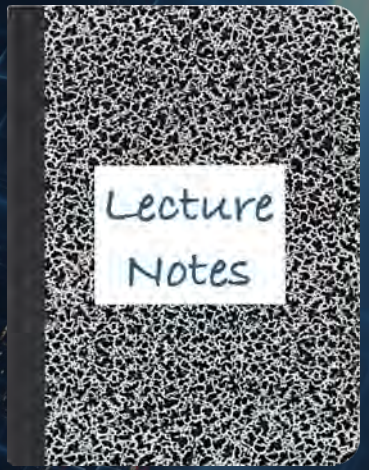
CS 417 – DISTRIBUTED SYSTEMS



Week 10: Scalable Distributed Computation

Parallel Data Processing Beyond Storage

Paul Krzyzanowski

Distributed Computation

1. **MapReduce**
   Foundational Batch Model
2. **BSP & Pregel**
   Iterative and graph computation
3. **Apache Spark**
   Generalized distributed dataflow
4. **Distributed ML**
   What changes when models are large

From Storage to Computation

Large-scale storage is solved. Now what?

The Scale Problem

100 ms per item
1 billion items

= **over 3 years** on one machine

The only answer is parallelism.

Divide the work. Execute concurrently.

What Makes It Hard

- Some problems need data regrouped by key
- Some need to exchange state across workers
- Some need repeated passes over the same data
- Failures are normal, not exceptional
- One slow worker delays the whole job

Six Challenges Every Framework Must Address

Partitioning

Divide input so workers can operate concurrently

Locality

Run code where the data already lives — don't move huge datasets

Communication

Regroup data by key, propagate updates, merge partial results

Synchronization

Decide when workers must wait before proceeding to the next step

Fault Tolerance

Recover from crashes without restarting the whole job

Skew

Prevent one heavy partition or slow worker from bottlenecking everything

MapReduce

Large-scale batch computation at Google scale

Why MapReduce Was Created

The Problem at Google (early 2000s)

- Web indexing, log analysis, data products
- All these jobs had the same broad form
- Every team wrote their own custom distributed code
- Most of the program was infrastructure, not logic
- Infrastructure handles: workers, partitioning, scheduling, recovery

Key Insight

Restrict the structure of computation.

Give the runtime enough control to manage the cluster automatically.

The programmer writes only the logic.

The framework handles everything else.

MapReduce

Created by Google in 2004

Jeffrey Dean and Sanjay Ghemawat

Inspired by LISP

Map(function, set of values)

- Applies function to each value in the set

```
(map 'length '(() (a) (a b) (a b c))) ⇒  (0 1 2 3)
```

Reduce(function, set of values)

- Combines all the values using a binary function (e.g., +)

```
(reduce #'+ '(1 2 3 4 5)) ⇒ 15
```

MapReduce – a high-level view

Map:

- Grab the relevant data from the source
- User function gets called for each chunk of input
- Spits out (key, value) pairs

Reduce:

- Aggregate the results
- User function gets called *for each unique key* with all values corresponding to that key

MapReduce

Map: (input shard) → intermediate(key/value pairs)

- Framework partitions input data into *M* shards
- The user's ***map*** function discards unnecessary data and generates ***(key, value)*** sets
- The framework groups together all intermediate values with the same key & passes them to the ***reduce*** function

Reduce: intermediate(key/value pairs) → result files

- Called once for each unique key (that was generated by *map*)
- Input to reduce: key & set of all values associated with that key
- The user's reduce function processes these values (often doing things like counting)

Keys are distributed to ***reduce*** workers are by partitioning the intermediate key space into *R* pieces using a partitioning function (default = *hash(key) mod R*)

- The user specifies the # of partitions (R) and, optionally, the partitioning function

MapReduce Operations

```
map(data_item)   →   list(k, v1)
reduce(k, list(v1))   →   list(v2)
```

The shuffle-and-sort step is the center of MapReduce.

All pairs with the same key are gathered together before any reducer sees them.

MapReduce: what happens in between?

