NativeTask

100MB/s Map Task

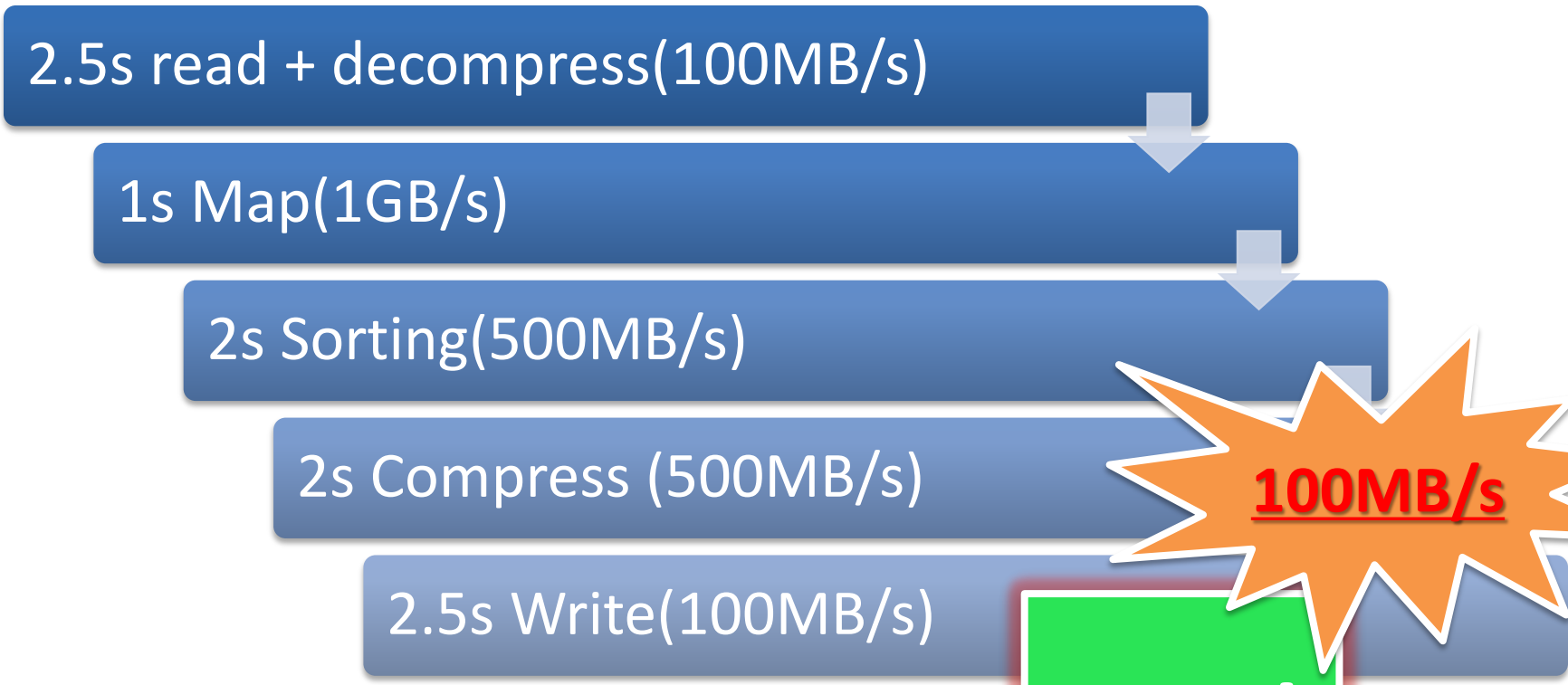
Sean Zhong xiang.zhong@intel.com
Wang, Huafeng huafeng.wang@intel.com
Zhang, Tianlun tianlun.zhang@intel.com

Agenda

- Motive
- What is native task?
  - Native task design principals
  - Native-Task's current status
- Native-Task's performance
- Native Runtime - General execution framework
- Proposals and Suggestions

How fast can map task be in theory?

1GB data, (250MB snappy ratio: **4/1**)



How fast is map task in reality?

1GB data, (512MB snappy ratio: **2/1**)

The snappy ratio is controlled by Hi-Bench

Hi-Bench WordCount(No Combiner)

7s read + decompress

30s Map

101s Sorting

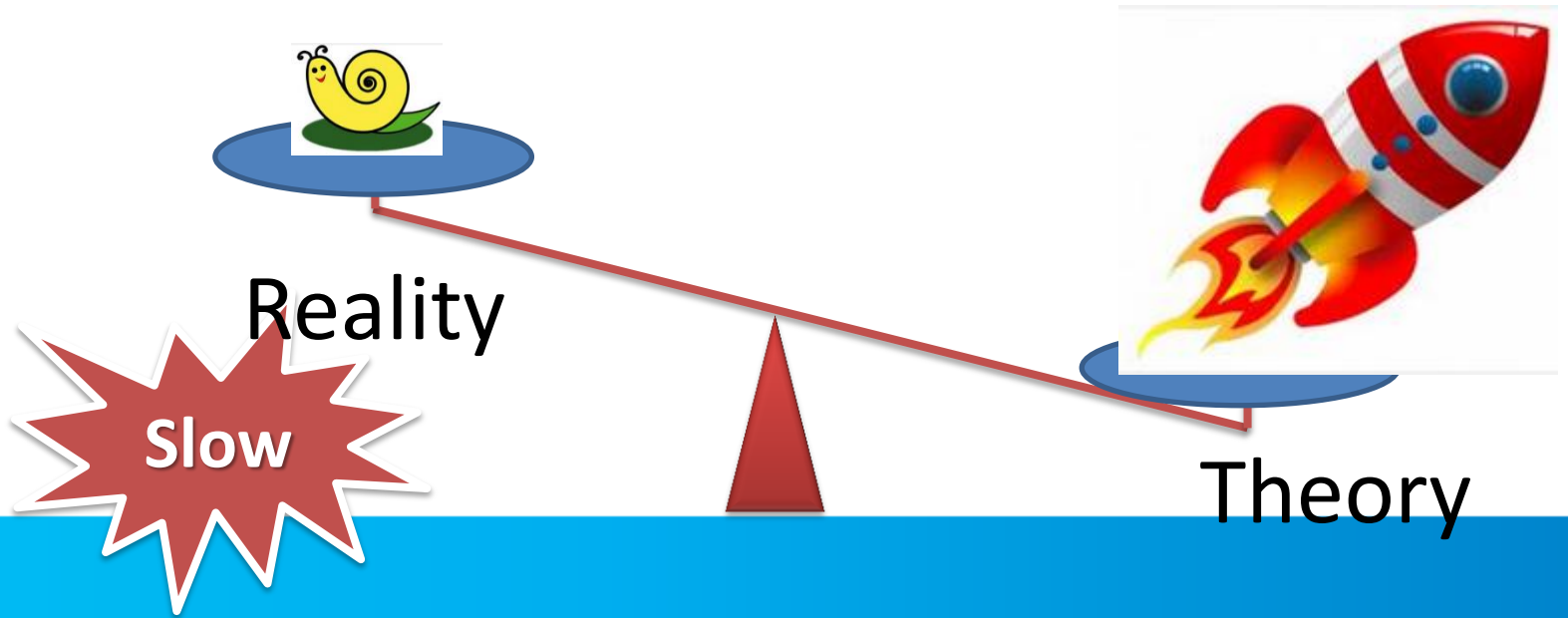
31.8s Compress and write

5.8MB/s

$7 + 30 + 101 + 31.8 =$ **170s!**

Hadoop is really very slow…

- For benchmark WordCount, common to see **2 min – 7 min** to process **800MB** uncompressed data for single map task. That ‘s about **1MB/s – 10MB/s** per map task. It is **SLOW!**



Why Hadoop is so Slow?

- IO bound, Compression/Decompression
- Inefficient Scheduling/Shuffle/Merge.
- Inefficient memory management
- Suboptimal sorting
- Inefficient Serialization & Deserialization
- Inflexible programming paradigm
- Java limitations.

Inefficient Scheduling/Shuffle/Merge

- **Scheduling**
  - **starting overhead too high, especially for iterative jobs.**
- **Shuffle**
  - Shuffle stage is doing pure IO, but it **occupy reduce slots, wasting CPU resources.**
- **Merge**
  - The merger performs **poor in merging data from multiple mappers,** resulting in too much IO, damaging the performance of IO bound Applications.

Inefficient memory management

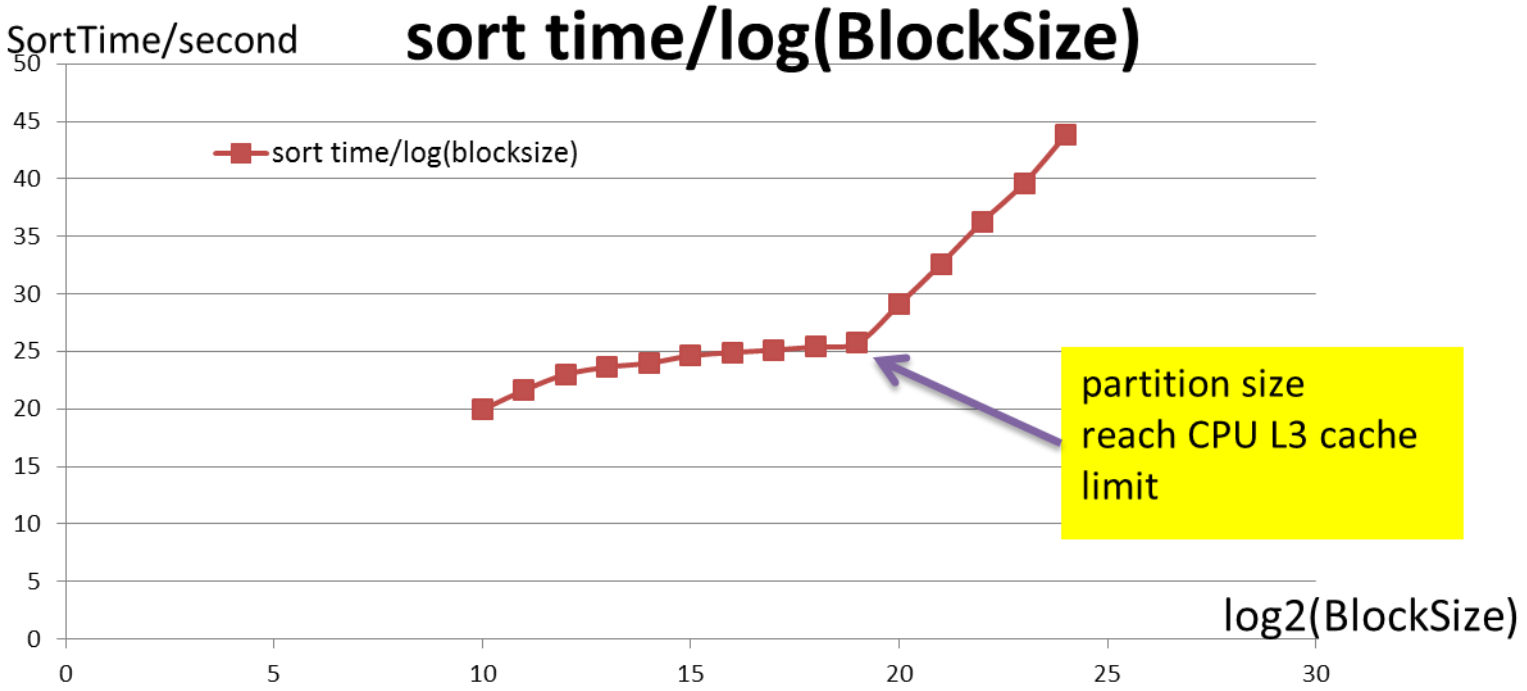
- **Sorting buffer**
  - The framework is very bad in managing sorting buffer, either wasting memory, or resulting in too many spills.
  - **The configuration need to be tuned carefully job by job**.
- **Too many memory copies.**
  - **The Streaming read/write has too many layers of decoration**, causing too many unnecessary memory copies.
  - Too frequent **small object allocation**, triggering frequent GC. GC will invalidate most CPU Cache, hurting performance.

