Unit 13
Big Data Analytics

Characteristics of Many Modern OLAP Applications

- Datasets keep increasing in size
- Individual nodes can not process entire dataset efficiently
- Individual nodes can not store entire dataset efficiently
- Need to resort to distributed processing and storage
- For storage: distributed filesystems
  - E.g., HDFS with super-fast reads but only append writes
- Luckily for many applications “reading” is sufficient
  - No writing to datasets as part of application
  - Performed in a second step, separate system, if needed

Approaches

- Many approaches exist for storing and/or processing data
- Key is ***efficient mapping*** of data to underlying hardware infrastructure based on 1. structure of data, 2. functionalities offered, and 3. guarantees provided, e.g.,
  - 1: OO vs relational data vs (*key*, *value*) pairs
  - 2: arbitrary position read&write vs arbitrary position read&append only vs sequential read&append only
  - 3: isolation for sequence of operations vs individual operations only
- Gives rise to ***many*** different "big data" approaches and systems, e.g., MapReduce, Pig/PigLatin, Flume, Mahout, Hive, Samza, Apex, Ignite, Kafka, Hedwig, Storm, Heron, Flink, Spark
  - ***Understanding tradeoffs (1./2./3.) and system concepts is crucial for making the right choice***
  - Vs pure SE principles (e.g. lines of code, modularization)
- We will briefly discuss three:
  1. MapReduce
  2. Spark/RDDs
  3. Storm

MapReduce (a.k.a. Map/Reduce, Map-Reduce, …)

- Introduced originally in Lisp programming language

- Popularized for distributed processing by Google in 2004 [Dean&Ghemawat;OSDI'04]

- Allows for distribution with strong potential for parallelization

- Well-suited for data structured as (*key*, *value*) pairs

- Simple abstraction, programmer implements 2 functions:
  - $map(k_1, v_1) \rightarrow list(k_2, v_2)$
  - $reduce(k_2, list(v_2)) \rightarrow list(v_3)$

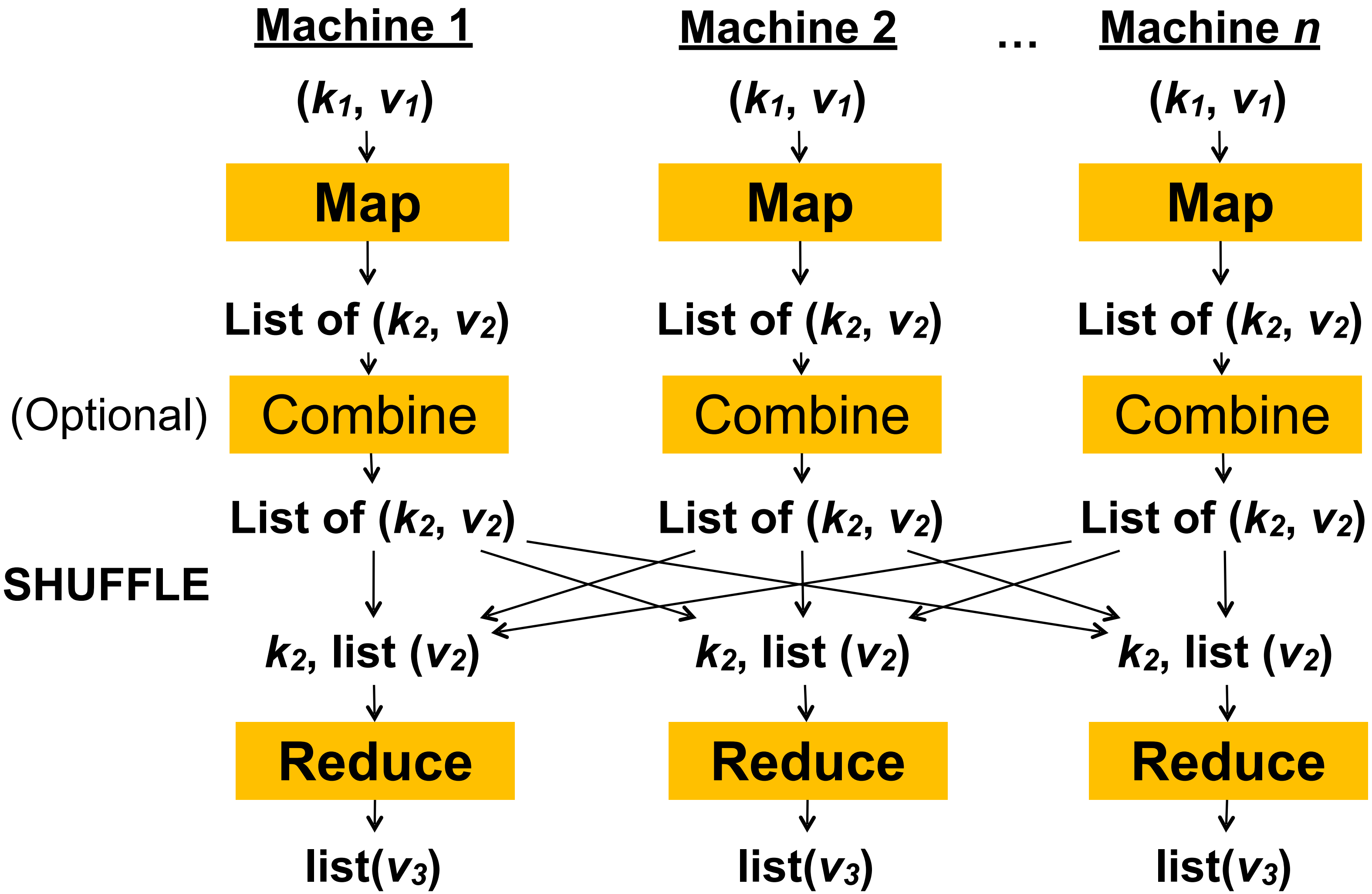
- Note: k_2s needn't be k_1s, v_2s needn't be v_1s, v_3s needn't be v_2s