- Map
  - Grab the relevant data from the source (parse into key, value)
  - Write it to an intermediate file
- Partition
  - Partitioning: identify which of *R* reducers will handle which keys
  - Map partitions data to target it to one of *R* Reduce workers based on a partitioning function (both *R* and partitioning function user defined)

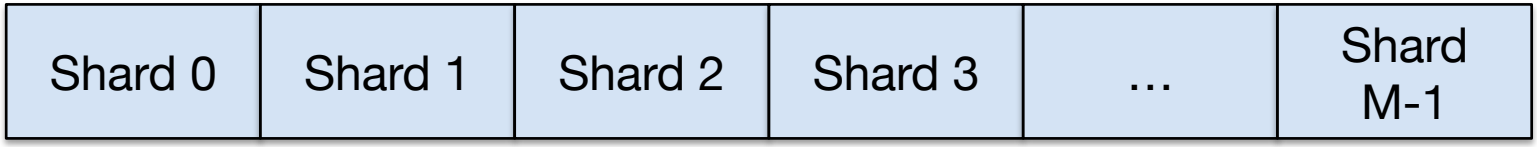
Map Worker

- Shuffle & Sort
  - Shuffle: Fetch the relevant partition of the output from all mappers
  - Sort by keys (different mappers may have sent data with the same key)
- Reduce
  - Input is the sorted output of mappers
  - Call the user *Reduce* function per key with the list of values for that key to aggregate the results

Reduce Worker

Step 1: Split input files into chunks (shards/splits)

Break up the input data into *M* pieces
(typically 64 MB to match GFS chunk size)

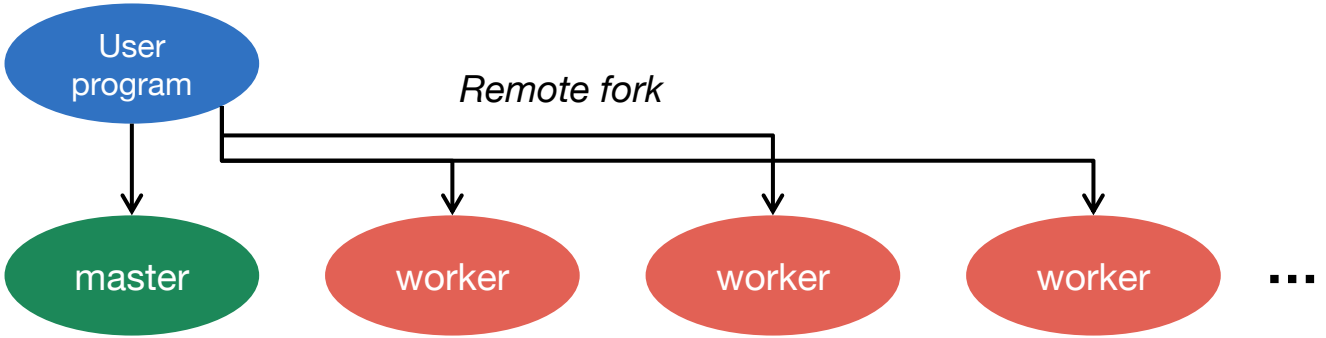


Input data

Divided into *M* shards (splits)

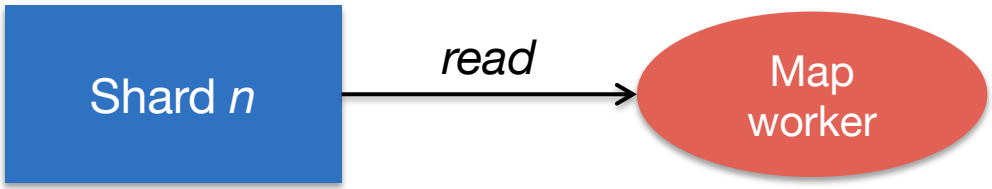
Step 2: Fork processes

- Start up many copies of the program on a cluster of machines
  - **One master**: scheduler & coordinator
  - Lots of workers
- Idle workers are assigned either:
  - *map* tasks (each works on a shard) – there are *M* map tasks
  - *reduce* tasks (each works on intermediate files) – there are *R* tasks
    - *R* = # partitions, defined by the user



