13th ANNUAL WORKSHOP 2017

ACCELERATING APACHE SPARK WITH RDMA

Yuval Degani, Sr. Manager, Big Data and Machine Learning

Mellanox Technologies

March 28th, 2017

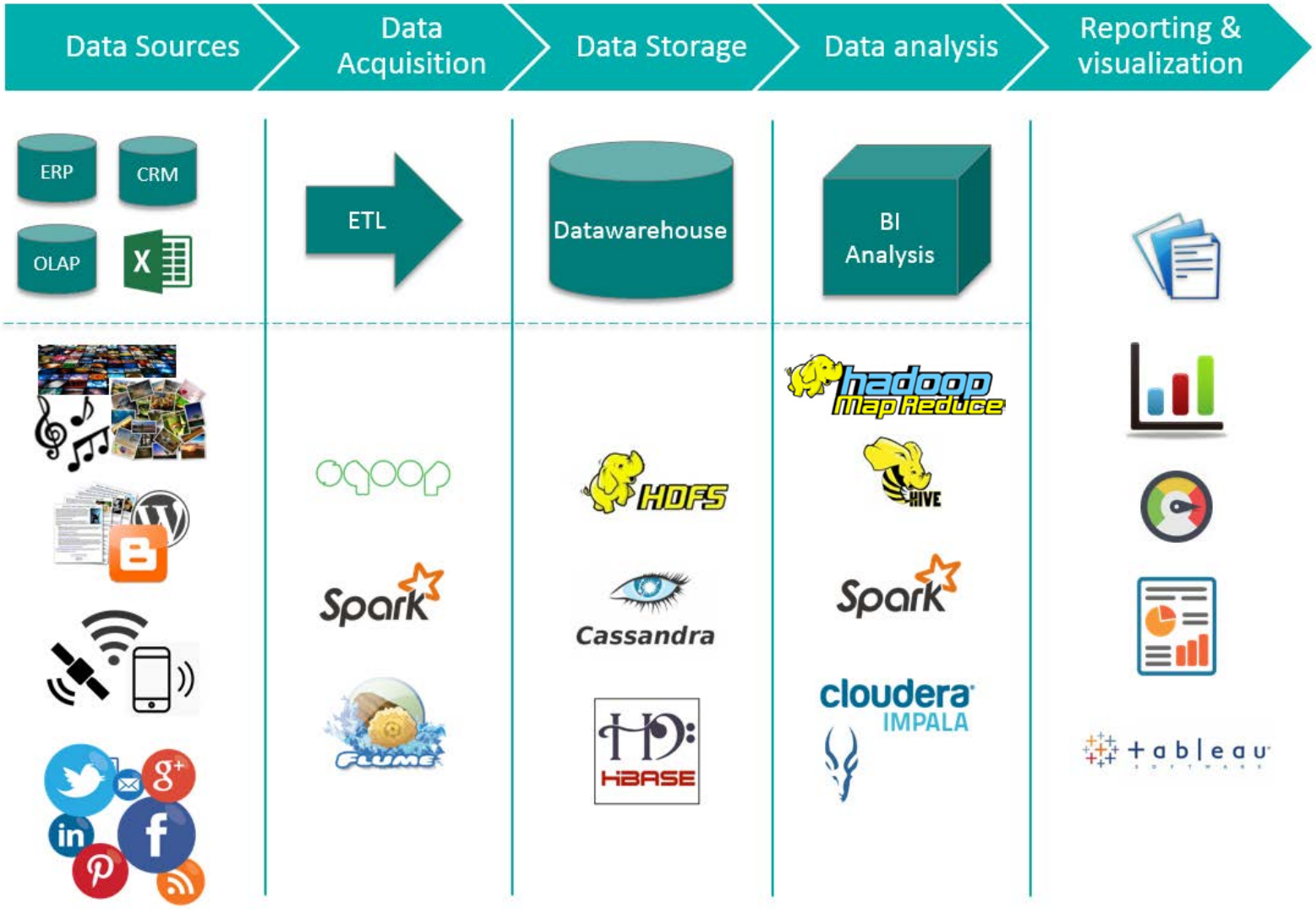
AGENDA

- **Apache Spark 101**
- **The Potential in Accelerating Spark Shuffle**
- **Accelerating Spark Shuffle with RDMA – Deep Dive**
- **Results**
- **Roadmap**

APACHE SPARK 101

APACHE SPARK 101

Spark within the Big Data ecosystem



APACHE SPARK 101

Quick facts

- **In a nutshell: Spark is a data analysis platform with implicit data parallelism and fault-tolerance**