Workflow

1. Data loaded in from file, *n* partitions created
2. *n* parallel mappers instantiated on *n* hosts, each obtains a partition
3. *map* function called for one partition entry (k_1, v_1) pair at a time, returns list of (k_2, v_2) pairs each time
4. Temporary files created with values v_2 per (same) key k_2
5. *m* parallel reducers instantiated on *m* hosts
6. Each reducer is assigned a subset of the keys k_2
7. *reduce* function called for a key k_2 at a time with *all* values from temporary files created for that key *by any* mapper, outputs list of values v_3 each time

- System phase taking care of 6. and distributing data to reducers is called “shuffling”
- System also responsible for monitoring “stragglers” (slow mappers or reducers) and re-initiating them in case of failures, as well as load balancing across hosts

Schematic Overview (Special Case of m=n)



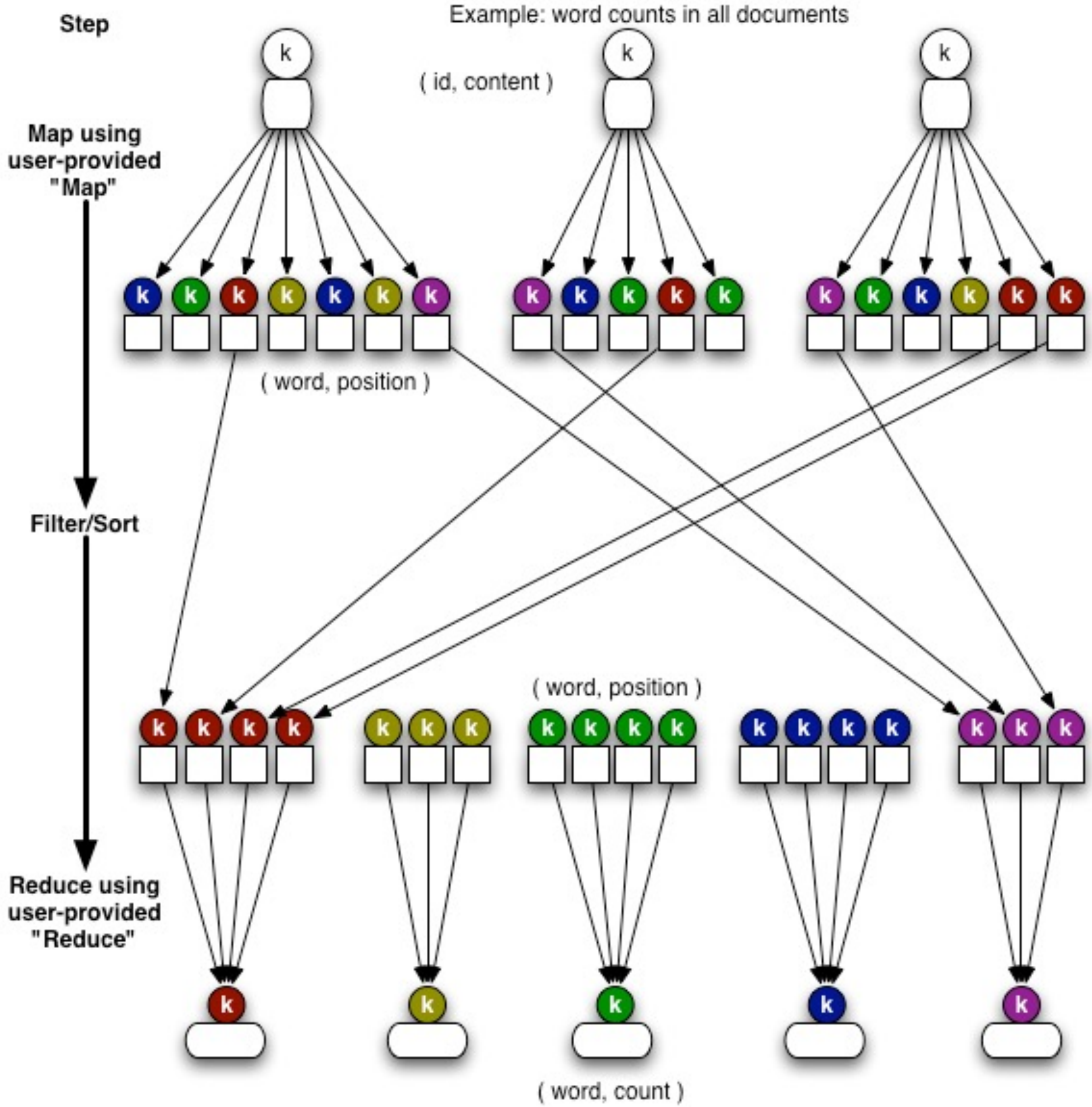
Example: Word Count (Pseudo-Code)

- Input: set of files/documents
- Output: words and the number of their respective occurrences

```
map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
        EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += ParseInt(v);
    Emit(AsString(output_key + “:” + result));
```

Word Count Illustrated



Limitations

- Model
  - Not all (parallelizable) tasks fit the abstraction
  - Can chain multiple ***stages*** with respective *map* and *reduce* functions
