) \leftarrow \sum_{u' \in parents(v)} (T_{u',v})$
      end for
    end if
  end while
  return $TreeEdges$
end procedure

route-maps of all involved operators must be updated in an atomic manner – i.e., all nodes must switch to the new route-maps at the same time. For example, if only a sub-set of the nodes have received new route-maps and the other nodes are using the old route-maps, then the resultant topology may not satisfy reduction semantics.

In VAYU, the controller generates new all-reduce overlays based on the observed network/CPU conditions of the nodes hosting the learner-operators. The controller uses the same *atomic route-map update protocol* (section 6.3), described in the previous chapter 6.

To achieve consistent group communication, all operators involved in reduction should follow the same route-maps. To satisfy this constraint, the spout ensures that all the emitted tick-tuples fall into the same batch. This ensures that all operators follow the same route-maps for reduction.

7.4 Experimental Evaluation

In this section, we present a comprehensive evaluation of the adaptive overlay heuristics presented in this chapter. The goal is to demonstrate the effectiveness and robustness of the proposed techniques. We conduct all experiments on a 30-node cluster. Each node has a 2.4 GHz quad-core Xeon processor with 8GB of RAM, connected via gigabit ethernet links.

To test the performance of all-reduce overlays under various network conditions, we implemented a malicious url detection algorithm [81], representative of an online learning application. In the *malicious url detection* application, the learner operators train a linear model using incoming (shuffled) spam urls from multiple sources: tweets, emails, blacklists, etc. For training, each learner runs regularized logistic regression implemented in vowpal-wabbit [86] using stochastic gradient descent (SGD). Learners synchronize their models (weight vectors) through all-reduce operations using a spanning tree overlay imposed on the learners [84]. In literature, other online learning applications have reported weight vectors for 20 million features [87]. To study the performance of reduction pipelines

on weight vectors of varying sizes, we introduce appropriate random features into the dataset. This application mainly uses shuffle-grouping and all-reduce overlays.

Effect of Link Congestion on Static and Dynamic Topologies

To test the effect of dynamic network-aware routing, we randomly choose certain nodes hosting learner-operators (in hashtag counting app) and decrease their in-bandwidth using *traffic control (TC)* and *intermediate functional block (IFB)* tools in linux. The controller detects the choked receiver via the metrics interface. Subsequently, the controller creates new-route maps and installs them in the topology.

7.4.1 Performance of Dynamic Overlays on Group Communications

In typical learning applications, learners periodically communicate to synchronize their models. In the following set of experiments, we compare the average model syncronization times observed in spanning tree overlays obtained through two techniques: a random, static binary tree (baseline) and the proposed MWD approach. In each case, we quantify the impact of link congestion on performance.

Performance Improvement from Min-Weighted-Degree (MWD) Approach for Varying Model Sizes

Figure 7.2 shows the impact of choking link bandwidth on average model sync times for different weight vector sizes. In this experiment, the in-bandwidth of a randomly selected node is choked to 100 Mbit/sec. Learner tasks, hosted on 20 nodes, are involved in the all-reduce operation. Our experiments demonstrate that the proposed MWD approach outperforms the random binary tree by a significant margin (more than two-fold speedup), for different model sizes. To further understand this result, we plot the model sync times observed by various learners in Figure 7.3. In case of MWD, only the latency of the single choked node is affected. This is because the heuristic places the choked node among the