Suboptimal sorting

- **Sorting Algorithm**.
  - Currently we use quicksort. We should switch to **Dual-pivot sort**, In our test, **Dual-pivot is 20% faster.**
- **Comparison**.
  - Currently comparison is done byte to byte. It should be done in **AVX/SSE pipeline, in QWORD(8 bytes)**
- **Sorting cache miss rate.**
  - Cache miss becomes a dominating factor in sorting. **The cache miss rate can reach higher than 80%.** And it will get worse and worse when the system load increase.

Cache miss hurts Sorting performance

- Sorting time **increase rapidly** as cache miss rate increase



- We divide a large buffer into several memory unit.
- BlockSize is the size of memory unit we doing the sorting.

Inefficient Serialization & Deserialization

- Deserialization results in **too many memory copies**.
  - **Deserialization from a bytes buffer to a object, is not necessary and waste of time.**
- Deserialization costs **too much CPU**
  - Deserialization itself is a complex call**, a lot of virtual function call,** using a lot of CPU times.
  - **Make the JIT harder to do optimization, like data vectorization.**

Inflexible programming paradigm

- **We need Non-sort Map task**
  - For many applications, we only need partition, don't need sort.(MAPREDUCE-2454 support customized sorter in Hadoop 2.03)
- **We want Map side aggregation.**
  - Hash table based aggregation. Supported in Hive, but the MR framework should have similar ability, as we are not limited to Hive use.
  - map-side join with dictionary server and etc.
- **Pipelined Map Reduce job**(partly solved in TEZ)
- **We want more flexible map data processing ability**, like SQL over Map output.

Java limitations in running Task

For **high IO bandwidth applications**:

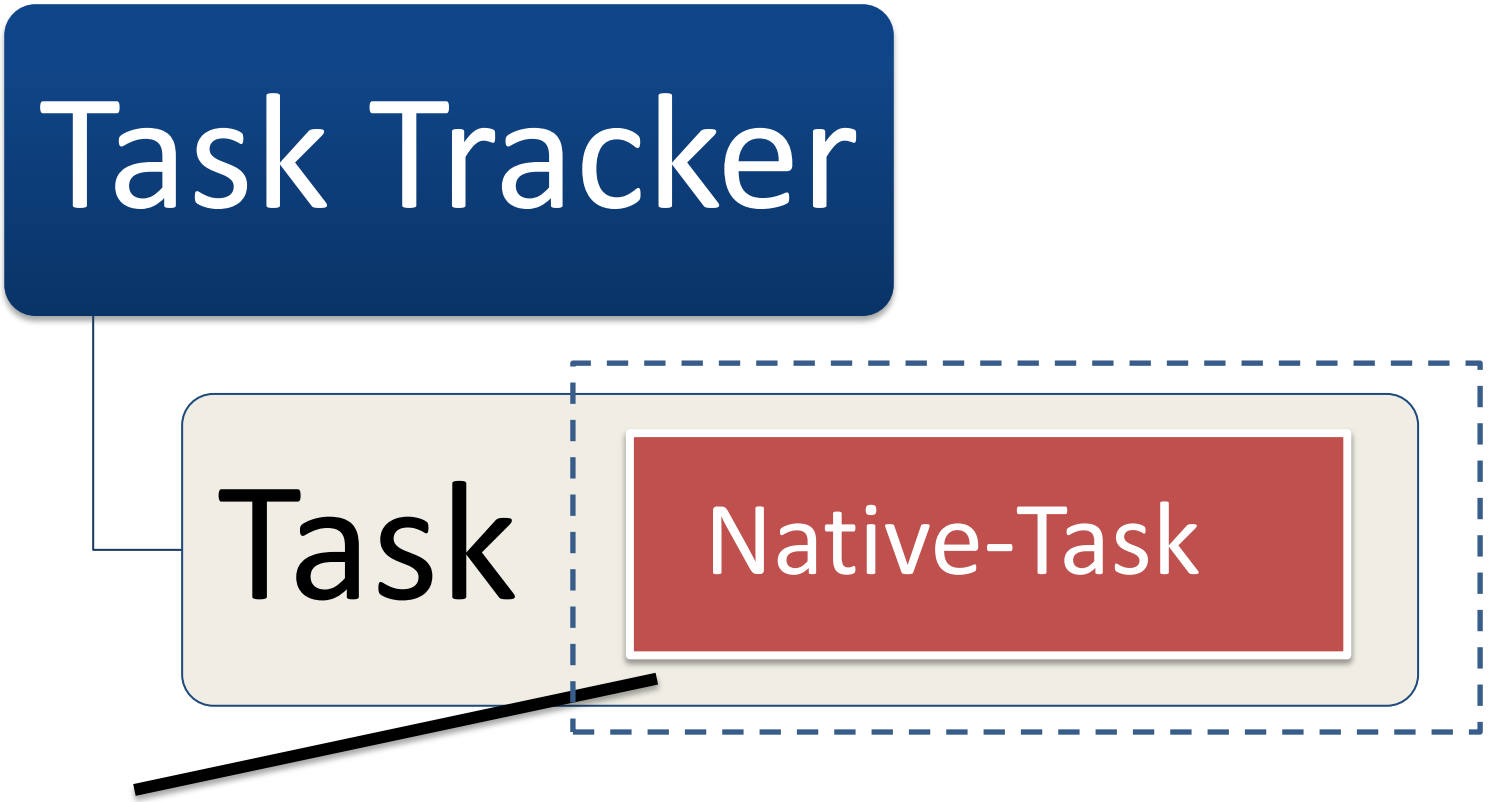
- **Hard to manage CPU resource precisely.**
  - For Map Task, we need **precise** control of CPU and memory usage. To make sure Task processes **DON'T interfere with each other.**
  - But JVM will start **tens of backend threads**, like **JIT compiler threads, GC threads; which can occupy more CPU than expected**. In some test, I observed 2000% peak CPU usage by a single Map task. **This will hurt the overall performance** in a shared cluster**.**

Java Limitations in running Task (cont.)

- **Frequent GC hurts Performance.**
  - GC will **Invalidate almost All CPU caches.**
  - Cost too much CPU, hurts other task's performance.
- **JIT optimization not quick enough to kick in.**
  - Many task runs very shortly, mostly for a few minutes. JIT optimization will not be able to kick in and optimize the code. JVM reuse will cause other problems like heap fragmentation.(link: Todd's comment on JIT impact)
- **JNI cross bound memory copy cost**
  - For high IO bandwidth App, **there is a lot of memory copies.** JNI cross bound copy will cost a lot of time. For example, we use checksum, compression, encryption, each of them will introduce another round trip between native and java.

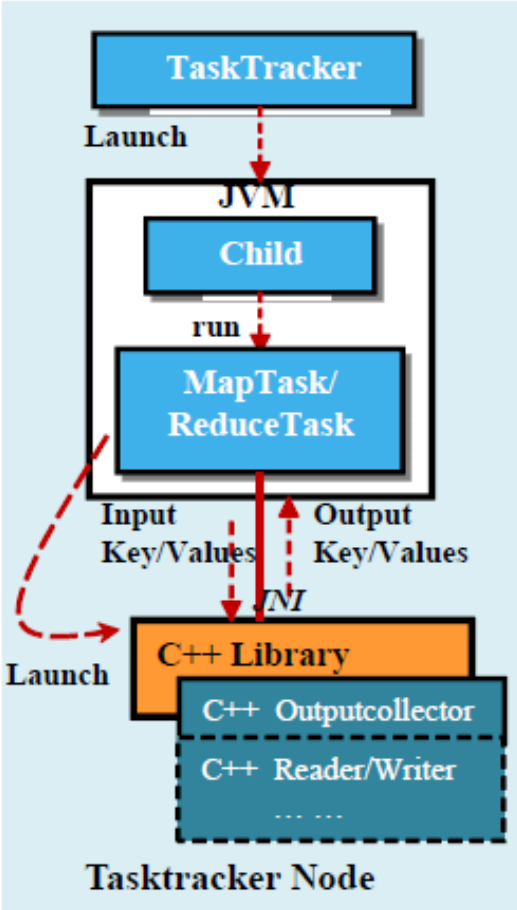
What is Native-Task?

Native-Task Engine

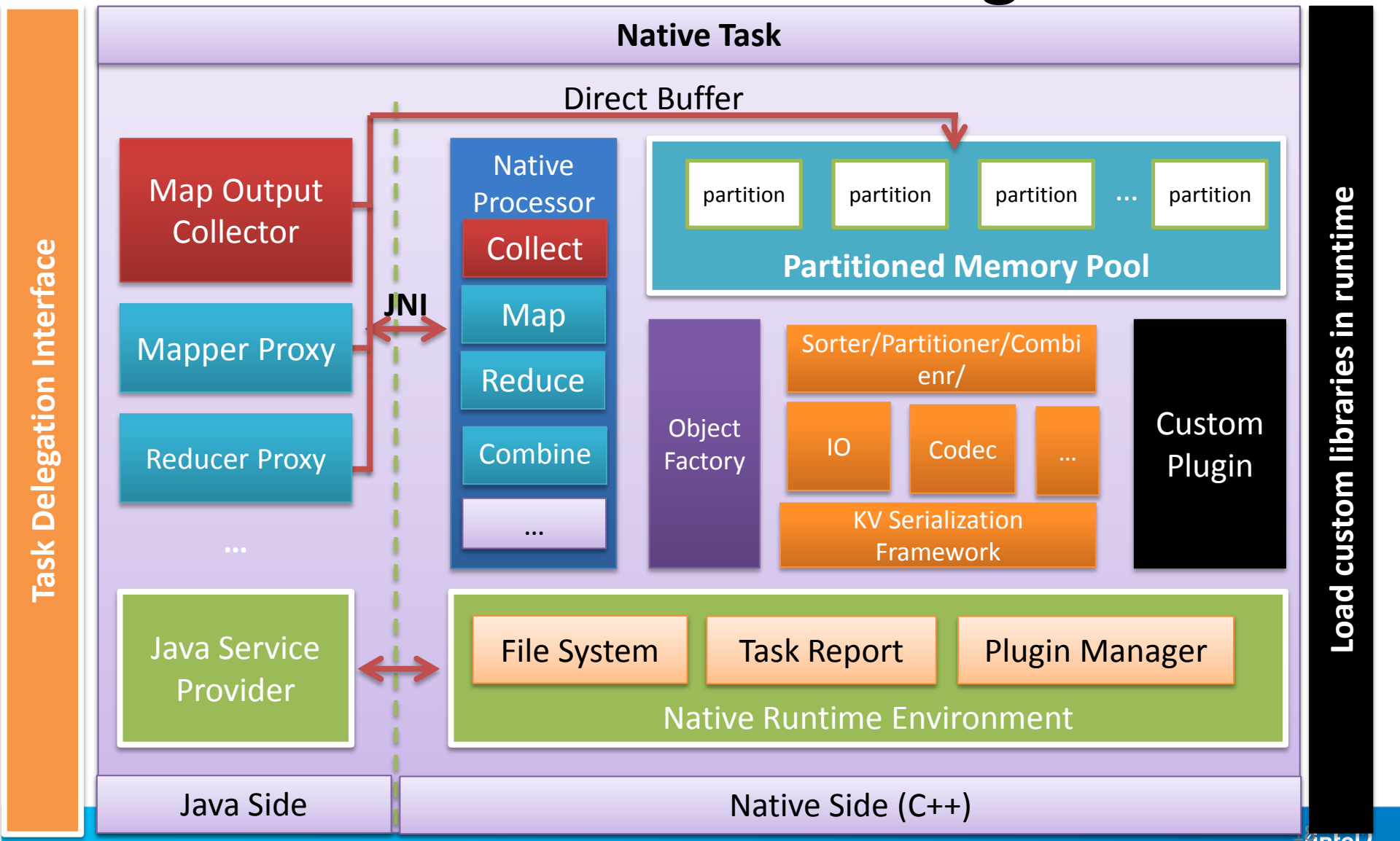


- **Native-Task is a native engine inside Task** written in C++, which focus on **task performance optimization**, while **leave** the **scheduling and communication** job to the **MR framework.**

Native-Task Dispatch flow



Native-Task Block Diagram



Native-Task at a glance

- **Native-Task Benefits:**
  - ✓**Fully compatible** with existing Java MR apps. **Transparently** support **HIVE, HBASE**.
  - ✓The **Sorting performance** is **10x – 20x faster.**
  - ✓Support **non-sort partitioning, group** and other flexible sorting paradigm.
- Native-Task **focus on map-stage performance optimization.**

Focus on **map-stage optimization**

Map stage optimization is **MOST** important!