Step 3: Run Map Tasks

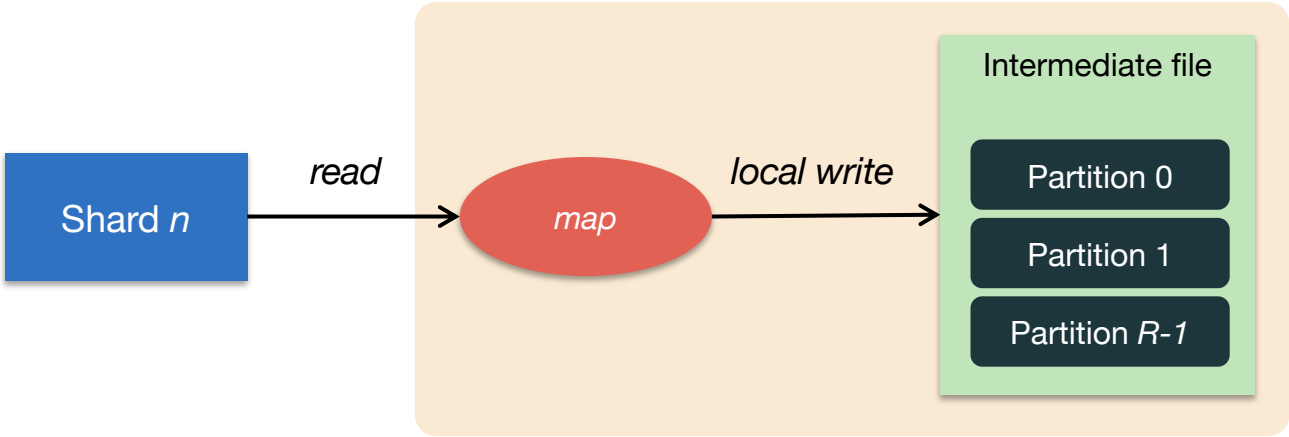
- Reads contents of the input shard assigned to it
- Parses key/value pairs out of the input data
- Passes each pair to a user-defined map function
  - Produces intermediate key/value pairs
  - These are buffered in memory



MapReduce frameworks support multiple input formats:
Input data may be a single file, directories of files, database query results, and various data formats, such as binary, text, line-oriented text, and key-value pairs

Step 4: Create intermediate files

- Intermediate key/value pairs produced by the user's *map* function buffered in memory and are periodically written to the local disk
  - Partitioned into *R* regions by a partitioning function
- Notifies master when complete
  - Passes locations of intermediate data to the master
  - Master forwards these locations to the reduce worker



Step 4a. Partitioning

- Map key-value data will be processed by *Reduce* workers
  - The user's Reduce function will be called once per unique key generated by Map.
- We first need to group all the *(key, value)* data by keys and decide which Reduce worker processes which set of keys
  - The Reduce worker will later sort the values within each keys