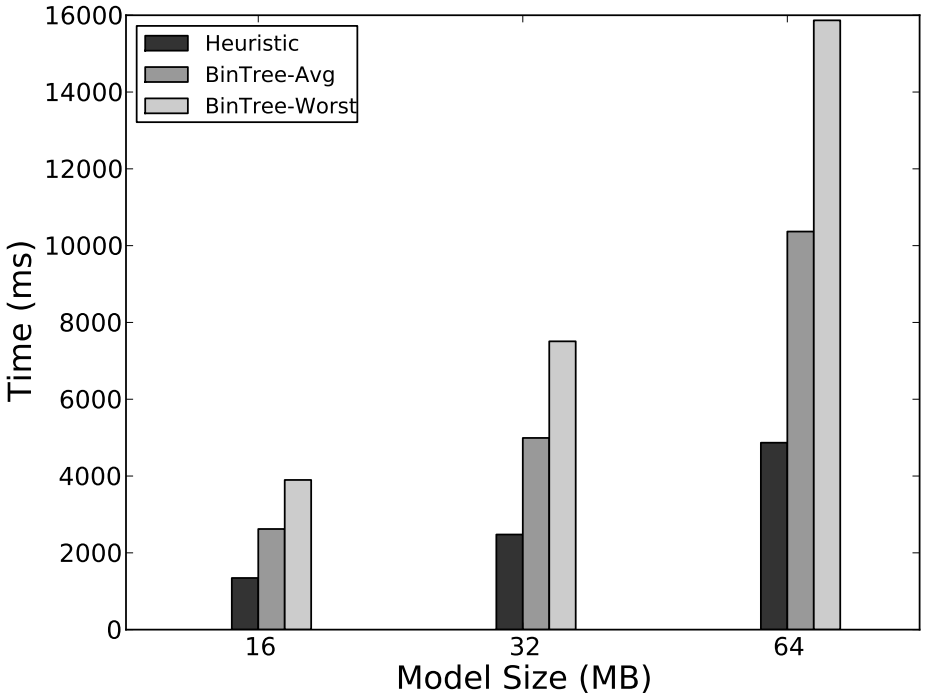

Figure 7.2.: Varying model size. In-bandwidth of a random node choked to 100Mbit/s

leaves of the spanning tree. On the other hand, a random binary tree, in its worst case, can place the choked learner in the interior of the tree thereby choking a significant portion of the pipeline. In this way, MWD achieves significantly better average synchronization times.

Impact of Varying Link Bandwidth

Figure 7.4 shows the average model sync times observed for a 32 MB model on 20 learner nodes, with different levels of in-bandwidth choking. It can be seen that as the choking increases, the improvement from our MWD approach increases as well (more than 15% improvement even for 400Mbit/s). This is due to the fact that MWD successfully localizes the lower bandwidth links to the lower levels of the tree.

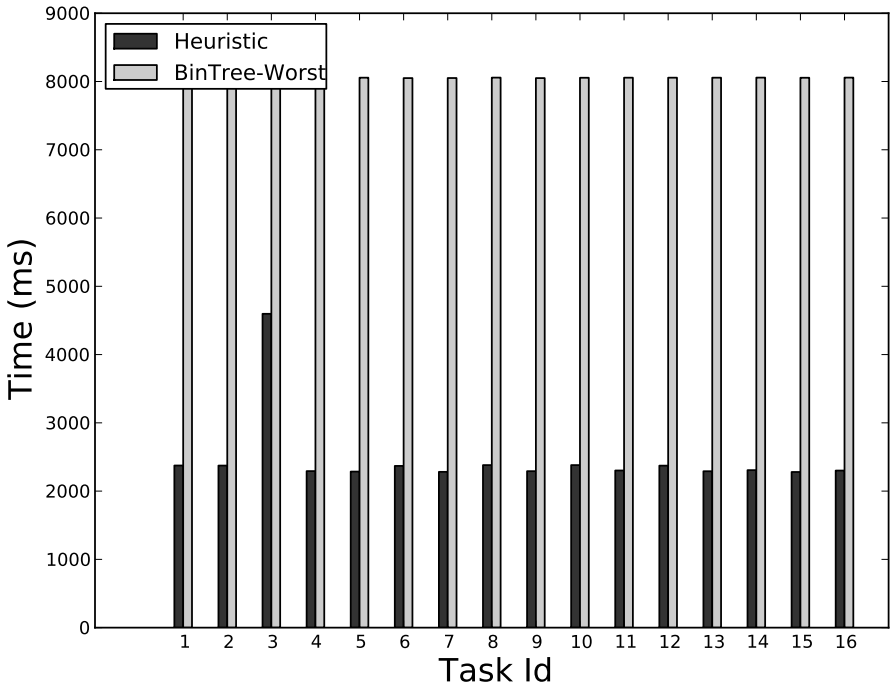

Figure 7.3.: Individual nodes' model sync times. In-bandwidth of a random node choked to 100Mbit/s

Impact of Complex Link Bandwidth Patterns

Figure 7.5, quantifies the effect of multiple choked links. The links are choked to the same magnitude of 200 Mb/s. The weight vector size in these experiments is 64MB. Increase in the number of choked links leads to increase in perfomance benefits of our MWD approach (more than 30%), when compared to the average-case binary tree. This can be explained as follows: as number of choked links increase, there are more chances of a random binary tree placing one of the choked nodes in the interior of the tree and thereby allowing the choked node to impact the overall pipeline throughput. However, note that the performance of the worst case binary tree, where all choked nodes are placed in the interior of the tree, does not vary substantially. This is because, our implementation divides the model into small parts and sends the parts as separate messages in a pipeline. Furthermore,