- **Most computation are done in map stage.**
  - For Internet work load, Mapper data volume / Reducer data volume ~= **10/1.** For many Hive typical queries, **the ratio is much bigger.**
- **Map stage occupy 90% of the overall job time** usually(for example, for word count: 99% time is doing map stage).

Native Task IS

- Native Task is focusing on map stage optimization.
- It do so by replacing partial implementation of task.
- It replaced the output collector implementation, and implement high efficient memory management, sorting, io and etc..
- It is compatible with existing MR, both on API level, and at code level.

Native-Task IS NOT

- Native-Task is not a rewrite of hadoop or mapreduce.
- It have not changed any part of job tracker, task tracker, namenode, datanode.
- It is not handling the communication with task tracker.
- It is not handling DFS read/write directly, it delegate it to java API.

Compare with Hadoop Pipes

- They both use native code.
- Hadoop Pipes **don't change Java MR framework**, it provides programming interface in C++ for better compatibility. The major objective is **compatibility**.
- Native-Task focus on **performance.** Native-Task **replace partial Task implementation with Native implementation,** while keeping Java Mapper and Java Reducer unchanged.
- Hadoop Pipes can **co-work** with Native-Task.

Design and Implementation

Native-Task design principals

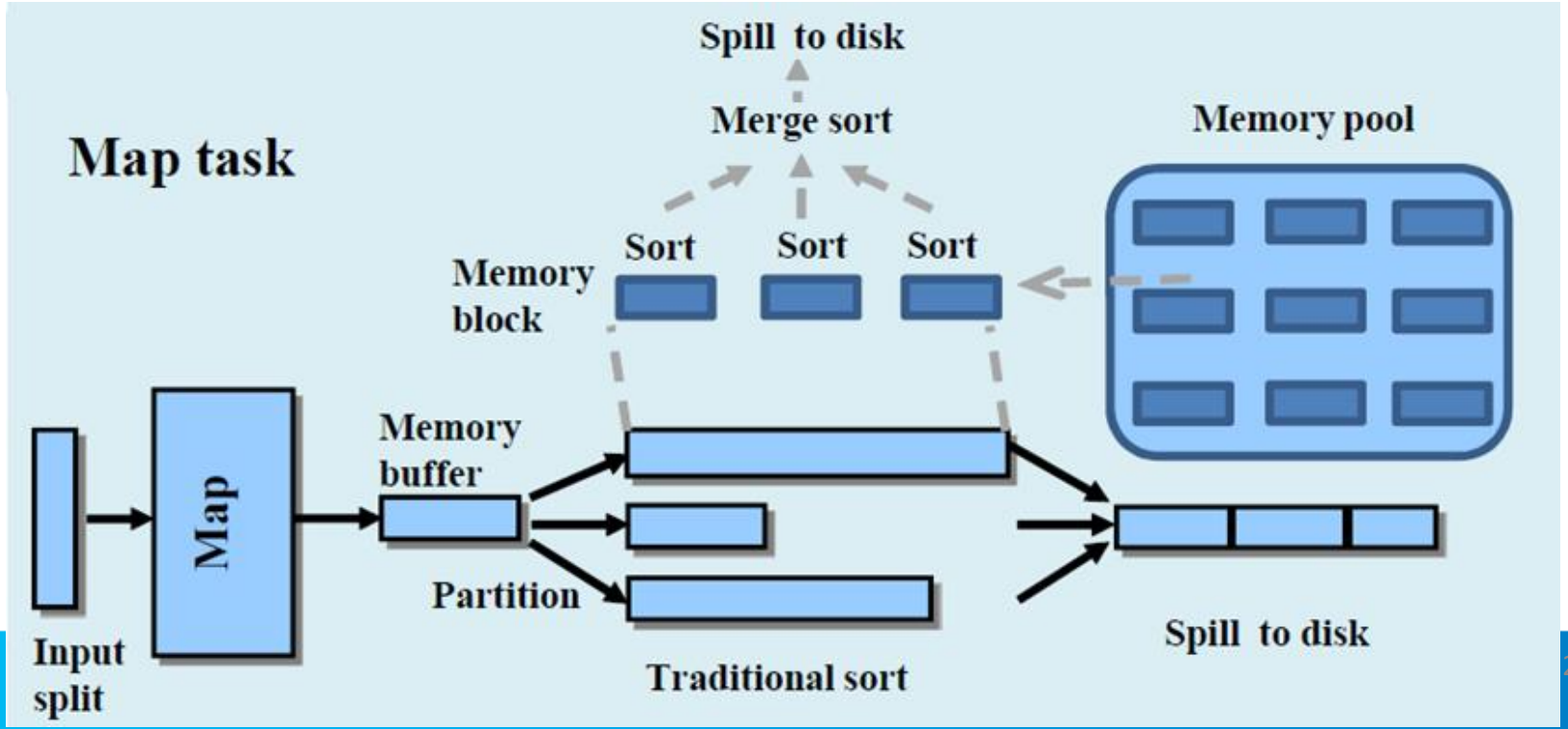
- **Performance is the key objective**
- **Compatibility with existing MR Apps**
  - We can immediately get performance boost for existing MR.
- **Flexible program paradigm** to support broader Apps.
- **Highly Extensible Plugin framework.**
  - Can be extend to plug in customized implementation in runtime.

Why Native-Task is faster?

- **High efficient Memory Management**
  - **More controllable Memory footprint**. Memory usage reduced by 14% for benchmark TeraSort.
  - **Self adaptive memory allocation,** no need to tune configuration job by job.
  - Avoid unnecessary memory copy in all places.
  - **Operate directly on the buffer, avoid Serialization/Deserialization cost.**
- **Use optimized compression/decompression codecs**
  - **Without JNI cross-bound cost**, reducing memory copy, we are faster than Java counterparts.

Why Native-Task is faster? (cont.)

- **Highly optimized sorting**. Sorting performance is 10x – 20x faster.
  - **Dual-Pivot quick sort.**
  - Partition based sorting, Implemented a cache aware sorting. **Reduce 70% cache miss.**
  - Aggressive function inline for comparison.



Why Native Task is faster? (cont.)

- **Use hardware optimization when necessary**
  - Implement **AVX/SSE** friendly **bytes comparison**, compare QWORD at a time. much faster than system call ::memcmp()
  - Intel compiler
  - **Native checksum**(5 times faster)
  - **Data manipulation with native instructions.**
- **Avoid Java runtime side-effects**
  - More **precise CPU cache control,** More **predictable memory footprint.**
  - **More controllable task CPU usage.** A single task will use single core, reduce the CPU impact of JIT compiler threads and GC threads.

Native-Task is **Compatible with existing MapReduce Application**

- Support **HIVE, HBASE transparently**
- **3 lines** change to Hadoop core. Trivial efforts to patch.
- **Automatically fallback to original Hadoop collector if some feature not supported.**
- This feature can be **turned on/off on job basis**.
- Designed TaskDelegation Interface to transfer control

```
class TaskDelegation {
  public static <K, V> MapOutputCollectorDelegator<K, V>
  getOutputCollectorDelegator(TaskUmbilicalProtocol protocol,
      TaskReporter reporter, JobConf job, Task task);
}
```

Flexible program paradigm

- Support **Non-Sort map task.**
  - Great for many Hive operations.
  - Significantly reduce map stage time.
- Support **Hash Join**.
- **Extensible** to **support more other sorting paradigm.**

Extensible plugin framework

- **Extensibility** is a **key principal** when designing Native Task.
- Almost all built-in code **can be replaced by customized implementation.**
- A framework to **support customized** native **combiner**, native **key comparator**, native **sorting algorithm**, customized **compression codec**, native **record** reader/**writer,** native **collector,** native **CRC** checksum**.**

Native-Task Status

Native-Task Feature List

- Implemented:
  - **Transparently support existing MR App.**
  - **ALL Value types are supported.**
  - **Most common key types** are supported.
  - **Java combiner** are supported.
  - **LZ4/Snappy** supported.
  - **CRC32 and CRC32C**(hardware checksum) supported.
  - Supports **Hive/Mahout transparently**.
  - **MR over HBase** supported (BytesWritable).
  - **Non-Sort Map**
  - **Hash Join**
  - **Pig**
- Not implemented:
  - **No support** for user **customized JAVA key comparator**, we have support for **customized NATIVE comparator**.

List of Supported Key Types

Hadoop.io

- org.apache.hadoop.io.BytesWritable;
- org.apache.hadoop.io.BooleanWritable;
- org.apache.hadoop.io.ByteWritable;
- org.apache.hadoop.io.DoubleWritable;
- org.apache.hadoop.io.FloatWritable;
- org.apache.hadoop.io.IntWritable;
- org.apache.hadoop.io.LongWritable;
- org.apache.hadoop.io.Text;
- org.apache.hadoop.io.VIntWritable;
- org.apache.hadoop.io.VLongWritable;

HBase

- org.apache.hadoop.hbase.io.ImmutableBytesWritable

Hive

- org.apache.hadoop.hive.ql.io.HiveKey

Pig

- NullableIntWritable
- NullableLongWritable
- NullableFloatWritable
- NullableDoubleWritable
- NullableBooleanWritable
- NullableDateTimeWritable
- NullableBigIntegerWritable
- NullableBigDecimalWritable
- NullableText
- NullableTuple

Mahout

- EntityEntityWritable
- Gram
- GramKey
- SplitPartitionedWritable
- StringTuple
- TreeID
- VarIntWritable
- VarLongWritabl

Performance Test

Hi-Bench test cases

Workbench	
Wordcount	CPU-intensive
Sort	IO-intensive
DFSIO	IO-intensive
Pagerank	Map :CPU-intensive Reduce :IO-intensive
Hivebench-Aggregation	Map :CPU-intensive Reduce :IO-intensive
Hivebench-Join	CPU-intensive
Terasort	Map :CPU-intensive Reduce : IO-intensive
K-Means	Iteration stage: CPU-intensive Classification stage: IO-intensive
Bayes	Key type NOT supported by Native-Task.
Nutch-Indexing	Map: very short running Shuffle: IO intensive Reduce: CPU intensive