Partition function

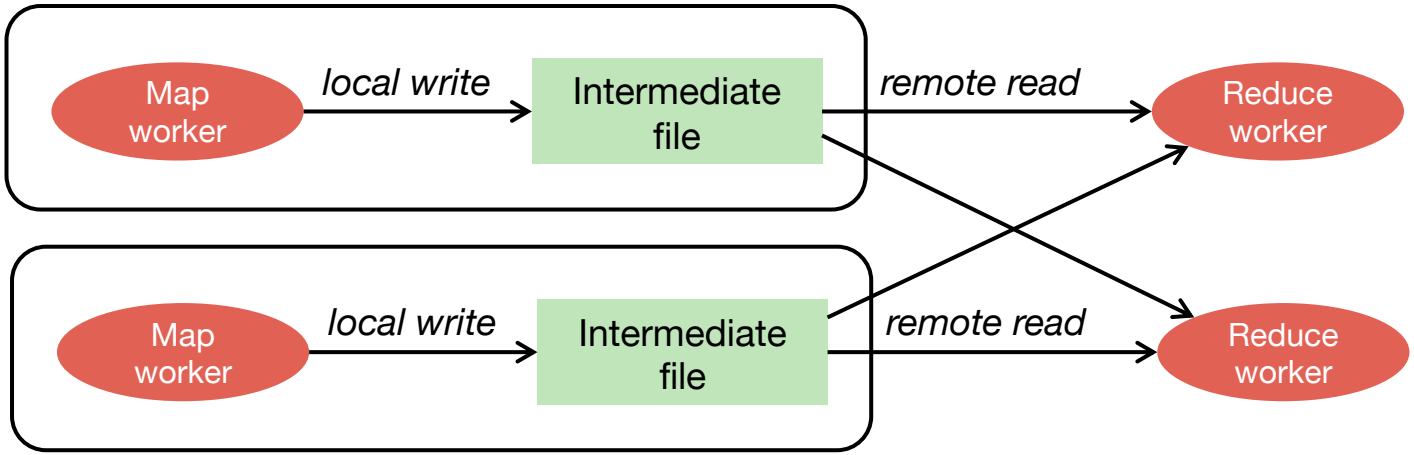
Decides which of R reduce workers will work on which keys

  - Default function to identify a reduce worker: *hash(key) mod R*
  - Map worker partitions the data by groups of keys for each *Reduce* worker
- Each *Reduce* worker will later read their partition from every *Map* worker

Step 5: Reduce Task: Shuffle & Sort

Reduce worker is notified by the master about the location of intermediate files for its partition

- **Shuffle**: Uses RPCs to read the data from the local disks of the *map* workers
- **Sort**: When the *reduce* worker gets all the (*key, value*) data for its partition from all workers
  - It sorts the data by the keys
  - All occurrences of the same key are grouped together

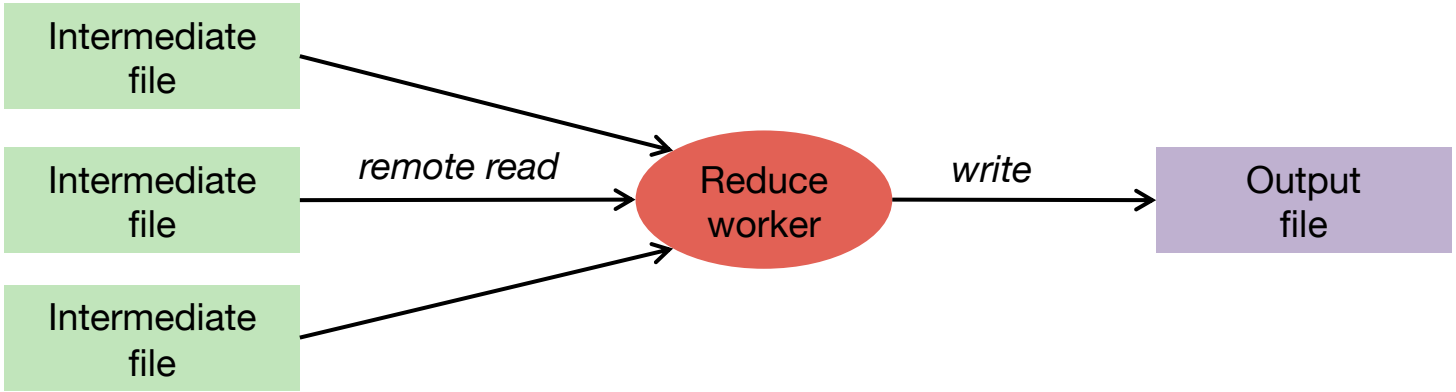


Step 6: Reduce Task: *Reduce*

- The *sort* phase grouped data by keys
  - This makes it easy to identify all the values from all the map workers that are associated with each key
- The user's ***Reduce*** function is given the key and the set of intermediate values for that key