- **Initial release: May, 2014**
- **Originally developed at UC Berkeley's AMPLab**
- **Donated as open source to the Apache Software Foundation**
- **Most active Apache open source project**

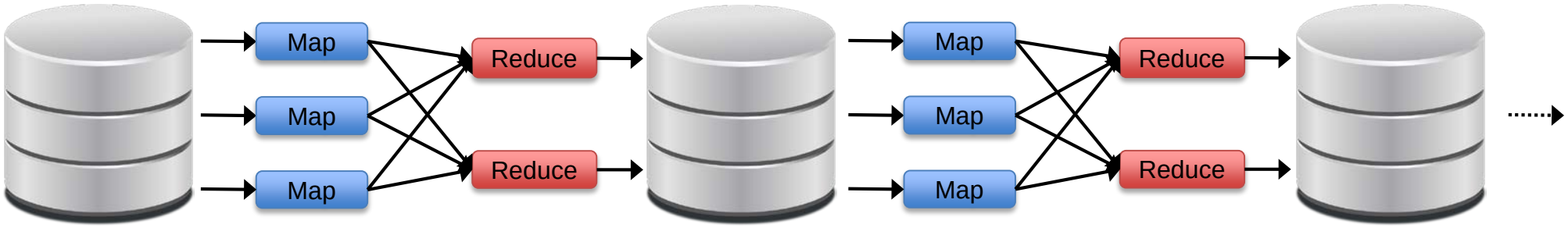
- **50% of production systems are in public clouds**

- **Notable users:**

APACHE SPARK 101

But why?

- **The traditional Map-Reduce model is bound to an acyclic data flow from stable storage to stable storage (e.g. HDFS)**

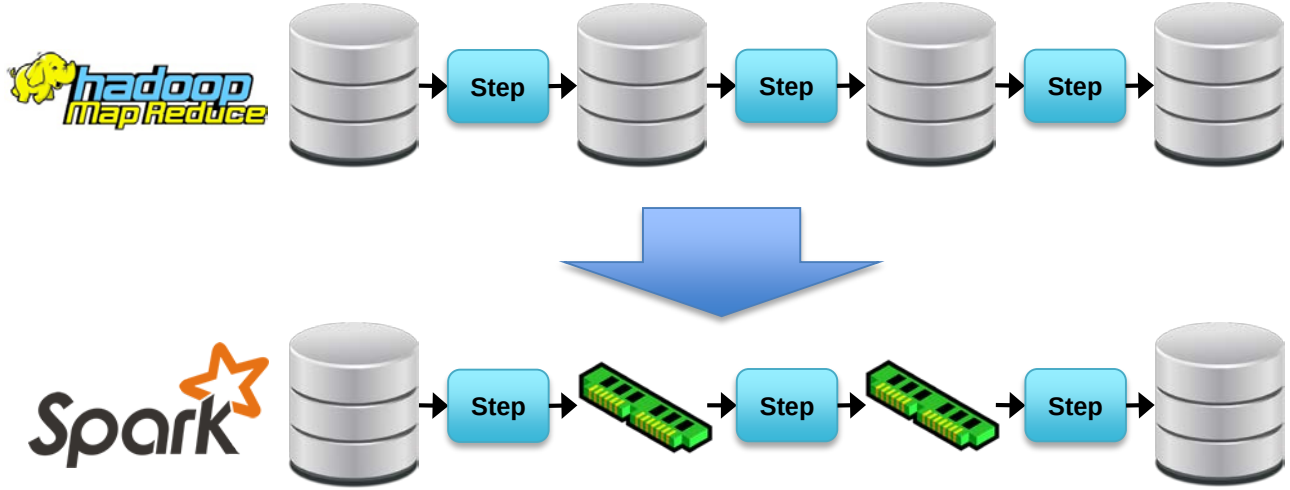


APACHE SPARK 101

But why?

- **<u>Some</u> applications frequently reuse data, and don't just digest it once:**
  - Iterative algorithms (e.g. machine learning, graphs)
  - Interactive data-mining and streaming
- **The acyclic data flow model becomes very inefficient when data gets repeatedly reused**

- **The solution: Resilient Distributed Datasets (RDDs)**
  - In-memory data representation
  - The heart of Apache Spark
  - Preserves and enhances the appealing properties of Map-Reduce:
    - Fault tolerance
    - Data locality
    - Scalability



APACHE SPARK 101

"Everyday I'm Shuffling"

- RDDs let us keep data within quick reach, in-memory
- But, cluster computing is all about moving data around the network!
- Data transfer e call "Shuffle"

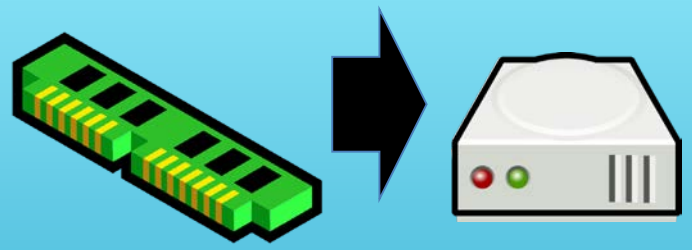
- So, we have r fit!