Cluster settings

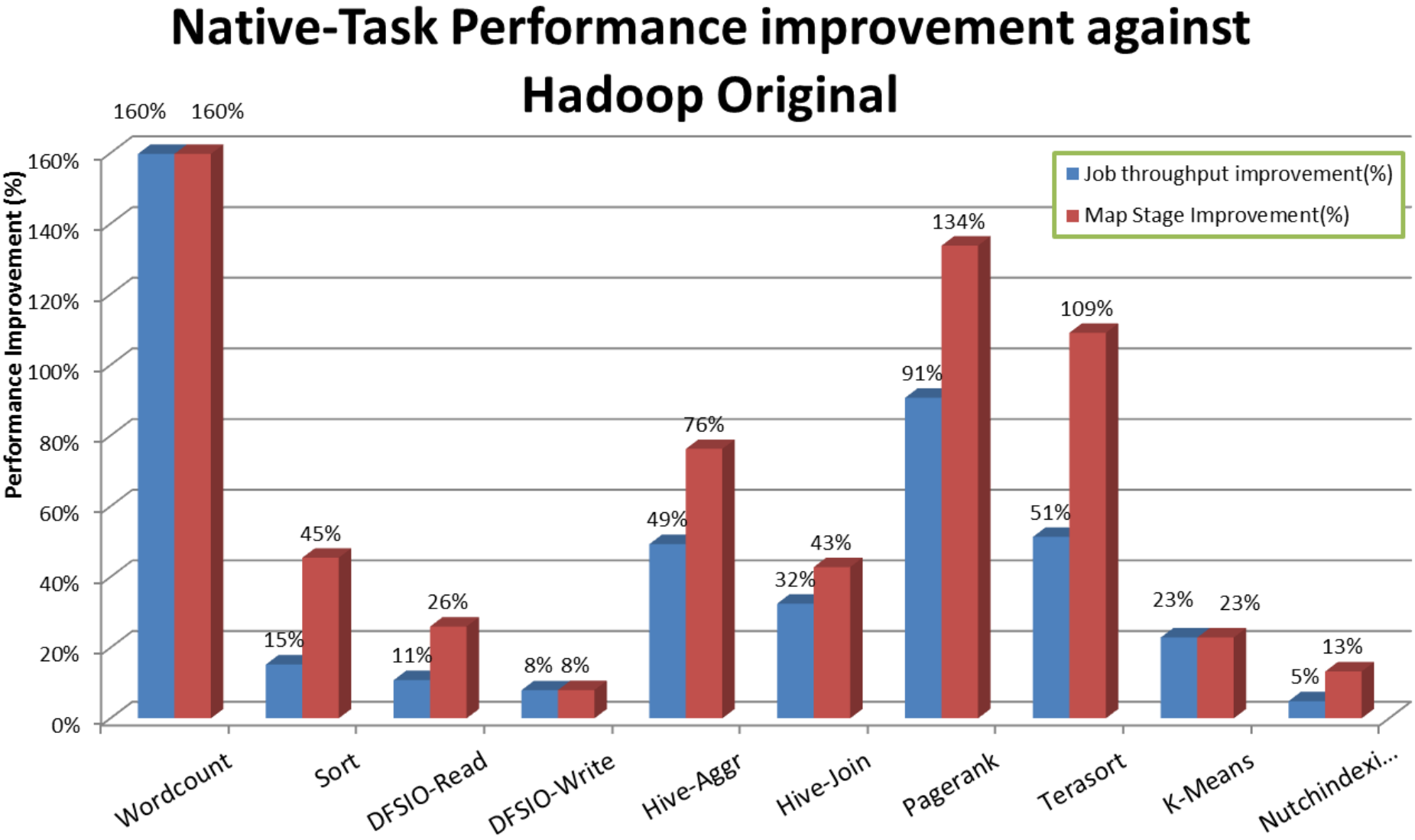
Cluster environment

Hadoop version	IDH 2.4, Hadoop 1.0.3-Intel (patched with native task)
Cluster size	4
Disk per machine	7 SATA Disk per node
Network	GbE network
CPU	4 * 8 core, E5-2680 (32 cores in total per node)
L3 Cache size	20480 KB per CPU
Memory	64GB per node
Map Slots	3 * 32 + 1 * 26 == 122 map slots
Reduce Slots	3 * 16 + 1 * 13 = 61 slots

Job Configuration

io.sort.mb	1GB
compression	Enabled
compression algo	snappy
dfs.block.size	256MB
Io.sort.record.percent	0.2
dfs replica:	3

Native-Task Benchmark (Hi-Bench)

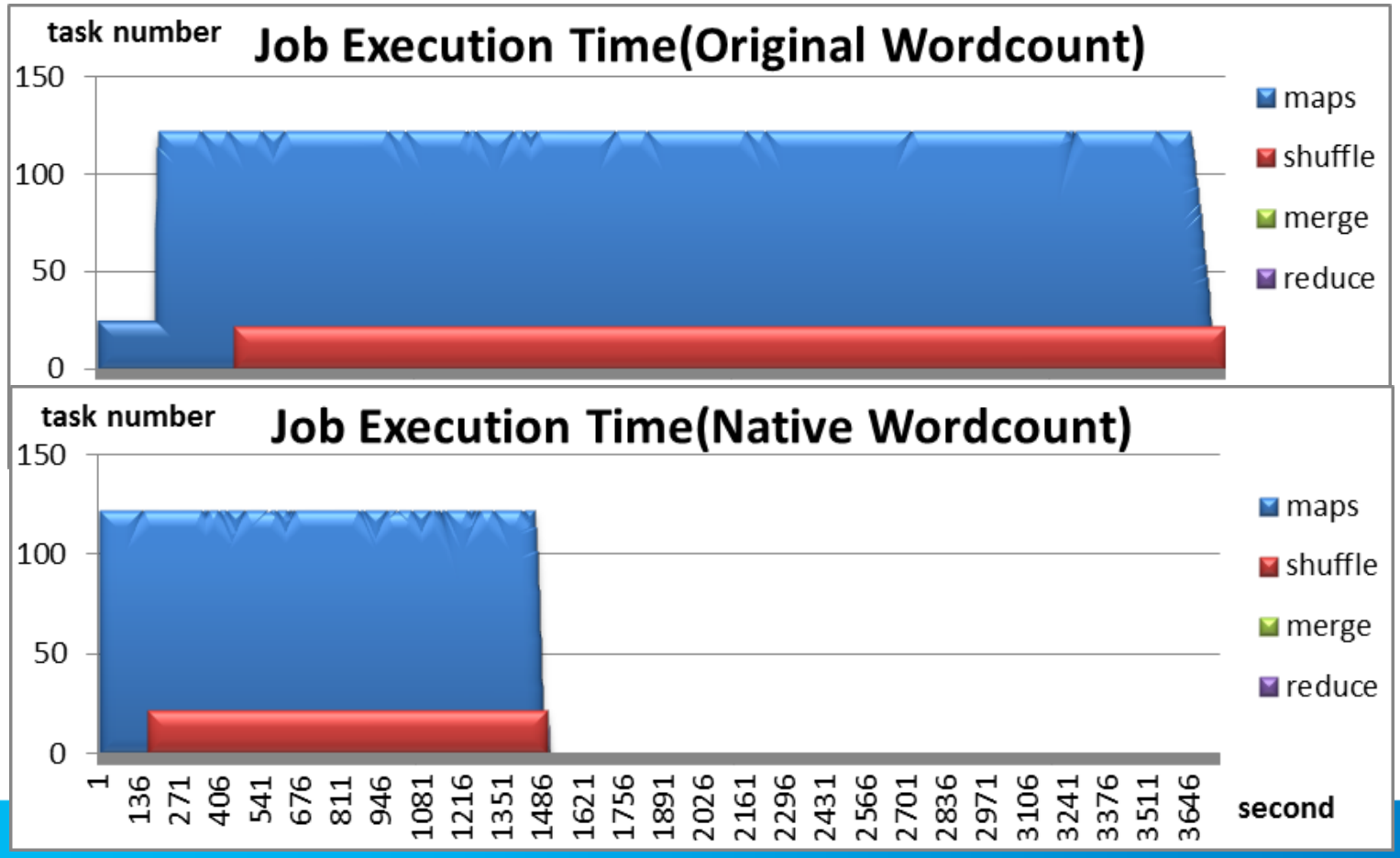


Hi-Bench Performance

	Data before compression	Data after compression	Original job run time(s)	Native job run time(s)	Job throughput Improvement	Map stage throughput Improvement
Wordcount	1TB	500GB	3957.11	1523.43	**159.8%**	**160%**
Sort	500GB	249GB	3066.97	2662.43	**15.2%**	**45.4%**
DFSIO-Read	1TB	NA	1384.52	1249.68	**10.8%**	**26%**
DFSIO-Write	1TB	NA	7165.97	6639.22	7.9%	7.9%
Pagerank	Pages:500M Total:481GB	217GB	11644.63	6105.71	**90.7%**	**133.8%**
Hive-Aggregation	5G Uservisits 600M Pages Total:820GB	345GB	1662.74	1113.82	**49.3%**	**76.2%**
Hive-Join	5G Uservisits 600M Pages Total:860GB	382GB	1467.08	1107.55	**32.5%**	**42.8%**
Terasort	1TB	NA	6360.49	4203.35	**51.3%**	**109.1%**
K-Means	Clusters:5, Samples:2G, Total:378GB	350GB	8706.82	5734.11	**22.9%**	**22.9%**
Nutch-Indexing	Pages:40M Total: 222GB	NA	4601	4388	**4.9%**	**13.2%**

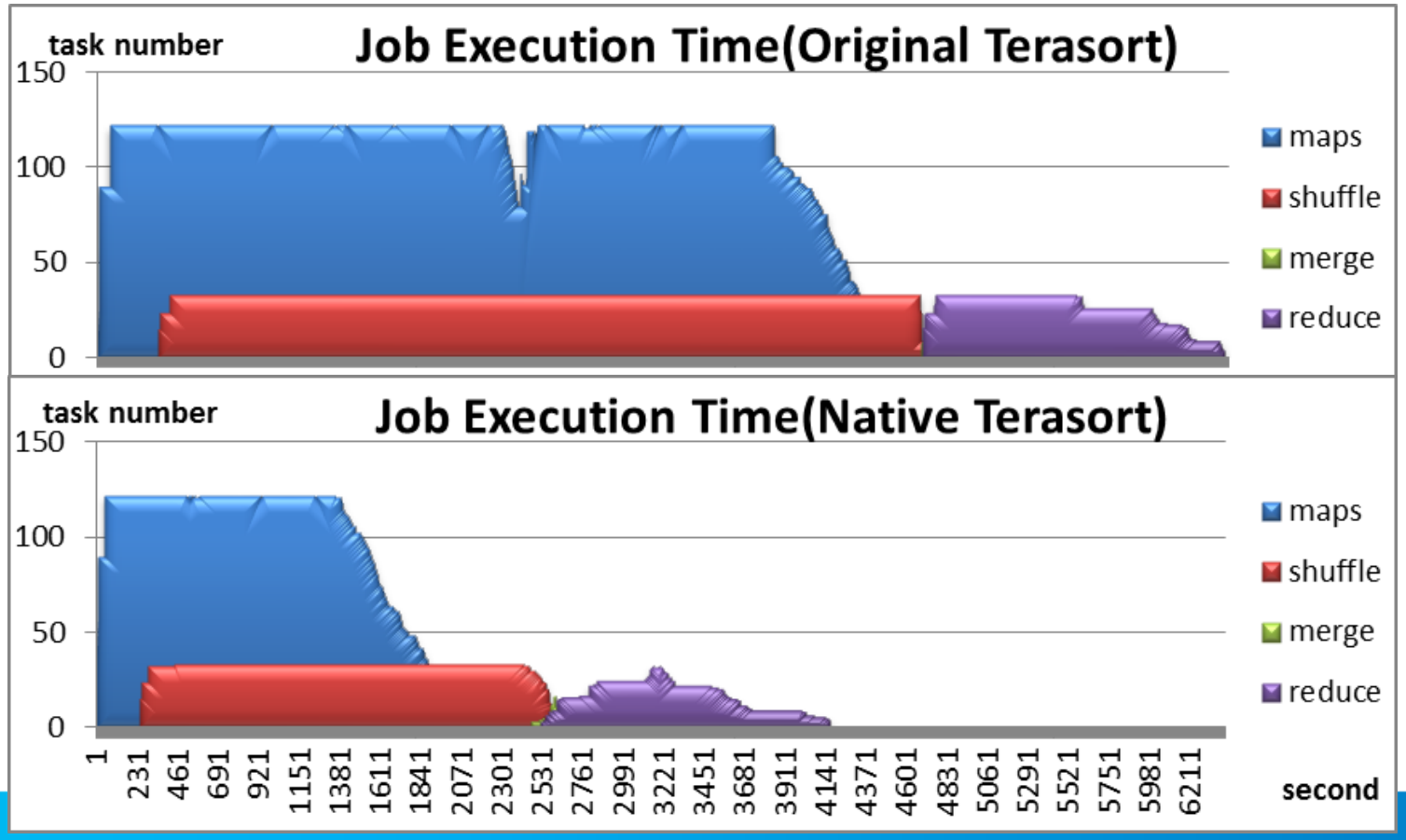
WordCount breakdown

- 1TB, Job throughput: **2.6x**, map stage: **2.6x**
- **CPU intensive.**



TeraSort Breakdown

- Job throughput: **1.5x,** map stage: **2.1x**
- **Map: CPU intensive, reduce: IO intensive**