< key, (value1, value2, value3, value4, …) >

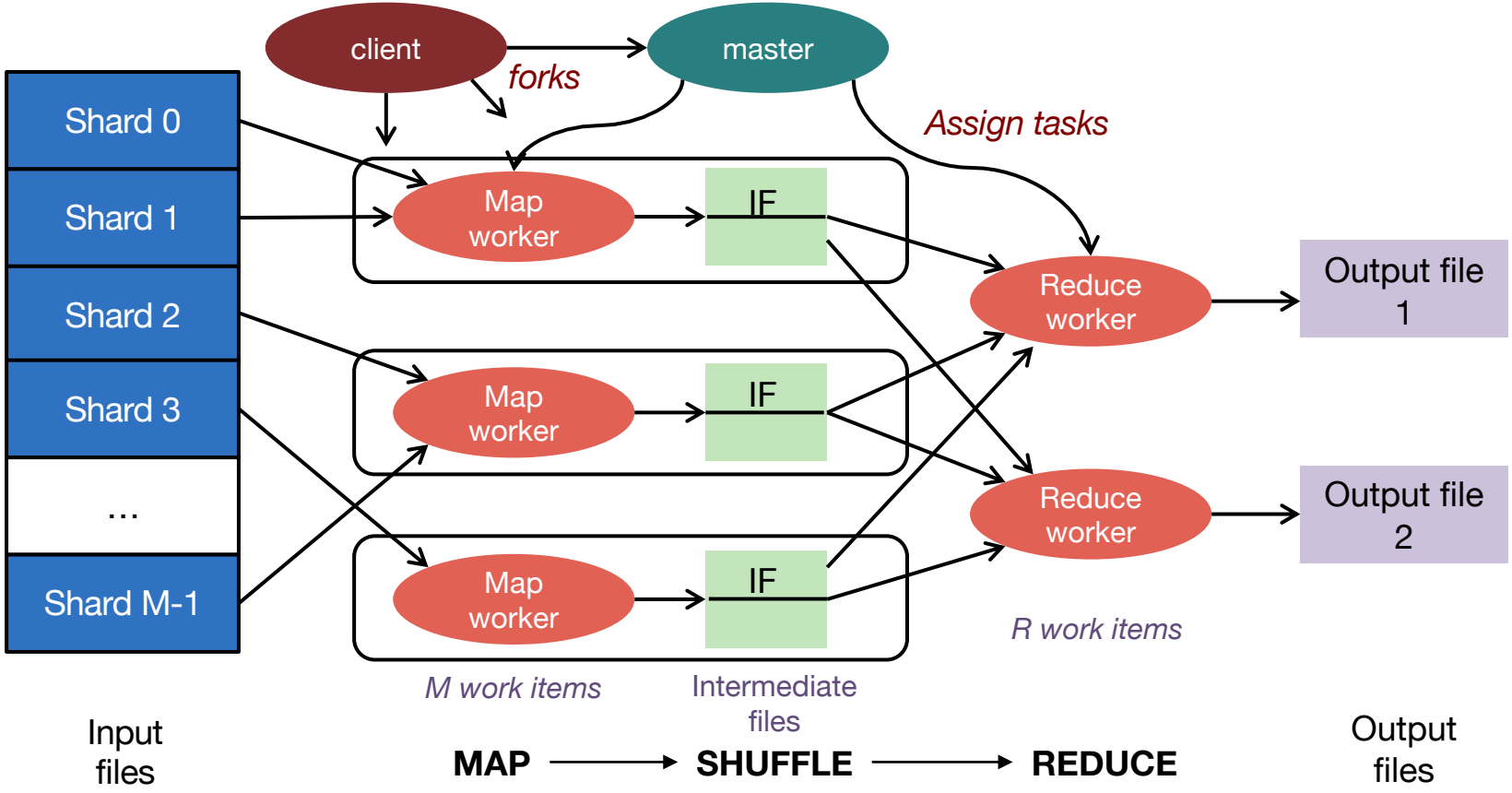
- The output of the *Reduce* function is appended to an output file



Step 7: Return to user

- When all *map* and *reduce* tasks have completed, the master wakes up the user program
- The *MapReduce* call in the user program returns and the program can resume execution
- Output of *MapReduce* is available in *R* output files