APACHE SPARK 101

Shuffle Basics

Shuffling is the process of redistributing data across partitions (AKA repartitioning)

1 Shuffle Write

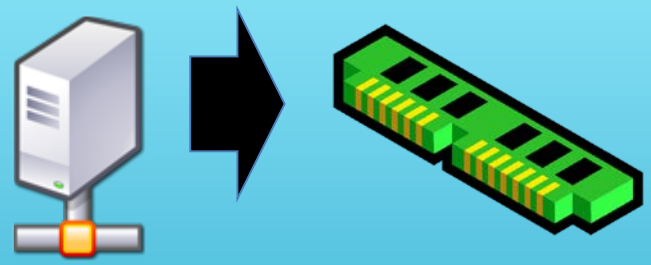


Worker nodes write their intermediate data blocks (Map output) into local storage, and list the blocks by partition

2 Broadcast block locations

The master node broadcasts a combined list of blocks, grouped by partition

3 Shuffle Read



Each reduce partition fetches all of the blocks listed under its partition – from various nodes in the cluster

THE POTENTIAL IN ACCELERATING SHUFFLE

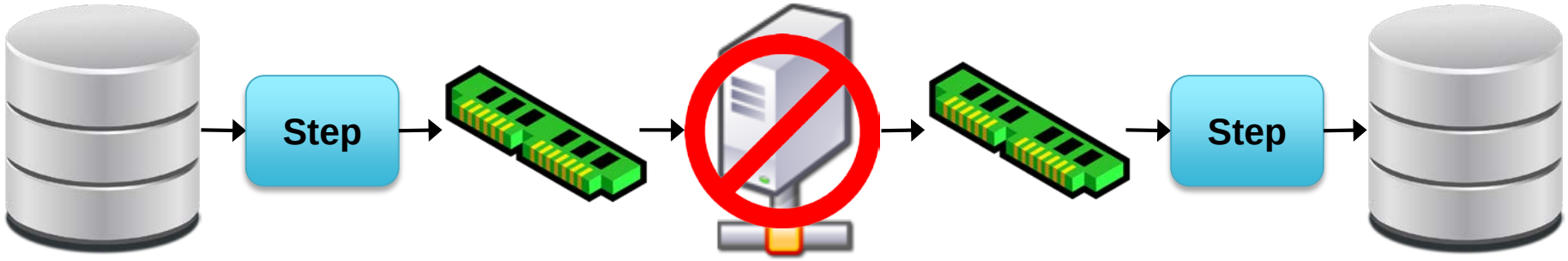
THE POTENTIAL IN ACCELERATING SHUFFLE

What's the opportunity here?

- **Before setting off on the journey of adding RDMA to Spark, it was essential to estimate the ROI of such work**

- **What's better than a controlled experiment?**

Goal	Quantify the potential performance gain in accelerating network transfers
Method	Bypass the network in Spark Shuffle and compare to the original: No network = maximum performance gain



THE POTENTIAL IN ACCELERATING SHUFFLE

Bypassing the network in Spark Shuffle

- **General idea:**
  - Do not fetch blocks from the network, just reuse a local sampled block over and over
  - No copying
  - No message passing what so ever (for Shuffle)
  - Everything else stays exactly the same! (no optimizations in Shuffle)
  - Compare to TCP

Phase	TCP Shuffle	Network bypass Shuffle
Worker node initialization	Read sampled shuffle block from disk for reuse in Shuffle	
On block fetch request	Send request and wait for data to come in to trigger completion	Immediately complete the block fetch request
On block fetch request completion	Drop incoming shuffle data block	No op
Deliver shuffle block	Duplicate (shallow copy) the sampled buffer from initialization and publish (trim to desired size)	