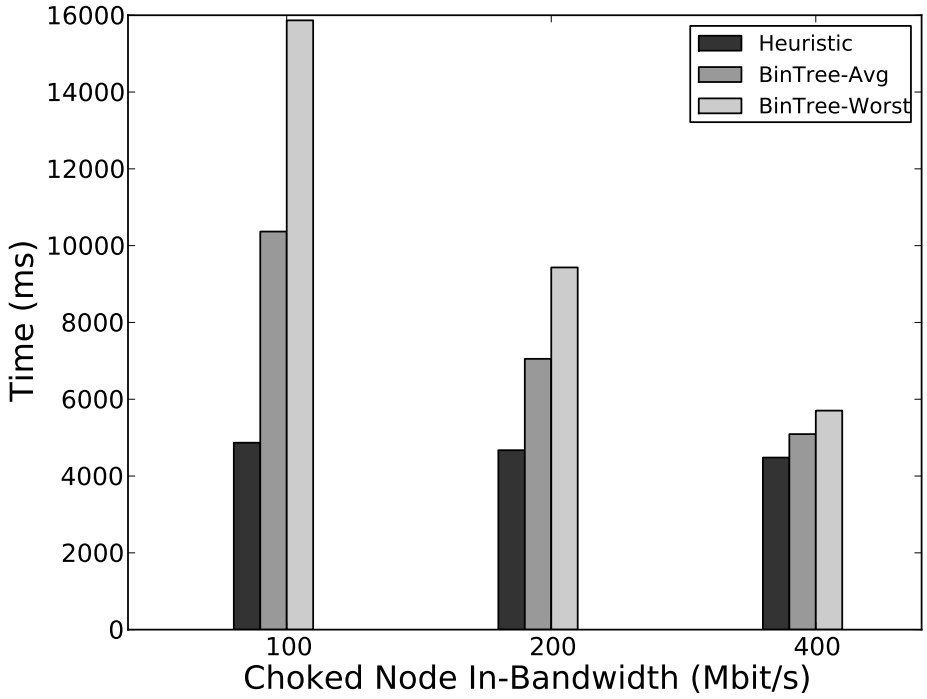

Figure 7.4.: Varying choked node bandwidth. Model size = 64MB. Num choked nodes = 1

the rate of the pipeline depends entirely on the slowest link, irrespective of the number of such slow links. However, if the model-size is small, the all-reduce implementation transmits the model as a single message, without any pipelining. In such cases, a random binary tree could place the choked nodes in different levels of the tree, leading to an accumulation of delays. In contrast, MWD places all the choked-nodes among the tree-leaves, ensuring that delays due to choked nodes are overlapped.

Figure 7.6 quantifies the average model sync time when nodes' in-bandwidths are sampled from two normal distributions: (i) mean is 500Mb/s and standard deviation is 200Mb/s; and (ii) mean is 700Mb/s and standard deviation is 300Mb/s. As evident from our results, when the standard deviation is high, link-bandwidths are dispersed, leading to increased scope for improving the topology. The difference in average sync time is more than 13% between the best and worst overlays for the case of large standard deviation.