MapReduce: the complete picture



Word Count — The Classic Example

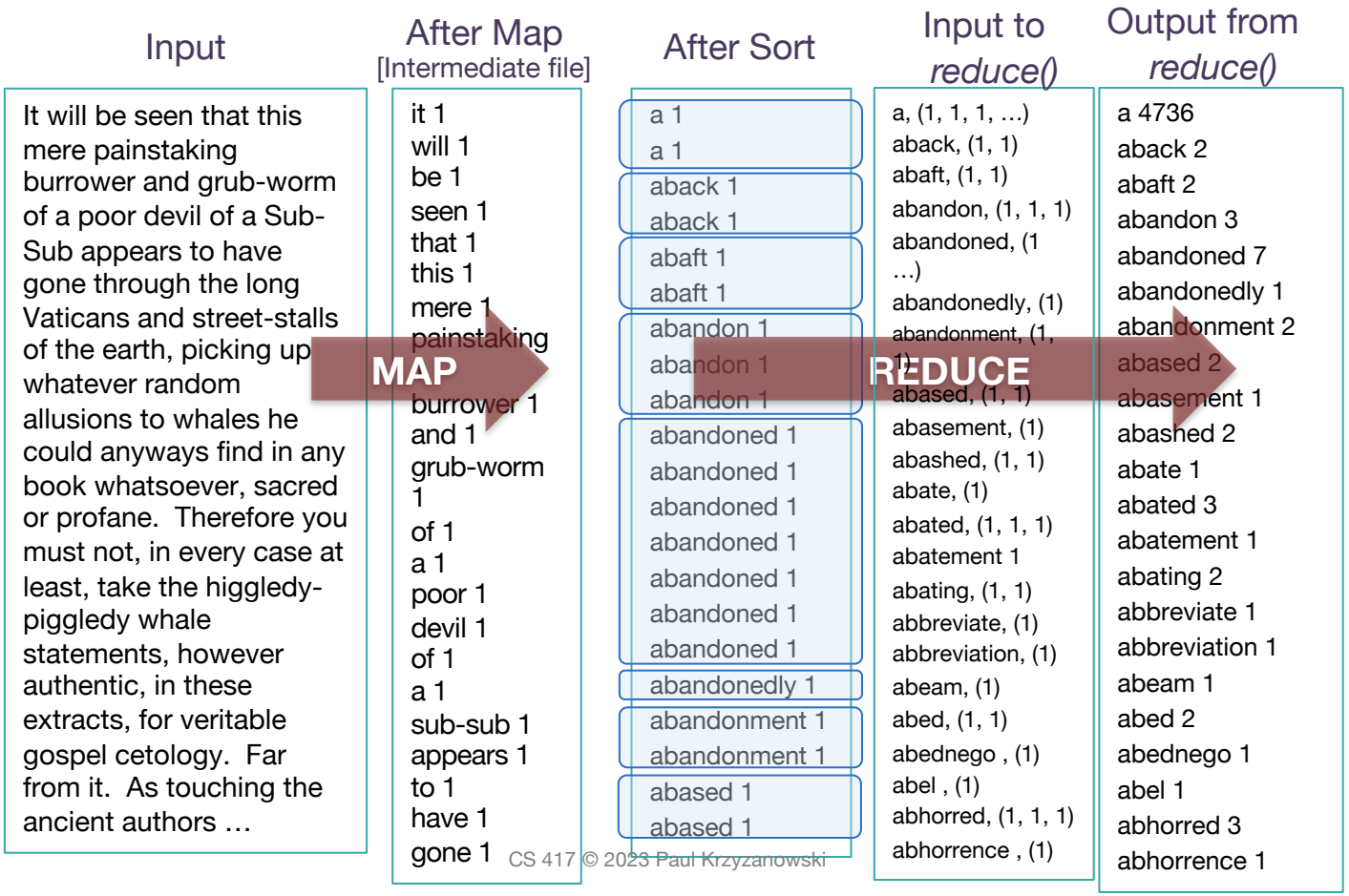
```
map(_, text):
    for each word w in text:
        emit(w, 1)
```

```
reduce(word, counts):
    total = 0
    for c in counts:
        total += c
    emit(word, total)
```