THE POTENTIAL IN ACCELERATING SHUFFLE

Testbed

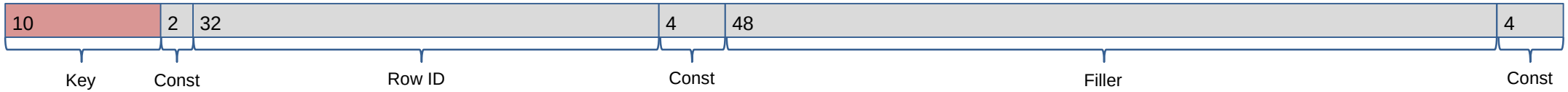
- **Benchmark:**
  - HiBench TeraSort
  - Workload: 600GB
- **Testbed:**
  - HDFS on Hadoop 2.6.0
    - No replication
  - Spark 2.0.0
    - 1 Master
    - 30 Workers
    - 28 active Spark cores on each node, 840 total
  - Node info:
    - Intel Xeon E5-2697 v3 @ 2.60GHz
    - 256GB RAM, 128GB of it reserved as RAMDISK
    - RAMDISK is used for Spark local directories and HDFS

THE POTENTIAL IN ACCELERATING SHUFFLE

TeraSort

- **TeraSort:**
  - Read unsorted data from HDFS
  - Sort the data
  - Write back sorted data to HDFS

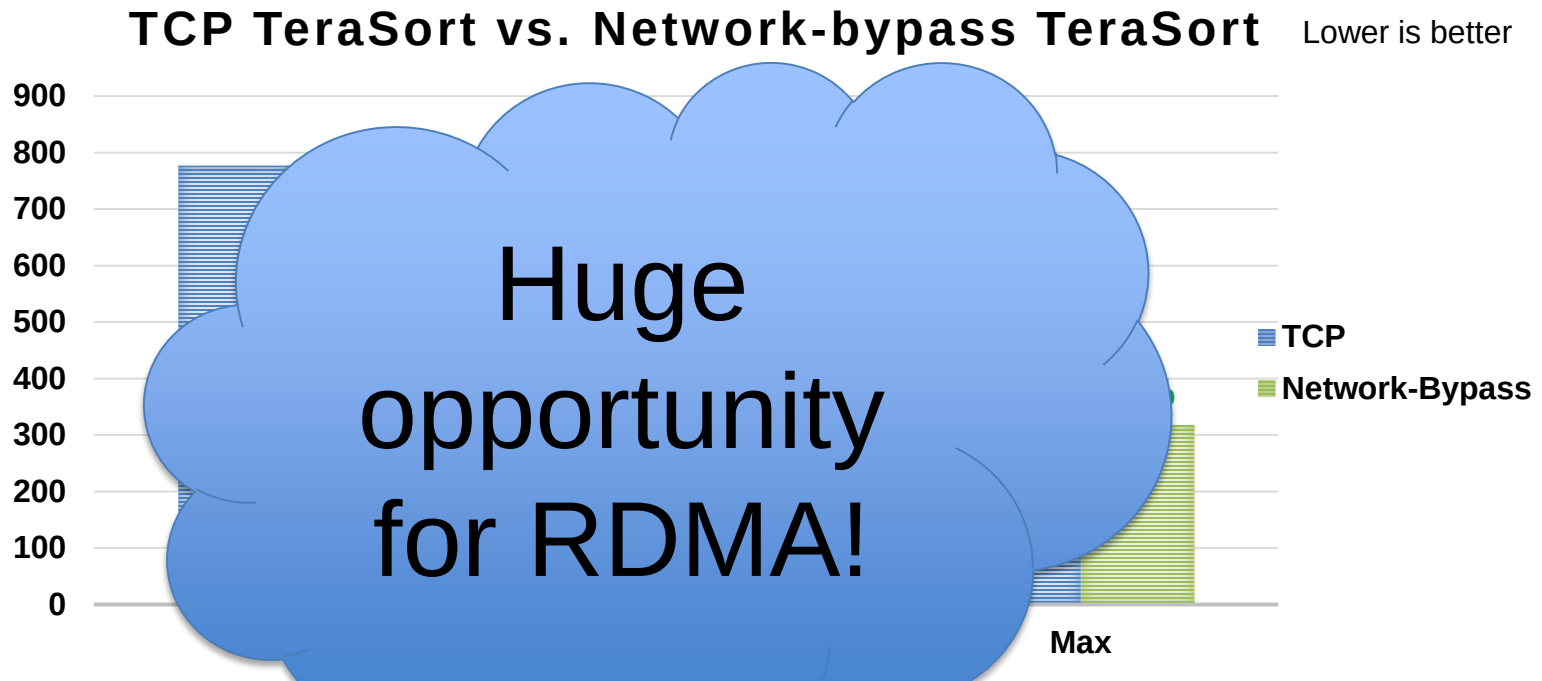
- **Workload:**
  - A collection of K-V pairs
  - Keys are random



- **Algorithm at high-level (all phases are distributed among all cores):**
  - Define the partitioner:
    - Read random samples from HDFS and define balanced partitions of key ranges
  - Partition (map) the data:
    - Read data from HDFS, and group by partition, according to the partitions defined in #1 (Shuffle Write)
  - Retrieve partitioned data (Shuffle Read)
    - Sort the K-V pairs by key
    - Write the sorted data to HDFS