    - Still no unbounded iterative (and recursive) computation stages (cf. great^{x}-grantchildren/parents)

- Performance
  - Filesystem becomes bottleneck
  - Can avoid distributed filesystem between different stages, but shuffling still uses local filesystem

- Many extensions proposed (for model and/or performance), e.g., aggregators (reducer-side)

Spark

- Original paper published in 2012 [Zaharia et al.;NSDI'12]
- Key tenets
  A. ***Dataflow-based*** *computation* from input data sets to results through sequence of operations
     – Main abstraction: ***resilient distributed datasets (RDDs)***
     – No modifications in place, modifications give rise to new RDD
     – Computation forms a DAG with nodes RDDs and arcs for operations
  B. ***Avoid filesystem*** during computations
     – All datasets ***materialized*** in RAM including ***intermediate*** datasets
     – Support for ***iterative*** and ***incremental*** computations
  C. Scaling by ***partitioning datasets across hosts*** (RAM)
     – API highlights operations that can be performed on individual partitions
  D. ***Lineages*** for tracking dependencies between datasets
     – Used for incremental computations and fault-tolerance (recomputing)
- A. from data flow programming (e.g., PigLatin [Olsen et al.;SIGMOD'08]), B. from main-memory DBMSs (e.g., [Pedone&Frolund;SRDS'00]), C. from distributed DBMSs, D. from snapshot techniques and provenance tracking (e.g., [Zhang et al.;VLDB'07])