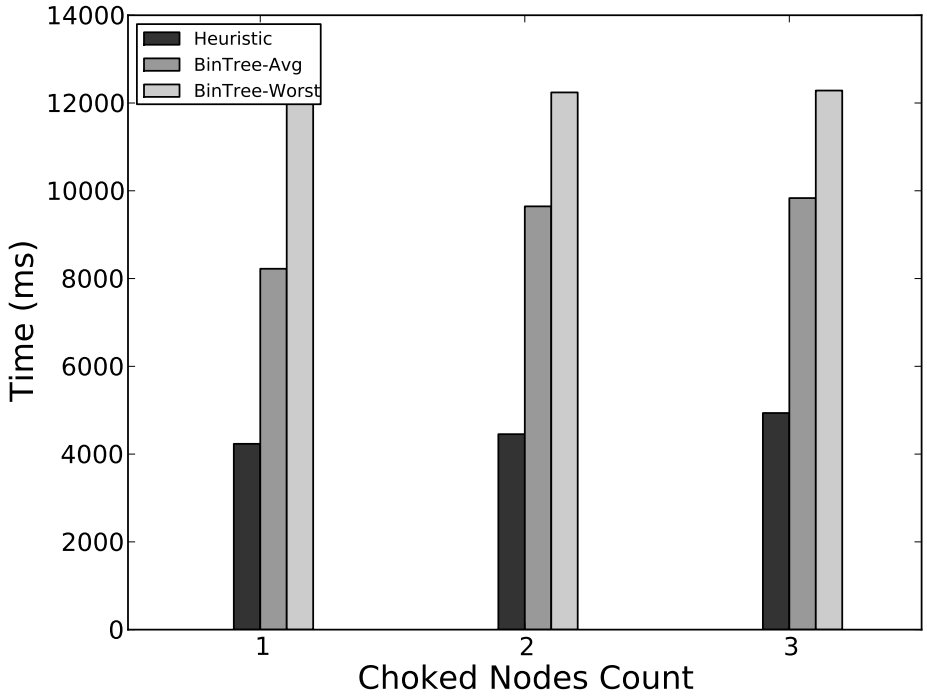

Figure 7.5.: Varying number of choked links. Model size = 64MB. Bandwidth choked to 200Mbit/s

7.5 Chapter Summary

Dynamic compute and network overheads can significantly impact the performance of streaming systems. In this chapter, we present efficient techniques for dynamic re-optimization of overlay topologies for group communication operations, through the use of a feedback-driven control loop. By abstracting the topology structure as versioned route-maps, the controller modifies overlays on-the-fly, enabling fast topology re-optimization with least system-disruption. All of the proposed techniques are implemented in a real system. we demonstrate significant improvement in performance (more than 15%) when the proposed MWD heuristic is used to generate pipelined all-reduce overlays for model synchronization. Given the importance of stream processing systems and the ubiquity of

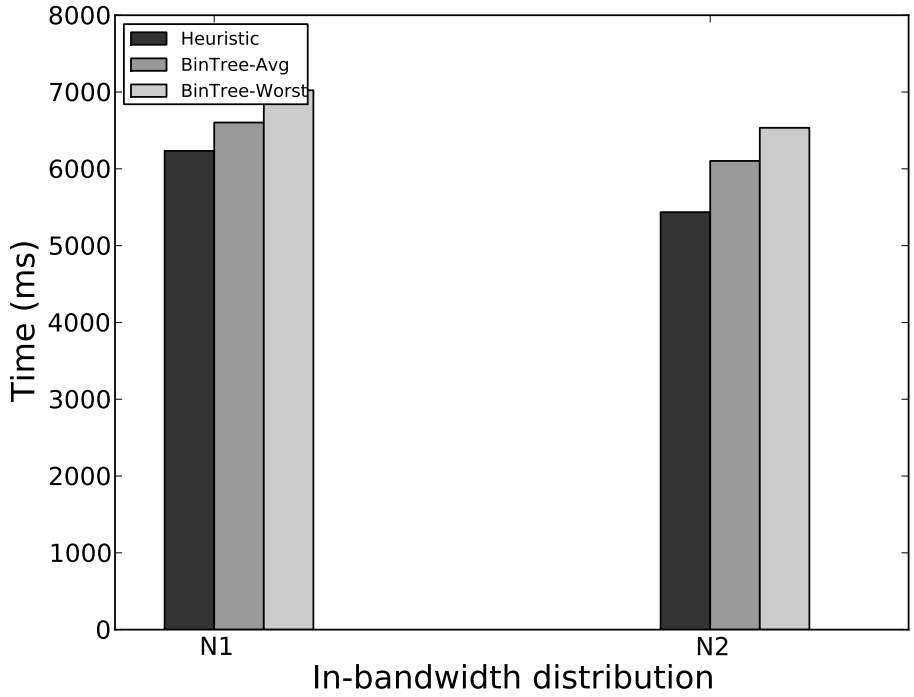

Figure 7.6.: Complex congestion pattern. Model size = 64MB. N1 = Normal (mean=500Mb/s, sd=200Mb/s), N2 = Normal (mean=700Mb/s, sd=300Mb/s)

dynamic network state in cloud environments, our results represent a significant and practical improvement.