What happens in between

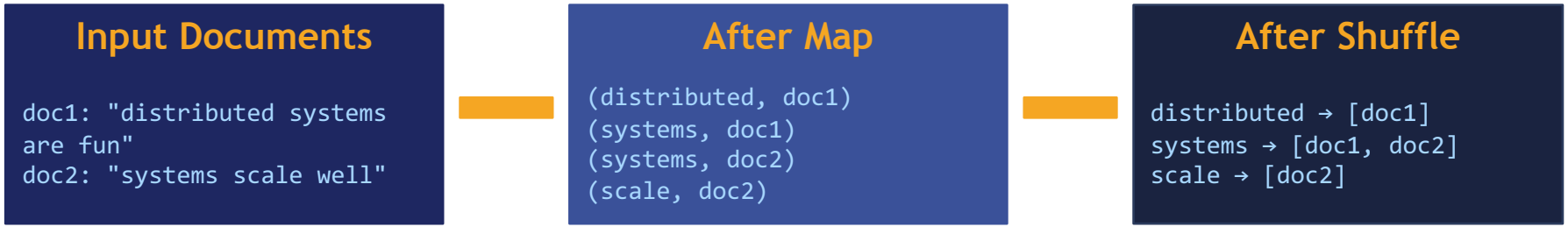
1. **map** emits:
   ("system", 1), ("system", 1), ("system", 1) ...
2. **shuffle** brings all "system" pairs together
3. **reduce** receives: system → [1, 1, 1]
4. reduce emits: (system, 3)

Example: Word Count



Inverted Index: A Realistic Example

Map emits (term, docID).
The shuffle reorganizes data from storage-based to key-based ownership.




```
map(docID, text):
    for each term t in text:
        emit(t, docID)
```

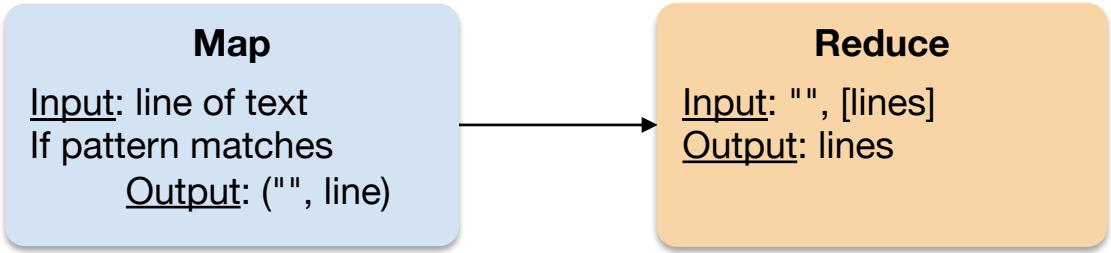
```
reduce(term, docIDs):
    emit(term, unique_sorted(docIDs))
```

The shuffle is the step that changes data's organization from storage-based to key-based.

Other Examples: Search

Distributed grep (search for words)

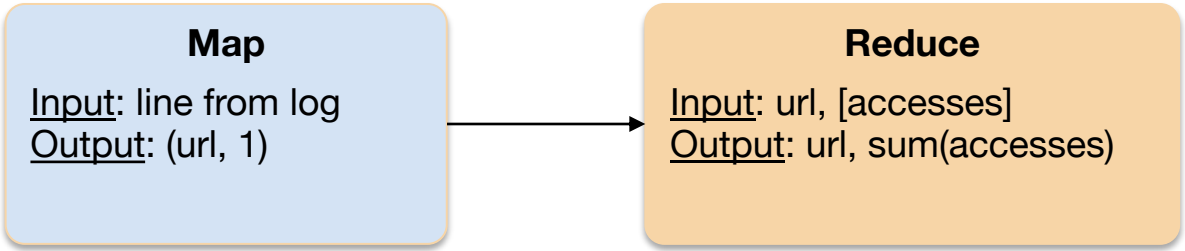
- *Search for words in lots of documents*
- Map: emit a line if it matches a given pattern
- Reduce: just copy the intermediate data to the output