Operations on RDDs

1. Transformations
   - Create RDDs from other (parent) RDDs
   - Typically one element at a time (and thus independently across partitions)
   - Some common transformations
     – *map*(*function*): *function* applied to every element to create new element
     – *filter*(*function*): *function* applied to every element to decide whether to “keep” it
2. Actions
   - Compute actual return values
   - Some common actions
     – *count*(): return the number of elements
     – *take*(*n*): return an array of first *n* elements
     – *collect*(): return array of all elements
     – *saveAsTextFile*(*file*): save to *file* in filesystem
     – *reduce*(*function*): aggregate all values using *function*

◆ Many alternative APIs and libraries, e.g.,
   - DataFrame (tables w/ named columns), Dataset (static typing for OOP)
   - SparkSQL
   - Spark streaming (discretized streams)
   - SparkML/MLlib (on DataFrame)

Examples (Pseudo-Code)

- Word count

```
counts = sc.textfile("hdfs://…")
           .flatMap(lambda line: line.split('\s'))
           .map(lambda word: (word, 1))
           .reduceByKey(operator.add)
counts.save("hdfs://…")
```

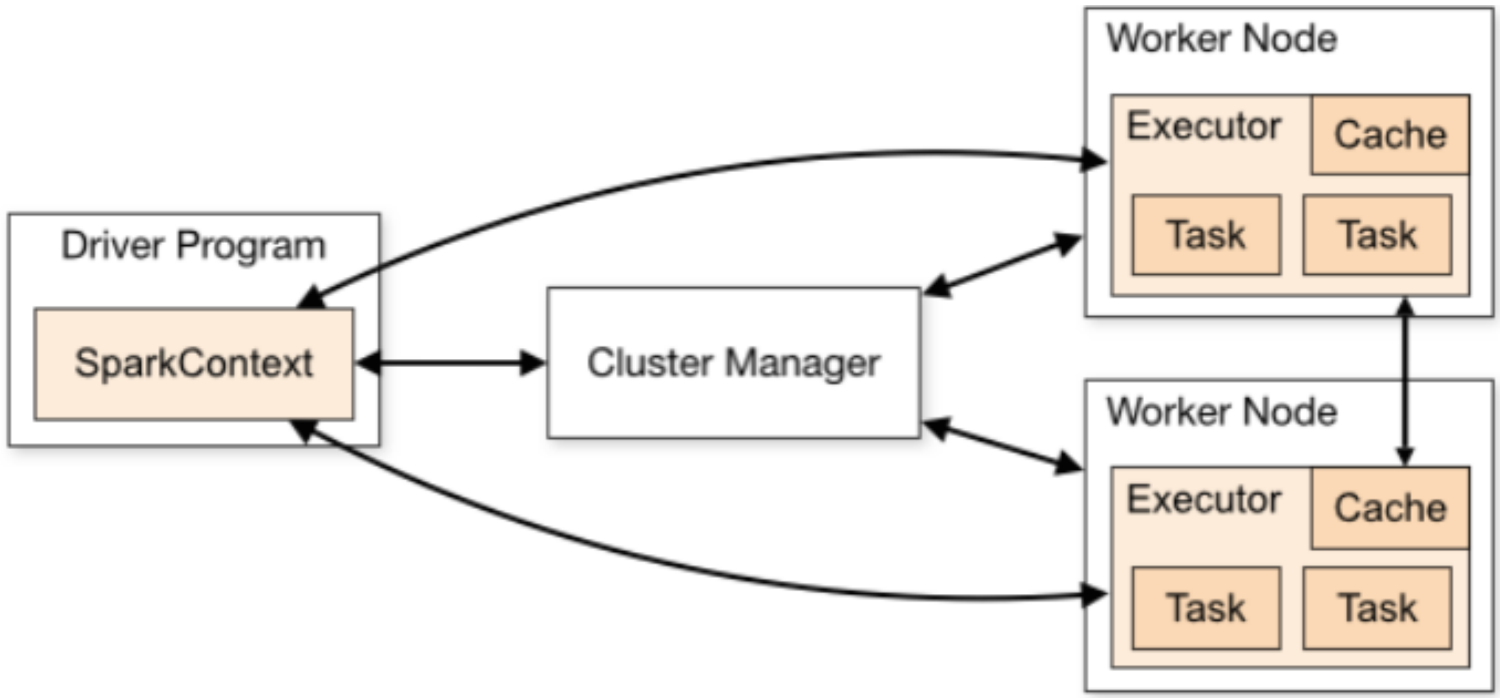
- General MapReduce

```
result = data.flatMap(map_fn)
           .groupByKey()
           .map(lambda (k, vs): reduce_fn(k, vs))
```