8 CONCLUSIONS

In the context of large amounts of data generated by web-user activity and sensor-measurements, efficient storage and compute frameworks for real-time analysis pose significant challenges. This dissertation focuses on dynamic techniques for improving throughput of the pipeline, from collection and processing of real-time streaming data, to efficient storage and use of the resultant state. Our techniques fall into two categories: (i) dynamic optimization of stream-processing pipelines through fine-grained bottleneck detection and diagnosis; (ii) dynamic lock localization techniques to improve throughput of distributed transaction protocols. The techniques primarily focus on system-optimizations needed to tolerate resource-interference in multi-tenant, cloud deployments. This dissertation also presents programming models and runtime-optimizations in batch-processing systems, for applications to exploit potential asynchrony and amorphous data-parallelism.

In the context of batch-processing systems, we use MapReduce as a platform for distributed execution of asynchronous algorithms. We propose partial synchronization techniques to alleviate global synchronization overheads. We demonstrate that when combined with locality enhancing techniques and algorithmic asynchrony, these extensions are capable of yielding significant performance improvements. To increase the application scope of traditional data-parallel compute engines (such as MapReduce) and to enable applications exhibiting amorphous data-parallelism, we propose the `TransMR` framework. `TransMR` is an extension of the MapReduce programming model with constructs to support transactional execution of computation units. The proposed runtime uses transactions on shared-state (hosted by a distributed key-value store) to detect runtime conflicts between concurrent computation units. We list the fault-tolerance properties of the system and show that it enables many applications, hitherto infeasible in the conventional data-parallel frameworks.

In the context of object stores, this dissertation introduces techniques for improving throughput of distributed transactions. Current generation of middle-ware systems for distributed transactions rely on disjoint groups of objects (entity-groups) on which efficient local transactional support is provided using multi-version concurrency control. A lock-based protocol is used to support distributed transactions across entity-groups. A significant drawback of this scheme is that the latency of distributed transactions increases with the number of entity-groups it operates on. This is due to the commit overhead of local transactions, and network overhead due to distributed locks. We address this problem using lock-localization – locks for distributed objects are dynamically migrated and placed in distinct entity-groups in the same data-store. This reduces the overhead of multiple local transactions while acquiring locks. Application-oriented clustering of locks in these new entity-groups leads to a decrease in network overhead. Separating locks from data in this manner, however, affects the latency of local transactions. To account for this, we propose protocols and policies for selective, adaptive, and dynamic migration of locks. Using TPC-C benchmark, we provide detailed evaluation of the system, validating its superior performance.

In the context of stream processing systems, we show that a single bottleneck in the pipeline (congested link or an overloaded operator) can drastically impact the system throughput. We present a number of techniques for addressing bottlenecks in stream engines through the use of a feedback-driven control loop. Our techniques fall into two major classes – network-aware routing for fine grained control of streams; and dynamic overlay generation for optimizing performance of group communication operations. Network-aware routing is useful in shaping stream-traffic based on the observed network/compute resources available along topology paths. Complex group communication operations such as all-reduce are used to synchronize large-state among operators. We show that optimization of cross-DAG overlays in a streaming model requires a cost function that is markedly different from ones used in literature for conventional messaging systems. To enable fast workflow re-optimization with least system-disruption, we present a light-weight protocol for consistent modification of pipelines. We present detailed algorithms, their implementa-

tion in a real system, and address issues of fault tolerance and performance. We show that our performance improvements are robust to dynamic changes, as well as complex congestion patterns. Given the widespread use of streaming systems and the need for dealing with dynamic system state (as observed in cloud environments), our techniques represent a significant and practical improvement.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 2008.

[2] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2010.

[3] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 1990.

[4] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. In *Proceedings of the International Conference on Very Large Data Bases*, 2012.

[5] Padhraic Smyth, Max Welling, and Arthur U. Asuncion. Asynchronous distributed learning of topic models. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, 2009.

[6] Joseph E. Gonzalez, Yucheng Low, Carlos Guestrin, and David O'Hallaron. Distributed parallel inference on large factor graphs. In *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence*, UAI '09, 2009.

[7] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *Proceedings of the Conference on Innovative Data System Research*, 2005.

[8] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the ACM european conference on Computer Systems*, 2013.

[9] Apache Samza: Distributed stream processing framework. `samza.incubator.apache.org`.

[10] Storm: Distributed realtime computation system. `storm.apache.org`.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 2008.

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[13] Hakan Hacigumus, Junichi Tatemura, Wang-Pin Hsiung, Hyun J. Moon, Oliver Po, Arsany Sawires, Yun Chi, and Hojjat Jafarpour. Cloud-DB: One size fits all revived. *IEEE Congress on Services Computing*, 2010.