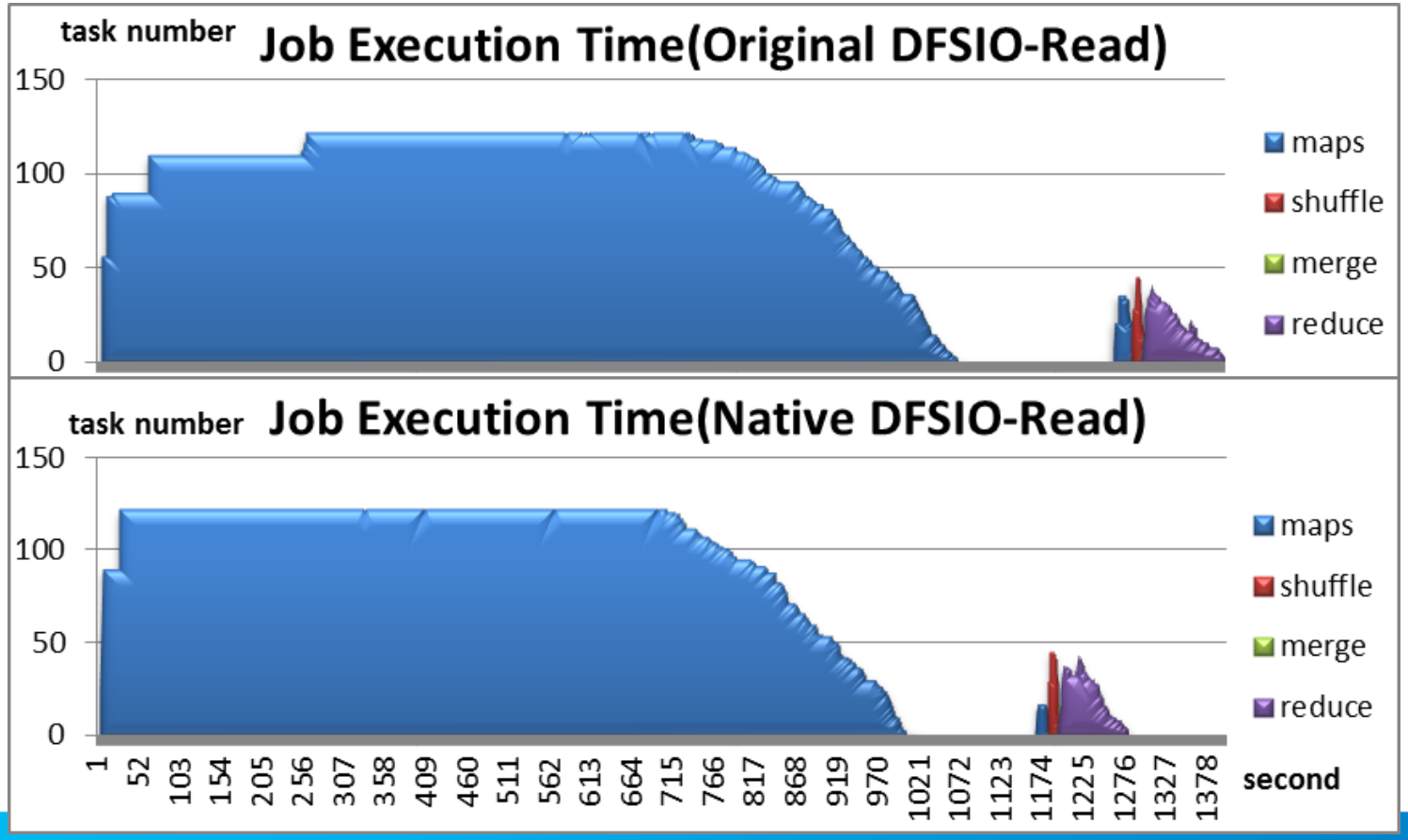


Sort Breakdown

- 500GB, job throughput: **1.15x**, map stage: **1.45x**
- **Reduce: IO intensive** (bound by network)

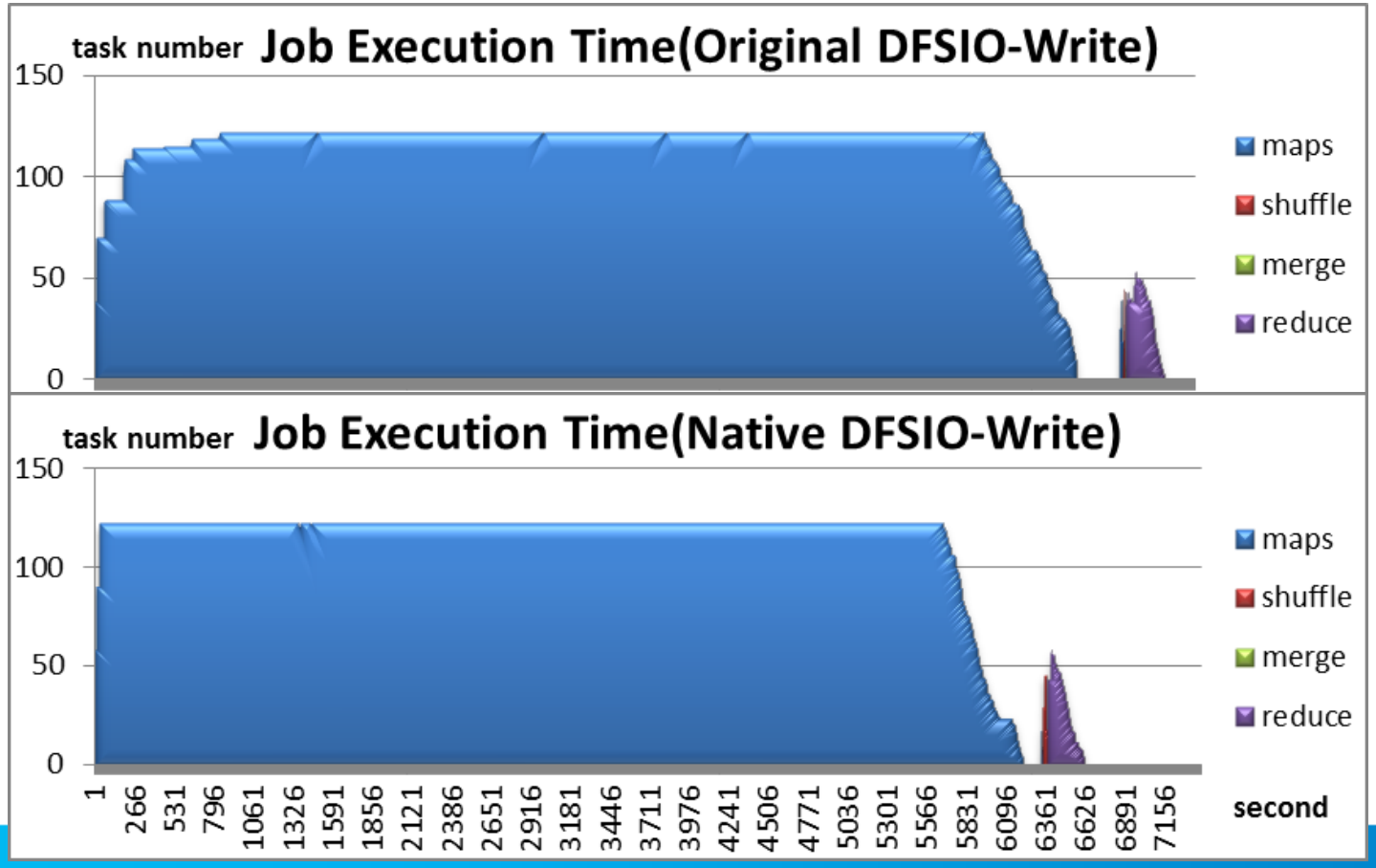
DFSIO-Read Breakdown

- Job throughput: **1.1x,** map stage: **1.26x**
- **IO intensive**, 2 Map-Reducer jobs



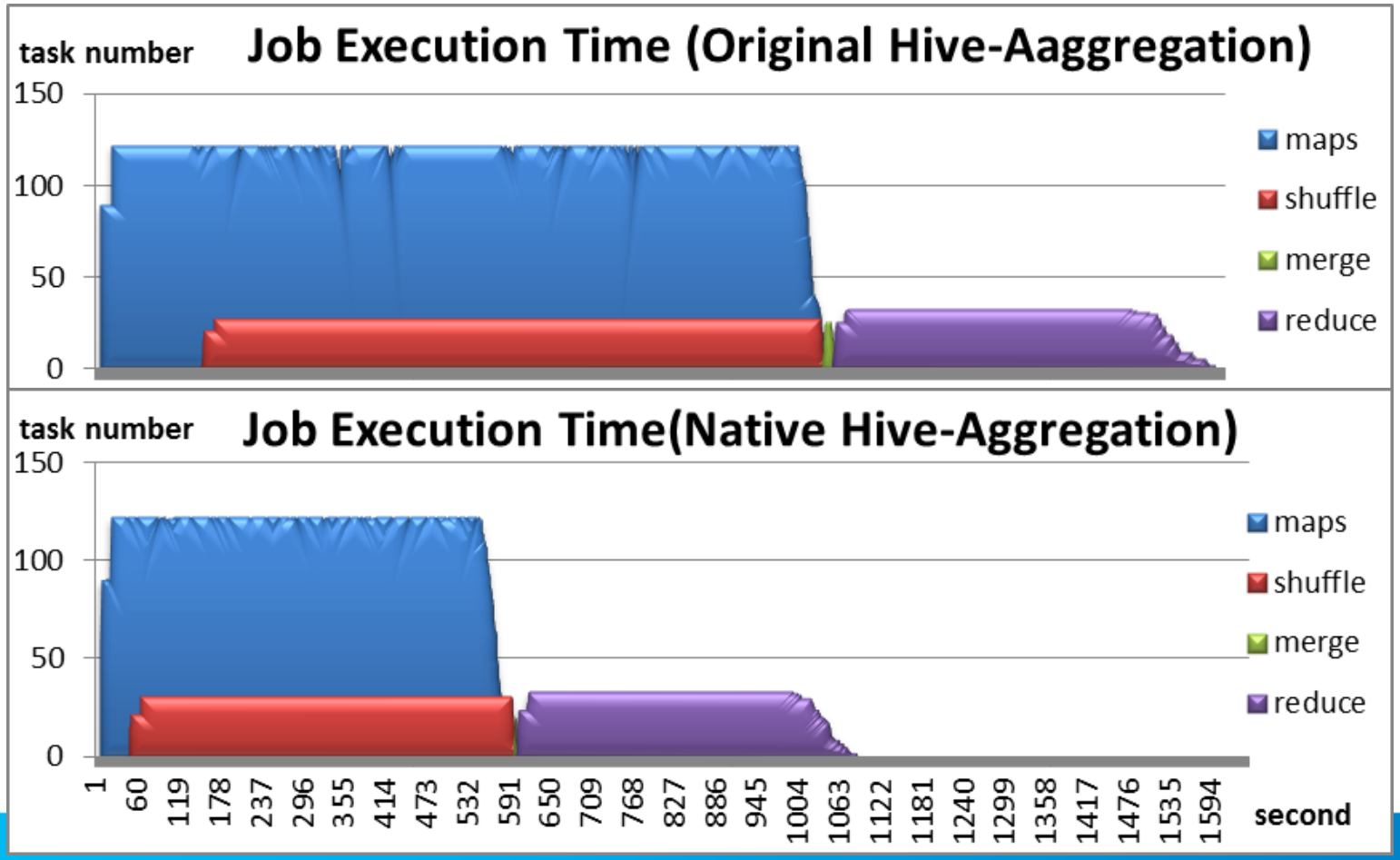
DFSIO-Write Breakdown

- Job throughput: **1.08x**, map stage: **1.08x**
- IO intensive, 2 Map-Reducer jobs