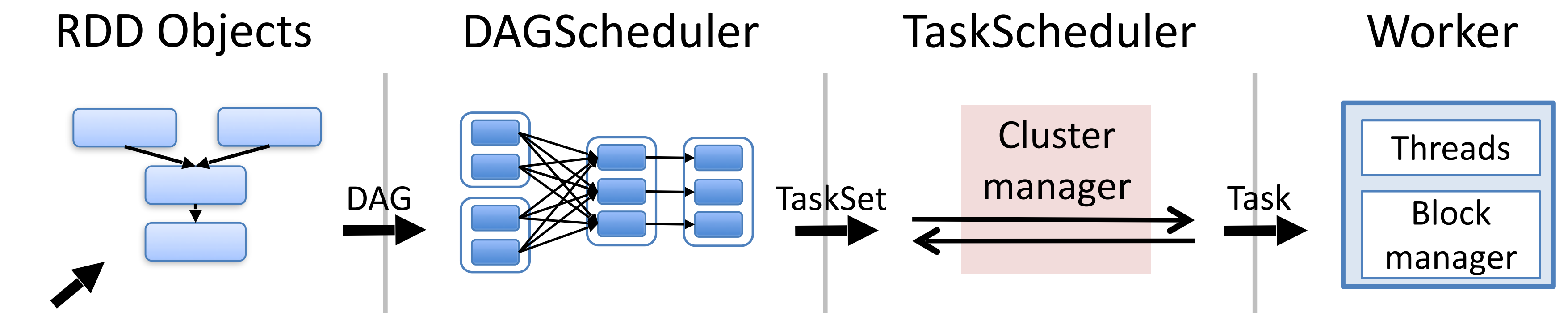
- General MapReduce with combiner

```
result = data.flatMap(map_fn)
           .reduceByKey(combiner_fn)
           .map(lambda (k, vs): reduce_fn(k, vs))
```

Architecture Overview



Tracing Execution




```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
```

build operator DAG

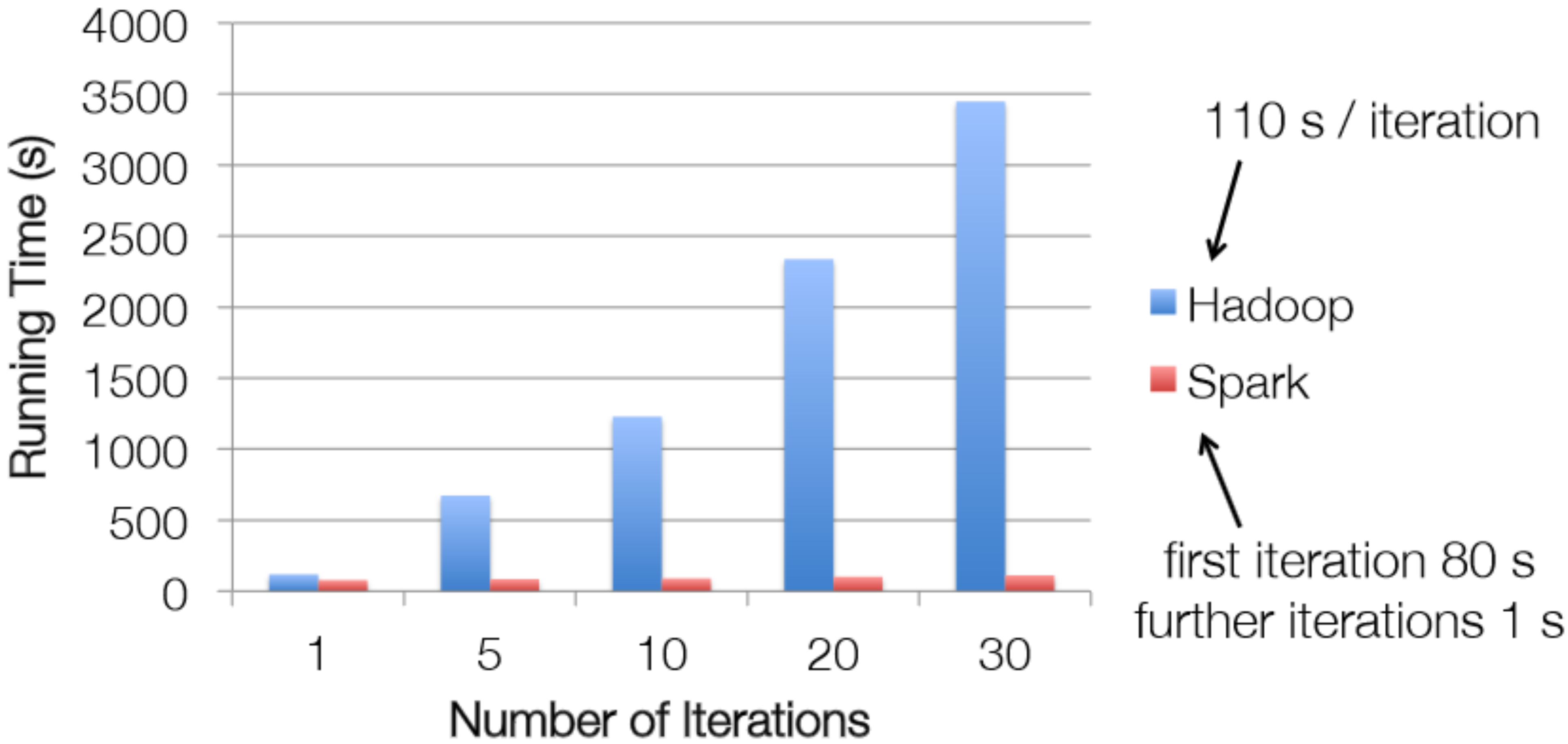
split graph into *stages* of tasks
submit each stage as ready