[14] Junichi Tatemura, Oliver Po, and Hakan Hacgümüş. Microsharding: A declarative approach to support elastic OLTP workloads. *SIGOPS Operating System Review*, 2012.

[15] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data System Research*, 2011.

[16] Hbase: Hadoop database. `http://hadoop.apache.org/hbase`.

[17] Daniel Shawcross Wilkerson, Simon Fredrick Vicente Goldsmith, Ryan Barrett, Erick Armbrust, Robert Johnson, and Alfred Fuller. Distributed transactions for google app engine: Optimistic distributed transactions built upon local multi-version concurrency control. `http://arxiv.org/html/1106.3325v1`.

[18] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[19] Lixia Liu and Zhiyuan Li. Improving parallelism and locality with asynchronous algorithms. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[20] Karthik Kambatla, Abhinav Pathak, and Himabindu Pucha. Towards optimizing hadoop provisioning for the cloud. *1st Workshop on Hot Topics in Cloud Computing, HotCloud*, 2009.

[21] Thomas Sandholm and Kevin Lai. Mapreduce optimization using dynamic regulated prioritization. In *SIGMETRICS/Performance '09: Proceedings of the 2009 Joint International Conference on Measurement & Modeling of Computer Systems*, 2009.

[22] Matei Zaharia, Andrew Konwinski, Anthony Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2008.

[23] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor system. In *Proceedings of the 13th Intl. Symposium on High-Performance Computer Architecture (HPCA)*, 2007.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 2008.

[25] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[26] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 2010.

[27] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[28] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, 2010.

[29] M. Isard, M. Budiu, Y. Yun, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. *ACM SIGOPS EuroSys*, 2007.

[30] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, 2011.

[31] Justin Levandoski, David Lomet, Mohamed F. Mokbel, and Kevin Keliang Zhao. Deuteronomy: Transaction support for cloud data. In *Proceedings of the Conference on Innovative Database Systems Research*, 2011.

[32] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems*, 2013.

[33] David B. Lomet, Alan Fekete, Gerhard Weikum, and Michael J. Zwilling. Unbundling transaction services in the cloud. In *Proceedings of the Conference on Innovative Database Systems Research*, 2009.

[34] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. Cloudtps: Scalable transactions for web applications in the cloud. *IEEE Transactions on Services Computing*, 2011.

[35] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: A scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, 2010.

[36] F. Junqueira, B. Reed, and M. Yabandeh. Lock-free transactional support for large-scale storage systems. In *Dependable Systems and Networks Workshops (DSN-W)*, 2011.

[37] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2012.

[38] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual technical conference*, 2010.

[39] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. In *Proceedings of the International Conference on Very Large Data Bases*, 2003.

[40] Rohit Wagle, Henrique Andrade, Kirsten Hildrum, Chitra Venkatramani, and Michael Spicer. Distributed middleware reliability and fault tolerance support in System-S. In *Proceedings of the 5th ACM International Conference on Distributed Event-based System*, DEBS '11, 2011.

[41] S4: Distributed stream computing platform. `incubator.apache.org/s4/`.

[42] YongChul Kwon, Magdalena Balazinska, and Albert Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. In *Proceedings of the International Conference on Very Large Data Bases*, 2008.

[43] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2013.

[44] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Re-optimizing data-parallel computing. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, NSDI, 2012.

[45] Rohit Khandekar, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Joel Wolf, Kun-Lung Wu, Henrique Andrade, and Buğra Gedik. COLA: Optimizing stream processing applications via graph partitioning. In *Middleware*, 2009.

[46] Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware*, 2008.

[47] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 24*, 2011.

[48] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. Towards a unified architecture for in-RDBMS analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2012.

[49] Hadoop: Open-source software for reliable, scalable, distributed computing. `http://hadoop.apache.org`.

[50] Karthik Kambatla, Naresh Rapolu, Suresh Jagannathan, and Ananth Grama. Relaxed synchronization and eager scheduling in mapreduce. Technical report, Purdue University Technical Report CSD TR 09-010, 2009.

[51] Price D. J. de S. A general theory of bibliometric and other cumulative advantage processes. *Journal of the American Society for Information Science*, 1976.

[52] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. *Advances in Neural Information Processing Systems 19*, 2007.

[53] Elad Yom-tov and Noam Slonim. Parallel pairwise clustering. In *SDM'09, Proceedings of SIAM Data Mining Conference*, 2009.