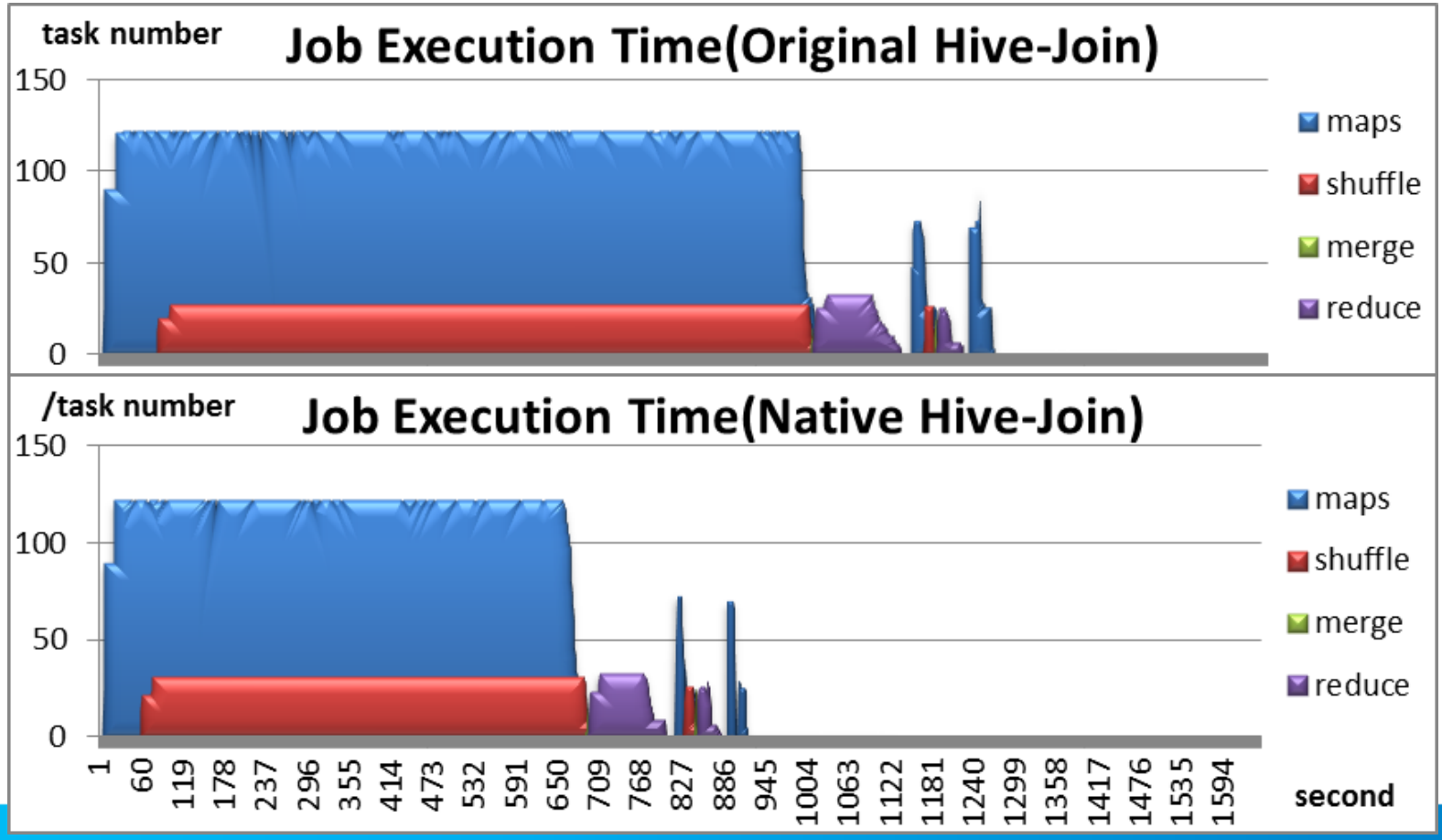
Hive-Aggregation breakdown

- Job throughput: **1.5x**, map stage: **1.76x**
- CPU intensive



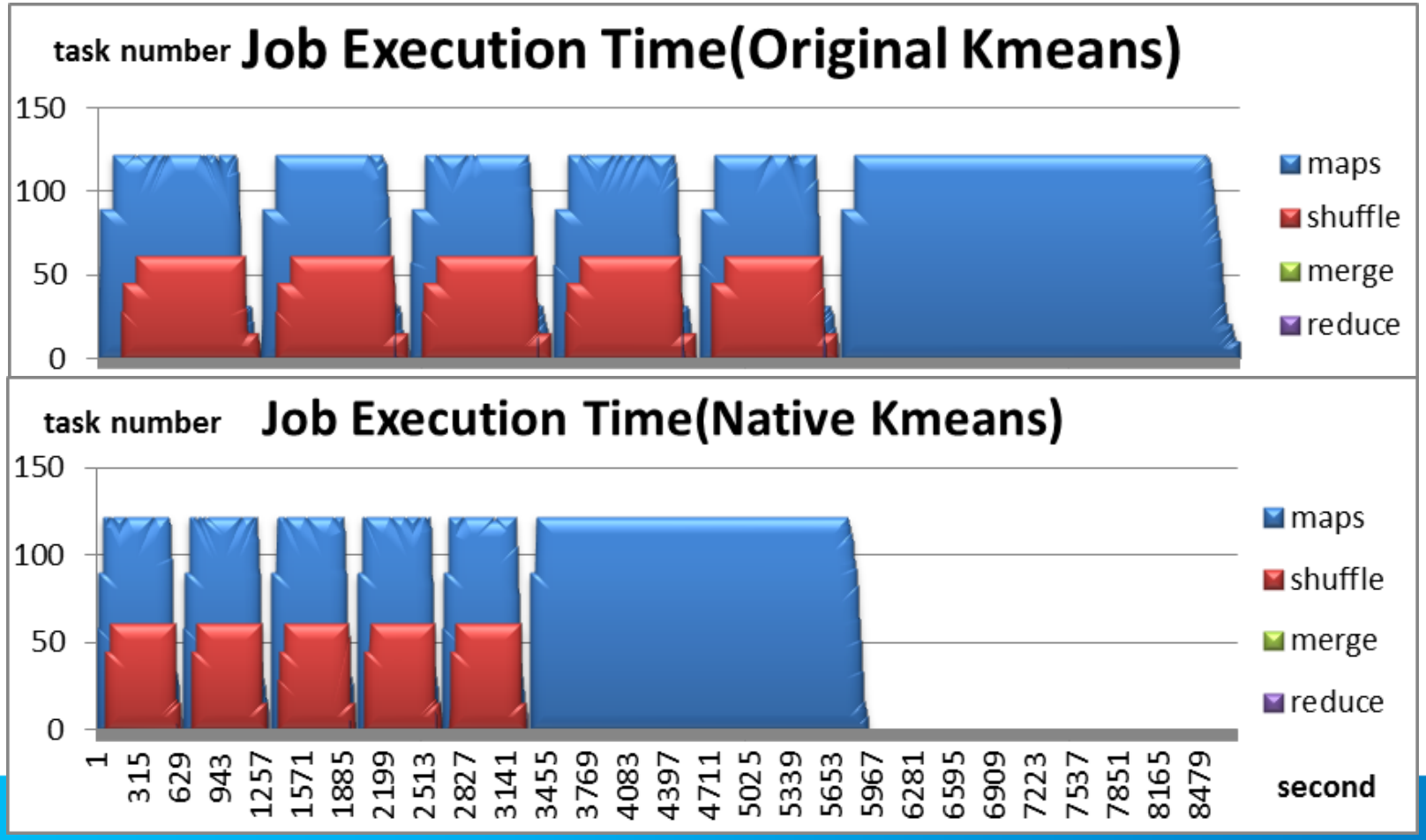
Hive-Join breakdown

- Job throughput: **1.32x**, map stage: **1.43x**
- CPU intensive, 4 Map-Reduce jobs



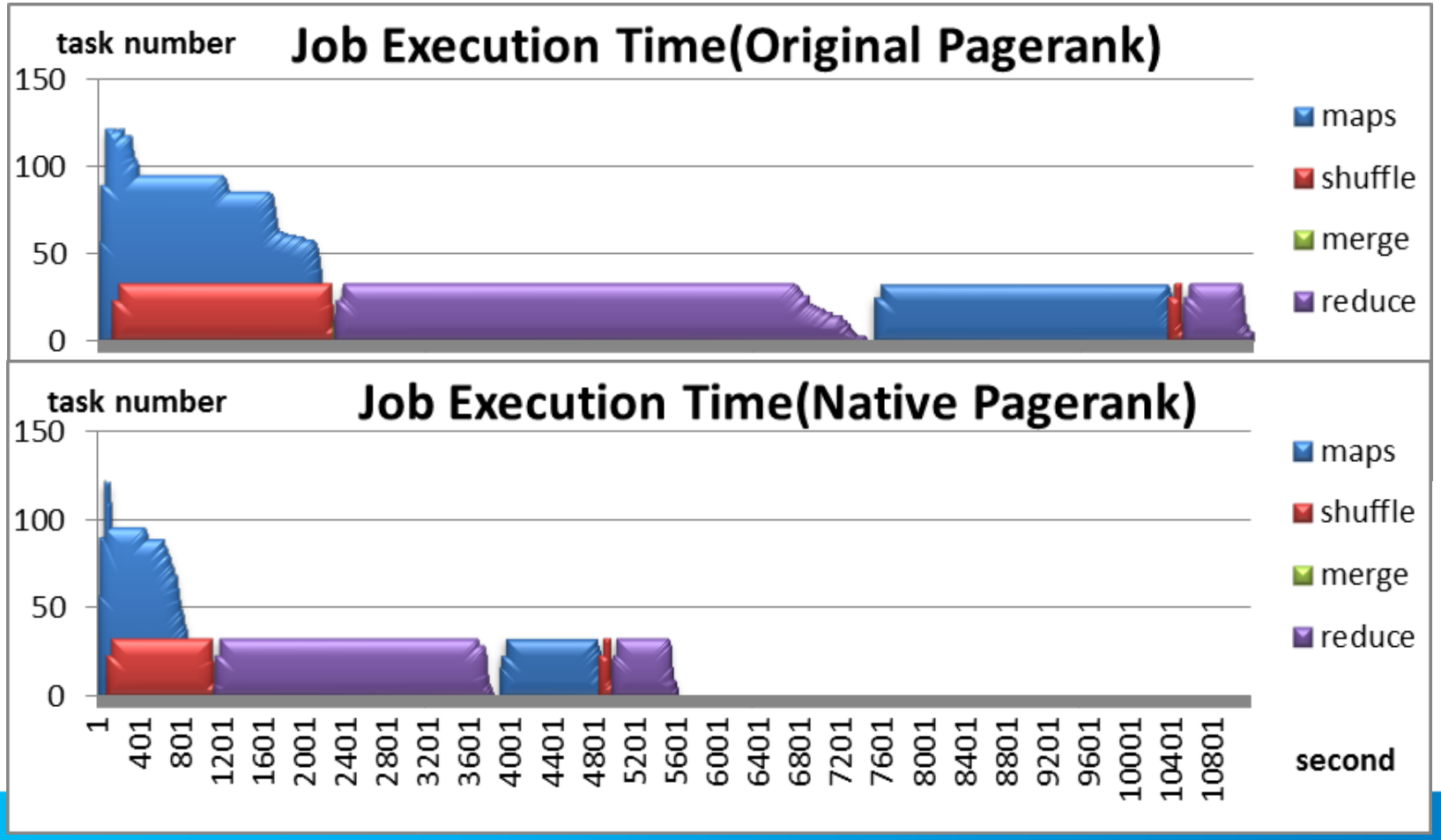
K-Means Breakdown

- Job throughput: **1.23x,** map stage throughput: **1.23x**
- CPU intensive, 5 iterations