launch tasks via cluster manager
retry failed or straggling tasks

execute tasks

store and serve blocks

MapReduce vs Spark: Logistic Regression



Limitations

- **Aggregation**
  - Performance excellent when “embarrassingly parallel”
  - Aggregation performance depends on aggregation function used
    - Sometimes best in many stages 2 x 2 x 2…. (e.g., HP Presto [Venkataraman et al.;EuroSys’13])
    - Sometimes best in 1 stage *n* x 1 (e.g., Spark original)
    - Often in between, depending on ***aggregation ratio*** (tradeoff computation vs communication)
  - Cf. LOOM [Culhane et al.;HotCloud’14]*,[Culhane et al.;INFOCOM’15], ROME [Blöcher et al.;ACM TOCS’22]
    - `treeReduce` (added after *) only poor subset solution using ***depth*** parameter
- **Security**
  - Cf. Seabed [Papadimitriou et al.;OSDI’16], Cuttlefish [Savvides et al.;SoCC’17], Opaque [Zheng et al.;NSDI’17], Symmetria [Savvides et al.;VLDB’20], Hydra [Mangipudi et al.;PLDI’23]
- **Correctness**
  - Cf. Multi-party session types for event-driven fault-tolerant distributed programming [Viering et al.;OOPSLA’21]

…

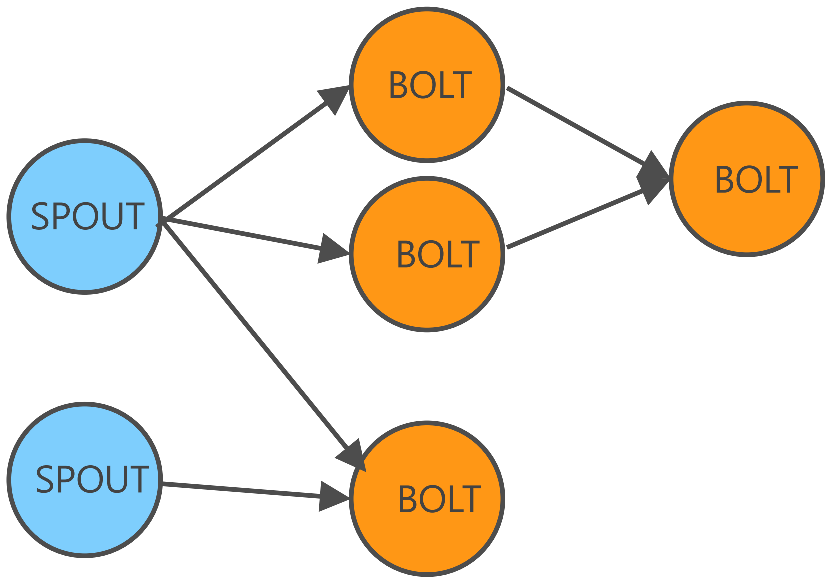
Another Issue: Reactivity

- Spark et al. are optimized for batch/mini-batch processing
- Support for incremental computation upon extension of input datasets