[54] Apache mahout: Scalable machine learning and data mining. `http://lucene.apache.org/mahout`.

[55] US Census Data, 1990. UCI machine learning repository. http://kdd.ics.uci.edu/databases/census1990/USCensus1990.html.

[56] Richard O. Duda, Peter E. Hart, and David G. Stork. Chapter 8. Pattern Classication. 2nd edition. *A Wiley-Interscience Publication*, 2001.

[57] R.E. Bryant. Data-intensive supercomputing: The case for DISC. Technical report cmu-cs-07-128, Carnegie Mellon University. 2007.

[58] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, 2011.

[59] Hbase: Hadoop database. A scalable big data store. http://hadoop.apache.org/hbase.

[60] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 1981.

[61] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, 2006.

[62] Eric A. Brewer. Towards robust distributed systems (abstract). *Principles of Distributed Computing*, 2000.

[63] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. *International Symposium on Workload Characterization*, 2008.

[64] Trinity: General purpose distributed graph system over a memory cloud. http://research.microsoft.com/en-us/projects/trinity.

[65] HyperGraphDB: A graph database. http://www.hypergraphdb.org/index.

[66] Titan: Scalable graph database. http://thinkaurelius.github.com/titan/.

[67] TPC-C benchmark. www.tpc.org/tpcc.

[68] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: A workload-driven approach to database replication and partitioning. In *Proceedings of the International Conference on Very Large Data Bases*, 2010.

[69] Metis: Graph partitioning and fill-reducing matrix ordering. http://glaros.dtc.umn.edu/gkhome/views/metis.

[70] Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, 2010.

[71] Lon Bottou and Yann LeCun. Large scale online learning. In *Advances in Neural Information Processing Systems*, 2003.

[72] Lon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In *Proceedings on Advances in Neural Information Processing Systems*, 2008.

[73] Matthew D. Hoffman, David M. Blei, and Francis Bach. Online learning for latent Dirichlet allocation. In *Advances in Neural Information Processing Systems*, 2010.

[74] Wang Lam, Lu Liu, Sts Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. Muppet: Mapreduce-style processing of fast data. In *Proceedings of the International Conference on Very Large Data Bases*, 2012.

[75] Badrish Chandramouli, Jonathan Goldstein, and Songyun Duan. Temporal analytics on big data for web advertising. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, 2012.

[76] Mesos: A distributed system kernel. `mesos.apache.org`.

[77] Krister Jacobsson, Lachlan L. H. Andrew, Karl H. Johansson, Steven H. Low, and et al. Ack-clocking dynamics: Modelling the interaction between windows and the network. In *Proceedings of the ACM INFOCOMM 2008 Conference*, 2008.

[78] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, STOC '97, 1997.

[79] Joel Nishimura and Johan Ugander. Restreaming graph partitioning: Simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, 2013.

[80] IPerf: A network bandwidth measurement tool. `https://github.com/esnet/iperf`.

[81] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. Identifying suspicious URLs: An application of large-scale online learning. In *International Conference on Machine Learning*, 2009.

[82] Zbigniew Jerzak and Holger Ziekow. The DEBS 2014 grand challenge. In *Distributed Event Based Systems*, DEBS, 2014.

[83] Alekh Agarwal, Oliveier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *Journal of Machine Learning Research*, 2014.

[84] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 2012.

[85] O. Beaumont, L. Marchal, and Y. Robert. Broadcast trees for heterogeneous platforms. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '05, 2005.

[86] Vowpal Wabbit: A fast out-of-core learning system. `https://github.com/JohnLangford/vowpalwabbit`.

[87] Kurt Thomas, Chris Grier, Justin Ma, Vern Paxson, and Dawn Song. Design and evaluation of a real-time url spam filtering service. In *IEEE Symposium on Security and Privacy*, SP, 2011.

VITA

VITA

Naresh Kumar Reddy Rapolu was born and brought up in Hyderabad, India. He obtained his B.Tech. in computer science and engineering from the Indian Institute of Technology (IIT), Roorkee, India. He obtained his M.S. and Ph.D. from the Department of Computer Science at Purdue University in May 2011 and May 2015, respectively. His research interests include distributed systems, concurrent programming and machine learning. During his Ph.D. studies, he interned with Google, Mountain View, California (2010) and NEC Laboratories, Princeton, New Jersey (2012 and 2013).