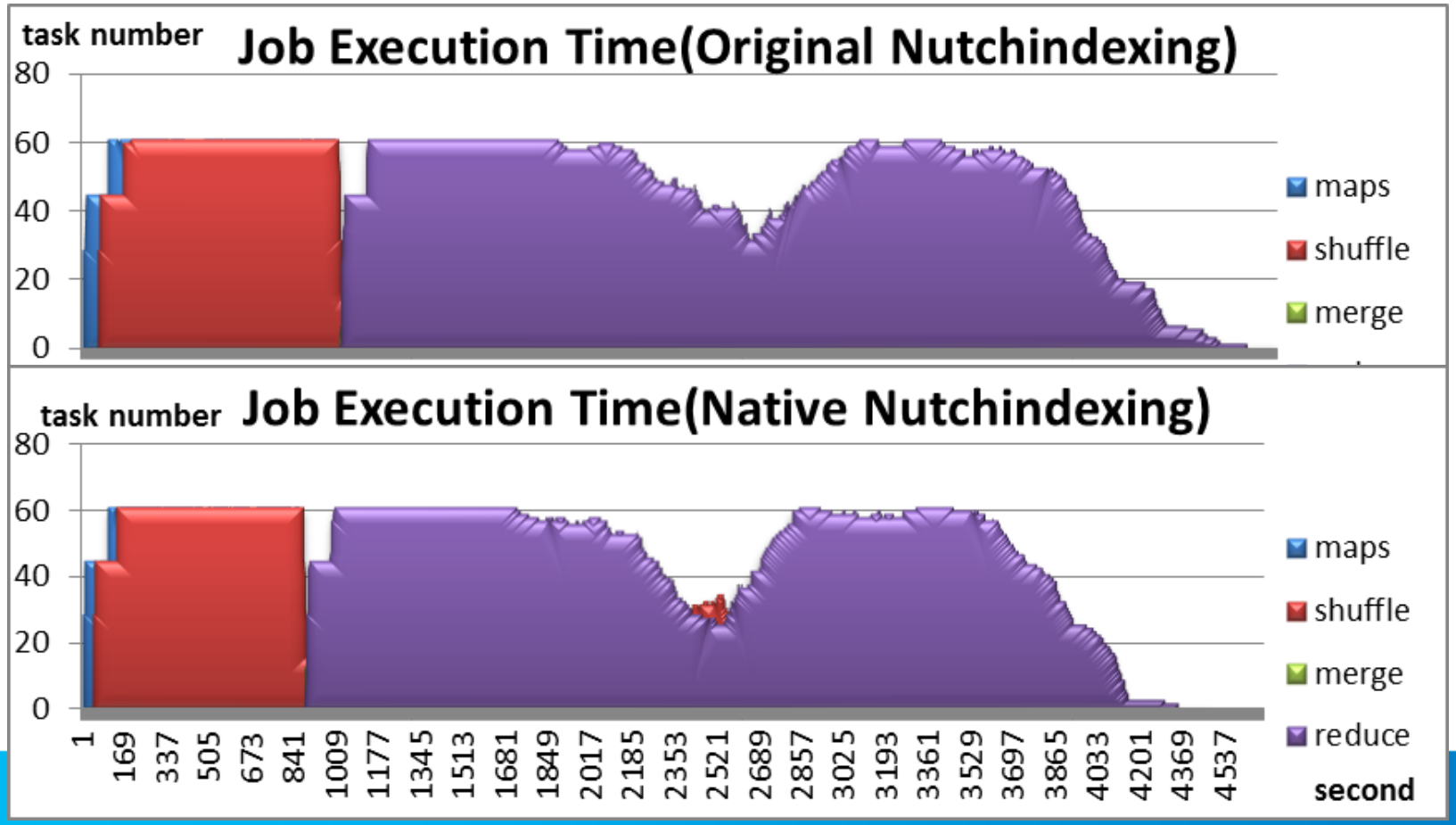
PageRank Breakdown

- Job throughput: **1.97x, map stage: 2.34x**
- CPU intensive, 2 map-reduce jobs

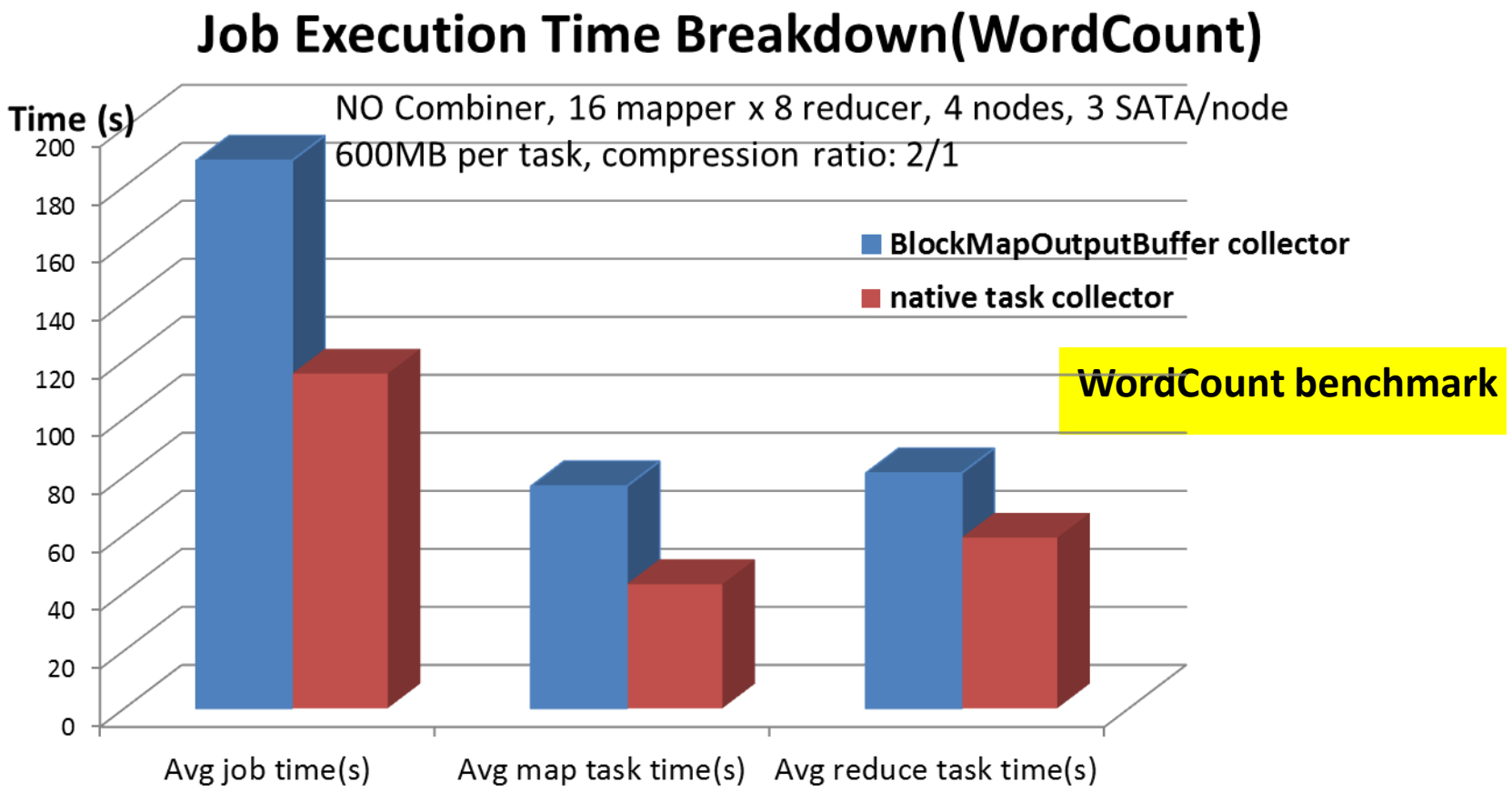


Nutch-Indexing Breakdown

- Job throughput: 1.05**x, map stage: 1.13x**
- Map is very short, Shuffle: IO intensive, Reduce: CPU intensive. 2 rounds reduce



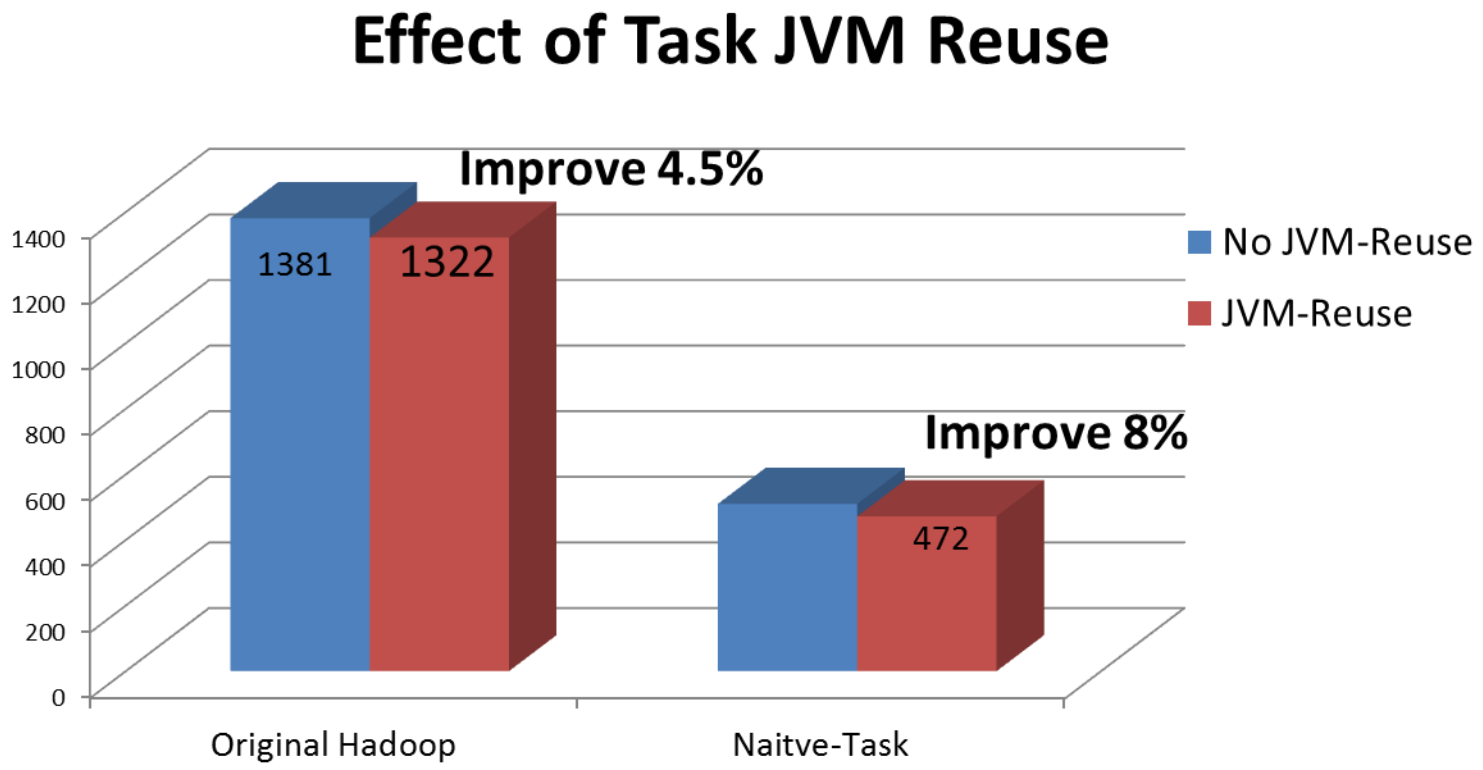
Compare with BlockMapOutputBuffer



- **70% faster** than BlockMapOutputBuffer collector.
- BlockMapOutputBuffer **supports ONLY BytesWritable**