THE POTENTIAL IN ACCELERATING SHUFFLE

Results



Runtime samples	Nor…	…k bypass	Opportunity
Average	777 seconds	356 seconds	54%
Max	815 seconds	389 seconds	52%
Min	755 seconds	318 seconds	58%

ACCELERATING SHUFFLE WITH RDMA

ACCELERATING SHUFFLE WITH RDMA

Challenges

Challenges in adding RDMA to Spark Shuffle

- **Spark is written in Java & Scala**
  - What RDMA API to use?
  - Java's Garbage Collection is a pain for managing low-level buffers
- **RDMA connection establishment – how can we make connections as long lived as possible?**
- **Spark's Shuffle Write data is currently saved on the local disk. How can we make the data available for RDMA?**
- **Must keep changes to Spark code to a minimum**
  - Spark is not very plug-in-able
  - Spark keeps changing rapidly – API changes, implementation changes
  - Maintain long term functionality

ACCELERATING SHUFFLE WITH RDMA

RDMA Facilities – Design Notes

- **RDMA API is available in Java through:**
  - JXIO-AccelIO – Abstract of RDMA through RPC (https://github.com/accelio/JXIO)
  - DiSNi – Open-source release of IBM's JVerbs – ibverbs like interface in Java (https://github.com/zrlio/disni)
    - We have decided to go with DiSNi since it gives us more flexibility
- **RDMA buffer management**
  - Centralized pool of buffers per worker node
  - Pool consists of multiple stacks of different sizes (in powers of 2)
  - Buffers are allocated off-heap and registered on demand
  - When they are no longer in use – they are recycled back to the pool
  - Survive for the length of a job (minutes)
- **RDMA connection establishment**
  - For simplicity, couple each TCP connection with a corresponding RDMA connection
  - RDMA is used for all block data transfers
  - Survive for the length of a job (minutes)

ACCELERATING SHUFFLE WITH RDMA

Shuffle Write – Design Notes

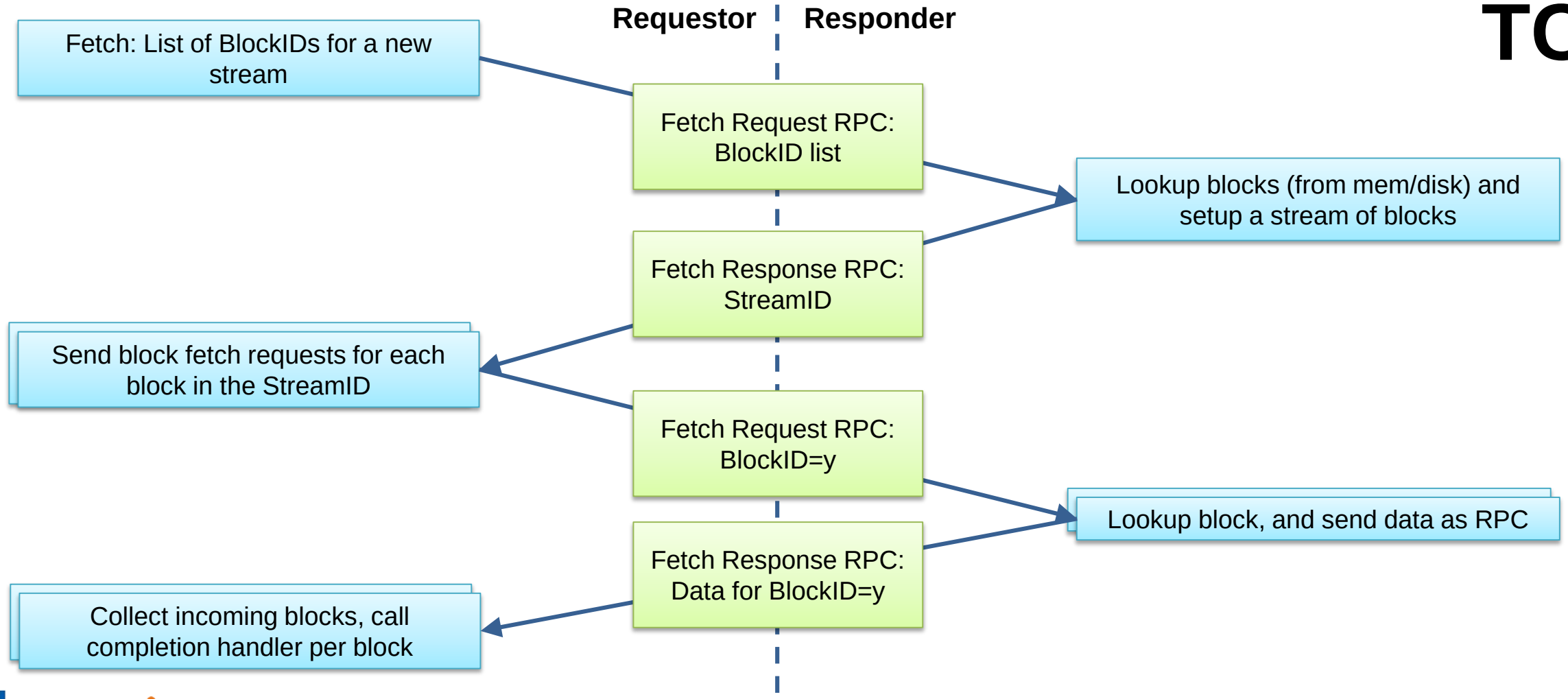
- **The challenge – get the Shuffle data in memory, and register as a Memory Region**