- What if arrival of new data is the norm and reaction times need to be minimized?
- Enter ***stream processing***
- Scenarios
  - Financial applications (e.g., electronic/algorithm trading, fraud detection)
  - Network monitoring
  - Social network analysis
  - Sentiment analysis on tweets
  - …
- Stream processor listens indefinitely for incoming data
  - Reacts

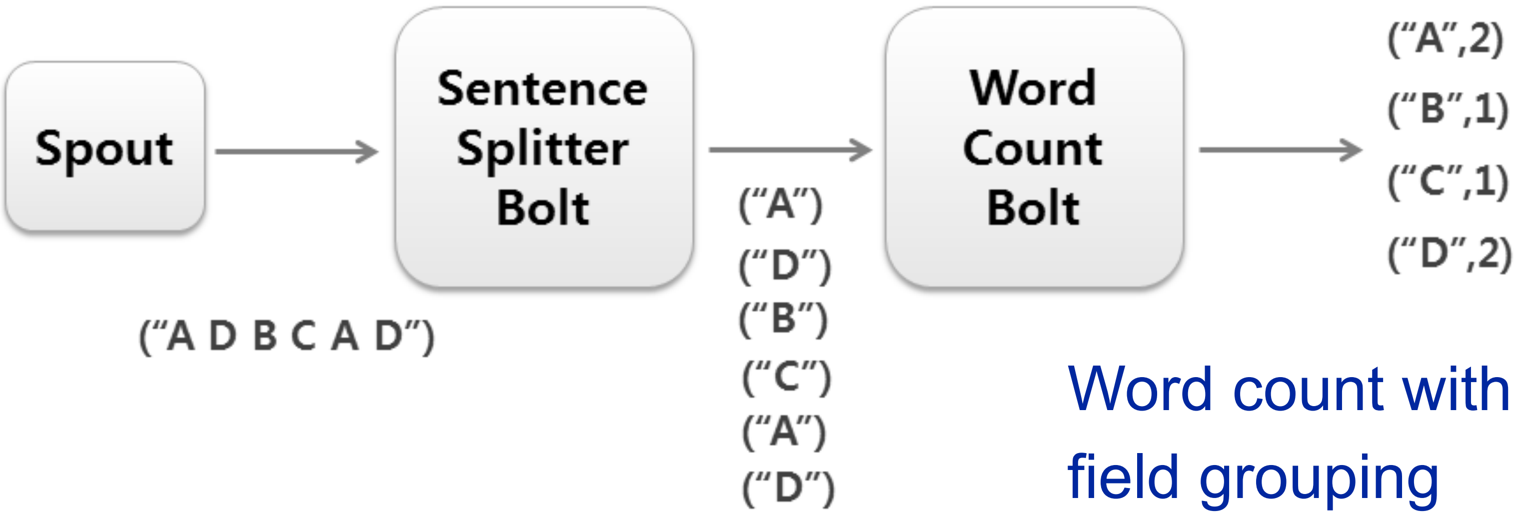
Storm

- Relies on explicit application-defined topology
- Concepts
  - ***Tuple***: ordered list of named values
  - ***Stream***: an unbounded sequence of tuples
  - ***Spout***: source of a stream emitting tuples
  - ***Bolt***: accepts tuple from (one if its) input streams, performs some computation (filtering, aggregation, join), possibly emits new tuple(s)



Parallelism

- Bolts can be replicated
- ***Groupings*** specify how tuples are routed to bolt replicas
  - Shuffle grouping: random distribution
  - Field grouping: portioning according to value of tuple attribute
  - All grouping: complete replication
  - Direct grouping: producer decides on replica
  - …
  - Custom application-defined grouping



Word count with field grouping

Key Ideas

- Distributed data analytics
- MapReduce
- Spark
- Limitations/open issues
- Stream processing