Effect of JVM reuse

- 4.5% improve for Original Hadoop, 8% improve for Native-Task

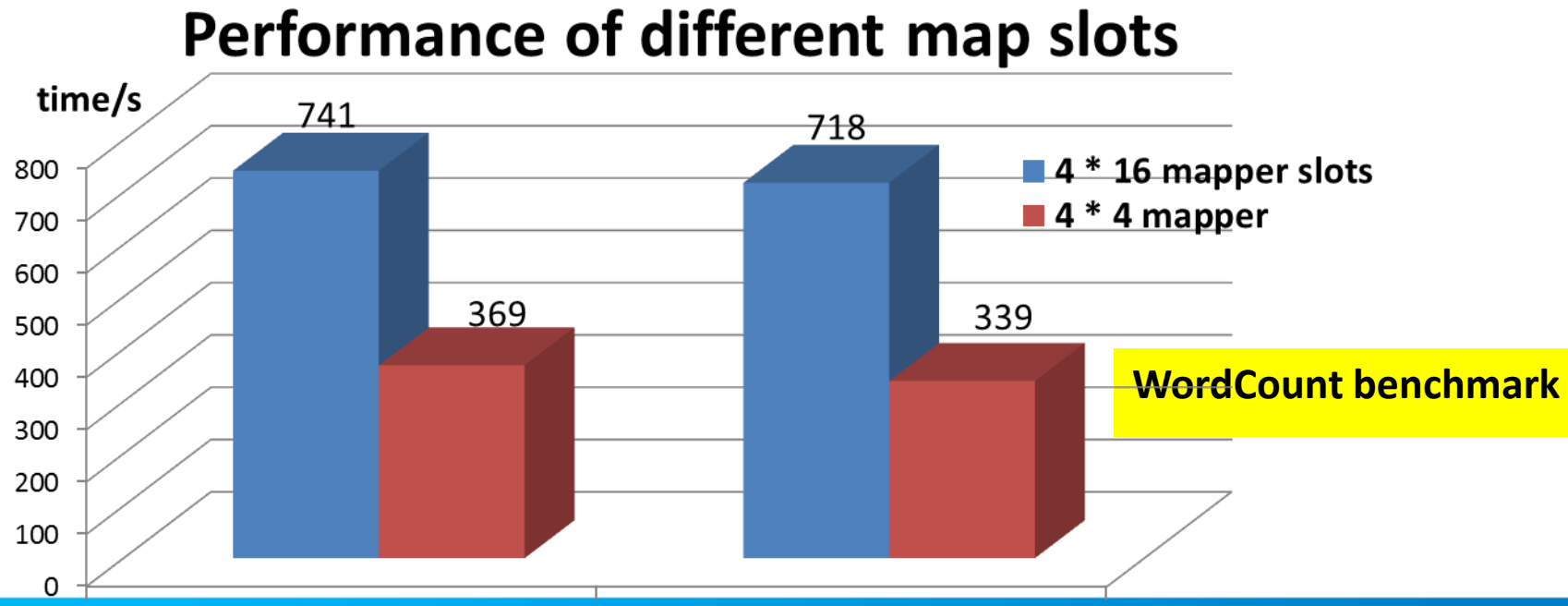


Can be **3x faster further**…

- The Hadoop **don't scale well when slots number increase**. We **expect 1.5x-2x performance increase** when **slots# doubles**, But **performance actually drops in current Hadoop.**
- Many map task only runs for **30s or less**, this will amplify the impact of framework scheduling latency.
- Performance can be **boosted 2x** further if **implementing the whole task in native. Currently only Map output collector is in Native.**

Hadoop don’t scale well when slots number increase

- 4 nodes(32 core per node), 16 map slots max, **CPU, memory, disk are NOT fully used.**
- **Performance drops unexpectedly when slots# increase**.



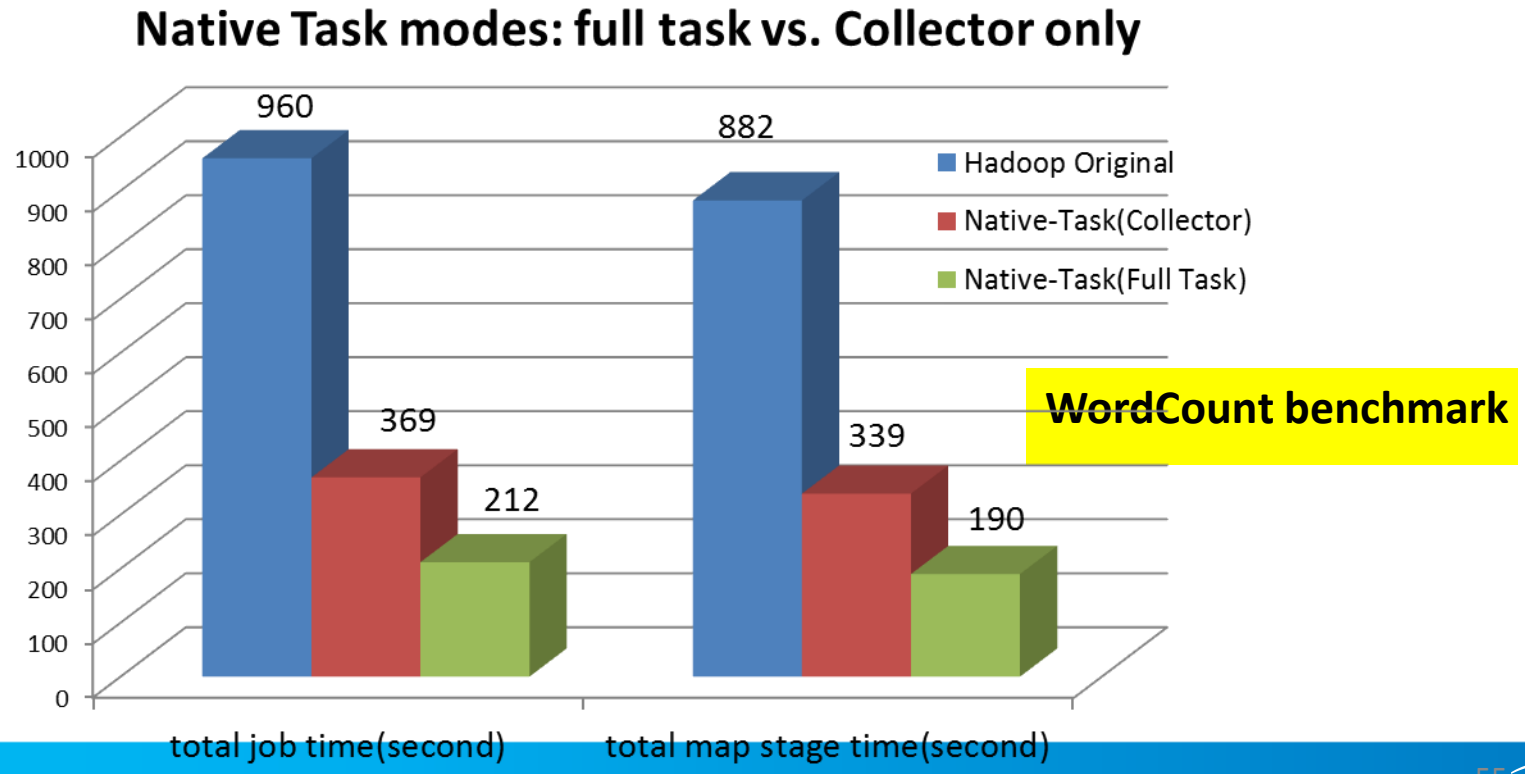
Beyond Native Collector optimization

Full Task optimization

- **To be compatible with existing Map Reduce Applications**, **Native-Task supports Java mapper/combiner,** with map output collector implemented in Native.
- **The java mapper/combiner is very inefficient**, need to be optimized.
- **Full Task optimization** will **optimize mapper/combiner in Native,** along with **record reader, record writer, partitioner, reducer, and etc..**

Native-Task mode: full task optimization

- **2x faster further for Native-Task full time optimization**, compared with native collector.



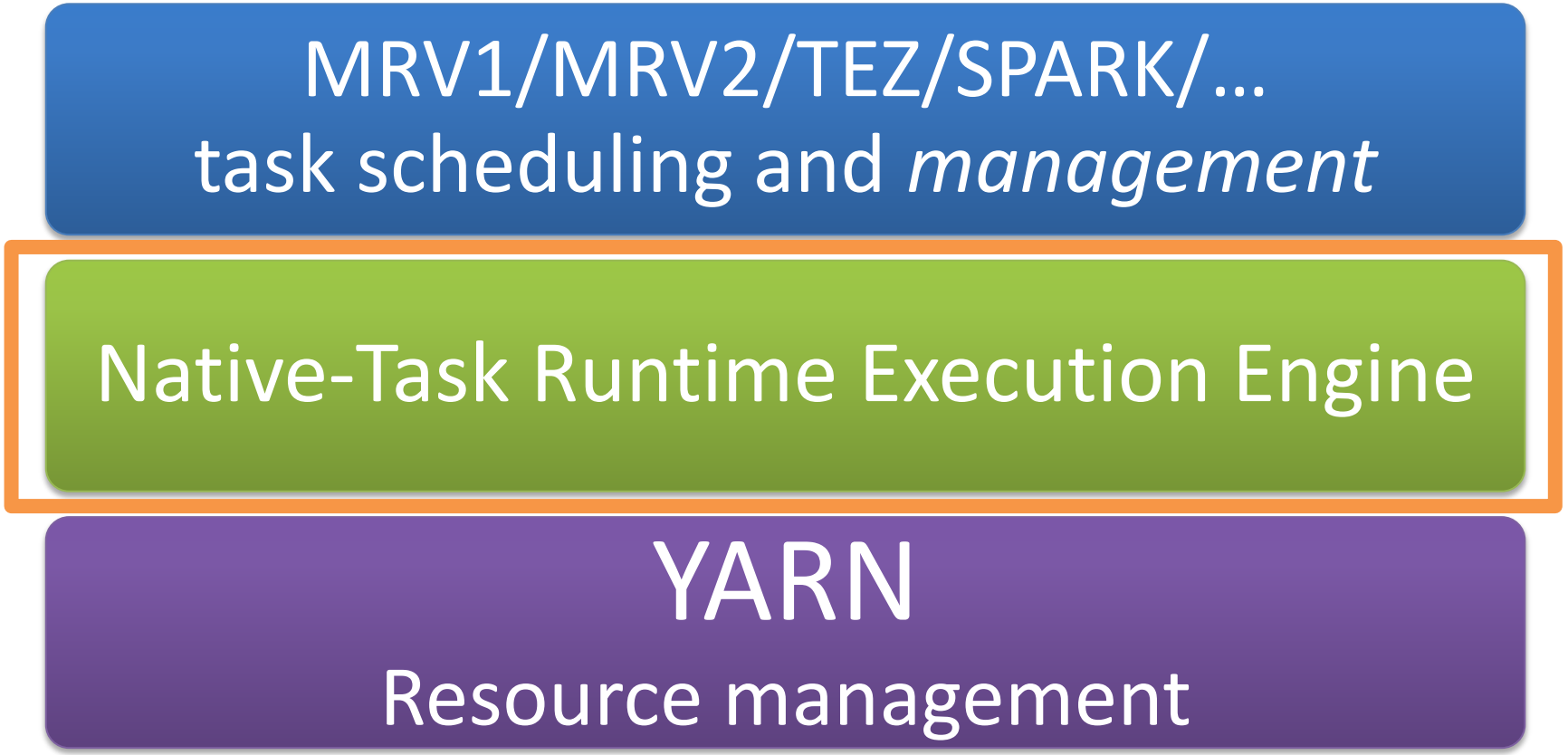
Full task optimization - Native Runtime

- **Native Runtime** is **General execution framework.** It can supports native mapper, native reducer, and …..
- **Native Runtime and Native Collector** are **two different modes** of Native-Task.
- Native Runtime is a **horizontal low level layer** under higher level Applications.
- **Native Runtime can be generic enough**, to host the **WHOLE** data processing workflow for **various upper layer App,** like Map-Reduce, TEZ.

Developer friendly

- The Native Runtime framework provides efficient C++ API and libraries.
- We can easily develop upper-level native applications or apply more aggressive optimizations.
- For example, we can build Hive on top of Native-Task.

Future of Native-Task



A horizontal layer under computation engine like MR.