- **We provide two modes:**

Method	In-memory	File-mapping
Description	Instead of saving Shuffle data to disk, save it directly into RDMA buffers	Map Spark's file on local disk to memory, and register as a Memory Region
Pros	• Faster and more consistent than file-mapping unless low on system memory • Less overhead for RDMA • Simpler indexing (buffer-per-block)	• Minimal changes in Spark code • Files are already in buffer-cache – mmap is not expensive • Comparable memory footprint to TCP • No functional limitations
Cons	• High memory footprint • Many changes in Spark's ShuffleWriters and ShuffleSorters • Limited functionality (Spills, low memory)	• mmap() occasionally demonstrates latency jitter • Indexing – multiple blocks per file as in TCP

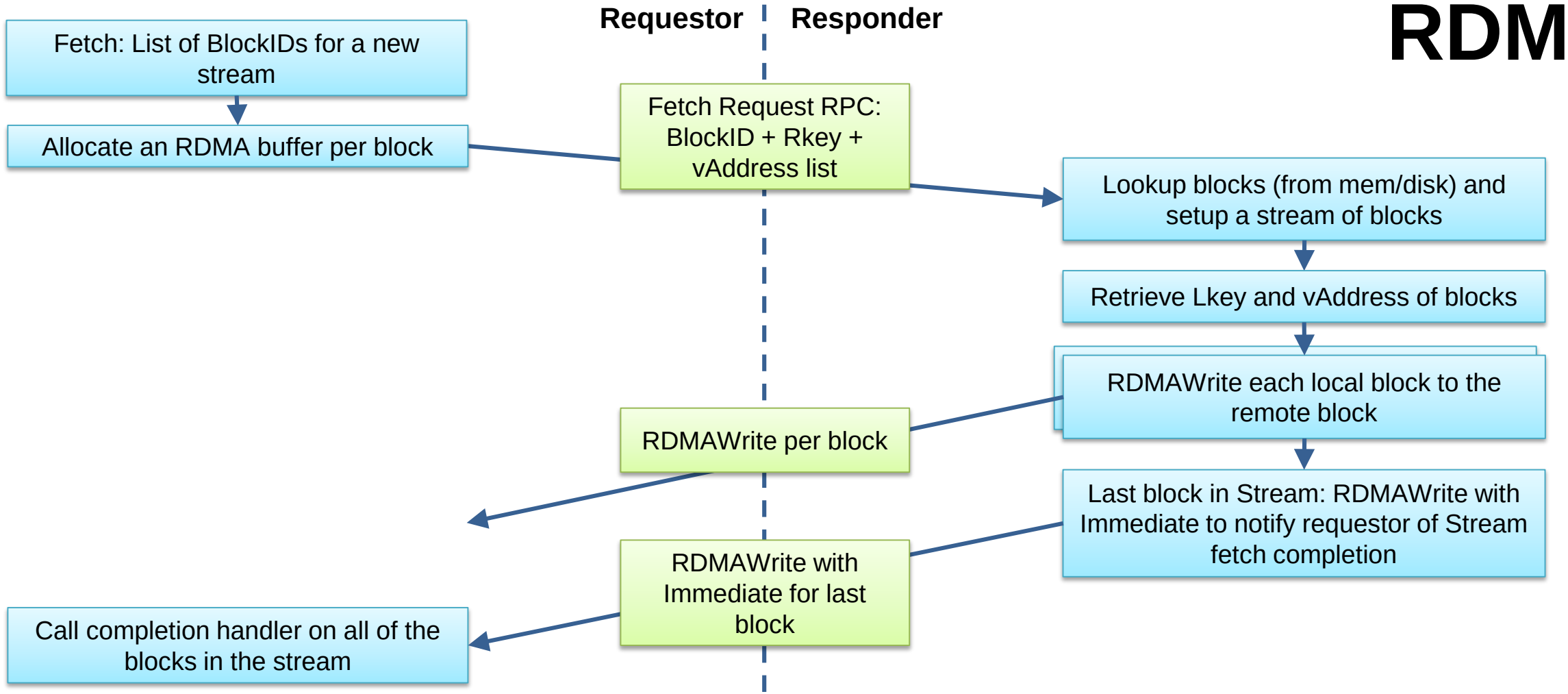
ACCELERATING SHUFFLE WITH RDMA

Shuffle Read Protocol in Spark



ACCELERATING SHUFFLE WITH RDMA

Shuffle Read Protocol in Spark



ACCELERATING SHUFFLE WITH RDMA

Shuffle Read: TCP vs. RDMA

- **N = Number of blocks**

	TCP	RDMA
Number of messages per fetch	2(N+1)	N+1
Number of copy operations per block	Responder: 1 to 2 times Requestor: 1 time Total: 2 to 3 times	0-copy

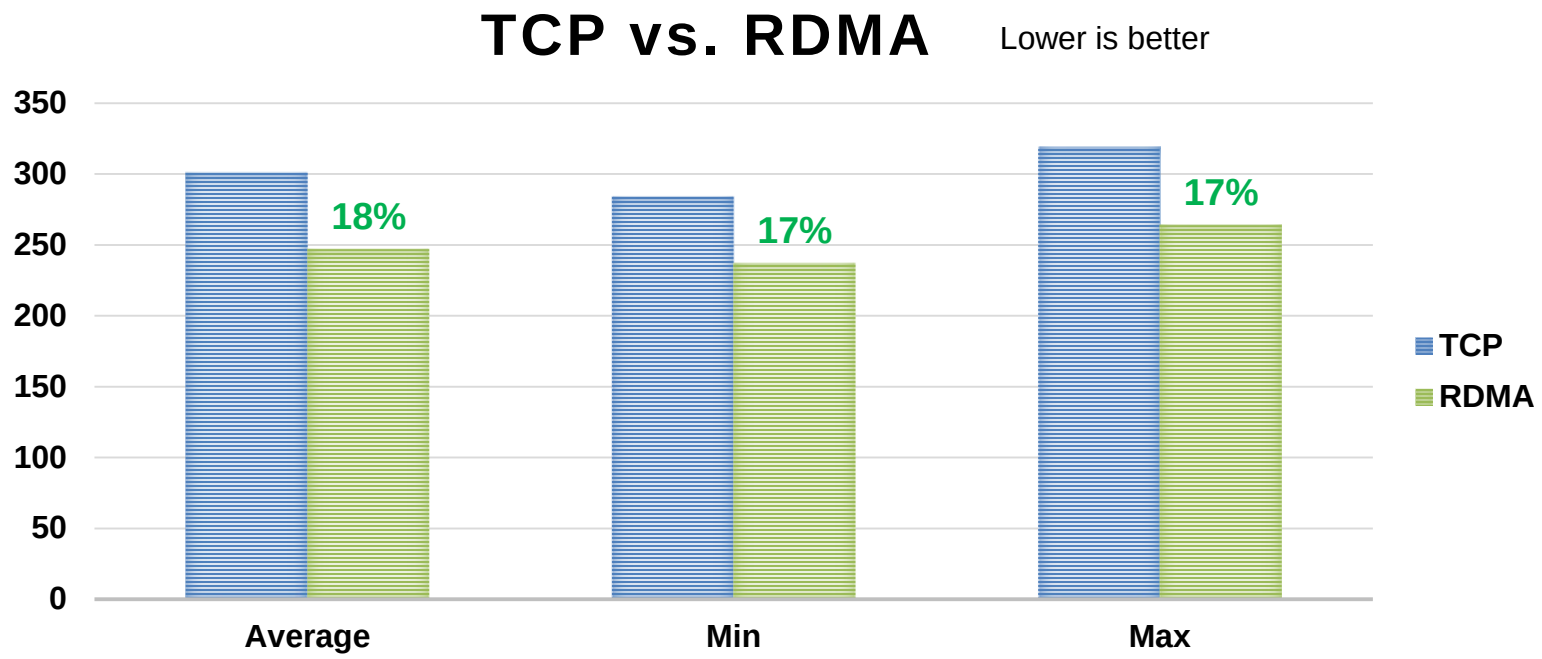
RESULTS

RESULTS

In-house TeraSort Results

Testbed:

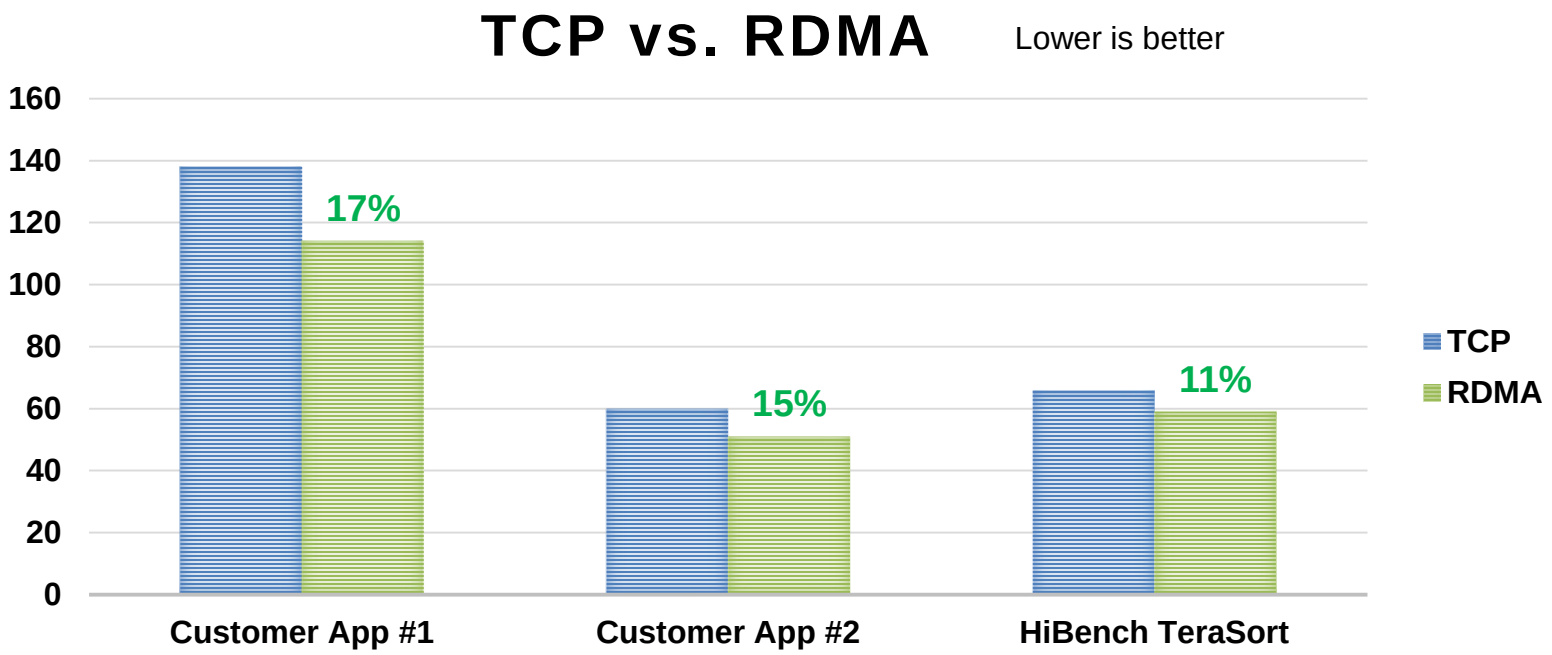
- HiBench TeraSort
  - Workload: 300GB
- HDFS on Hadoop 2.6.0
  - No replication
- Spark 2.0.0
  - 1 Master
  - 14 Workers
  - 28 active Spark cores on each node, 392 total
- Node info:
  - Intel Xeon E5-2697 v3 @ 2.60GHz
  - RoCE 100GbE
  - 256GB RAM
  - HDD is used for Spark local directories and HDFS



Runtime samples	TCP	RDMA	Improvement
Average	301 seconds	247 seconds	18%
Max	284 seconds	237 seconds	17%
Min	319 seconds	264 seconds	17%

RESULTS

Real-life Applications - Initial Results



Runtime samples	TCP	RDMA	Improvement	Input Size	Nodes	Cores per node	RAM per node
Customer App #1	138 seconds	114 seconds	17%	5GB	14	24	85GB
Customer App #2	60 seconds	51 seconds	15%	270GB	14	24	85GB
HiBench TeraSort	66 seconds	59 seconds	11%	100GB	16	24	85GB

ROADMAP

What's next?

- **Spark RDMA in GA expected in 2017**
- **Open-source Spark plugin**
- **Push to Spark upstream**

13th ANNUAL WORKSHOP 2017

THANK YOU

Yuval Degani, Sr. Manager, Big Data and Machine Learning

Mellanox Technologies