Other Examples: URL access counts

List URL access counts

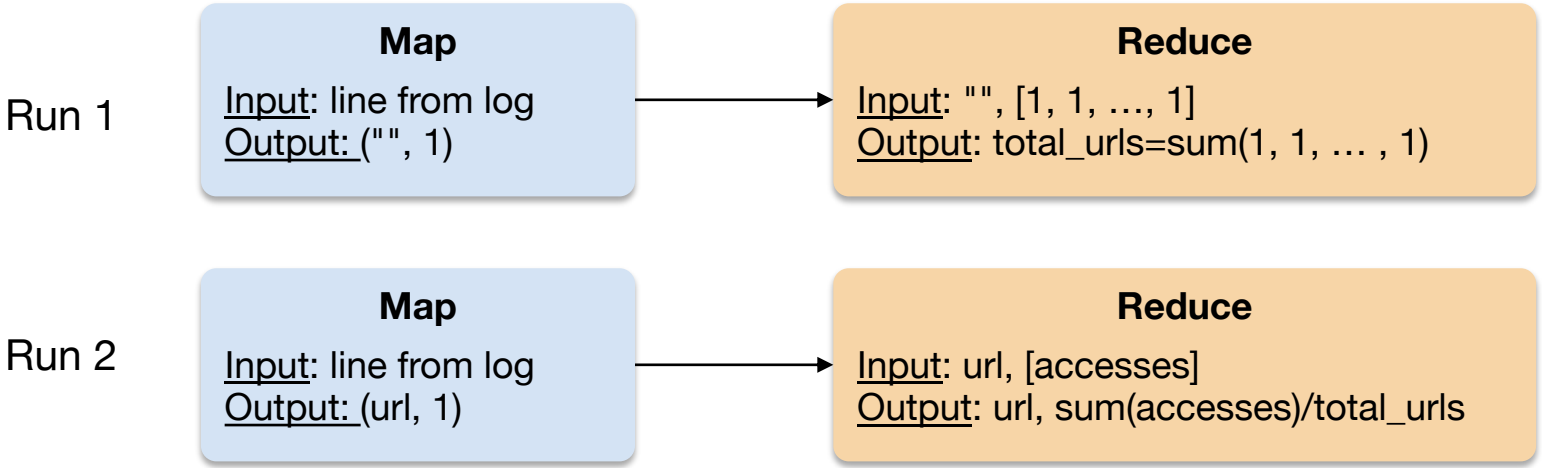
- *Find the count of each URL in web logs*
- Map: process logs of web page access; output <URL, 1>
- Reduce: add all values for the same URL



Other Examples: URL access frequency

Count URL access frequency

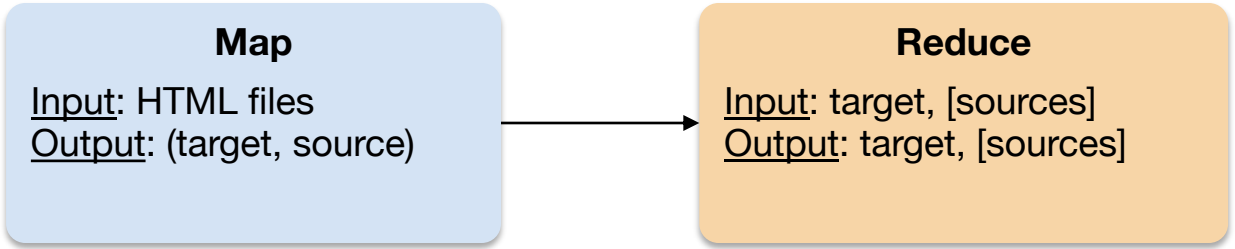
- *Find the frequency of each URL in web logs*
- Run 1: just count total URLs
- Run 2: just like URL count but now we stored **total_urls**



Other Examples: Reverse Links

Reverse web-link graph

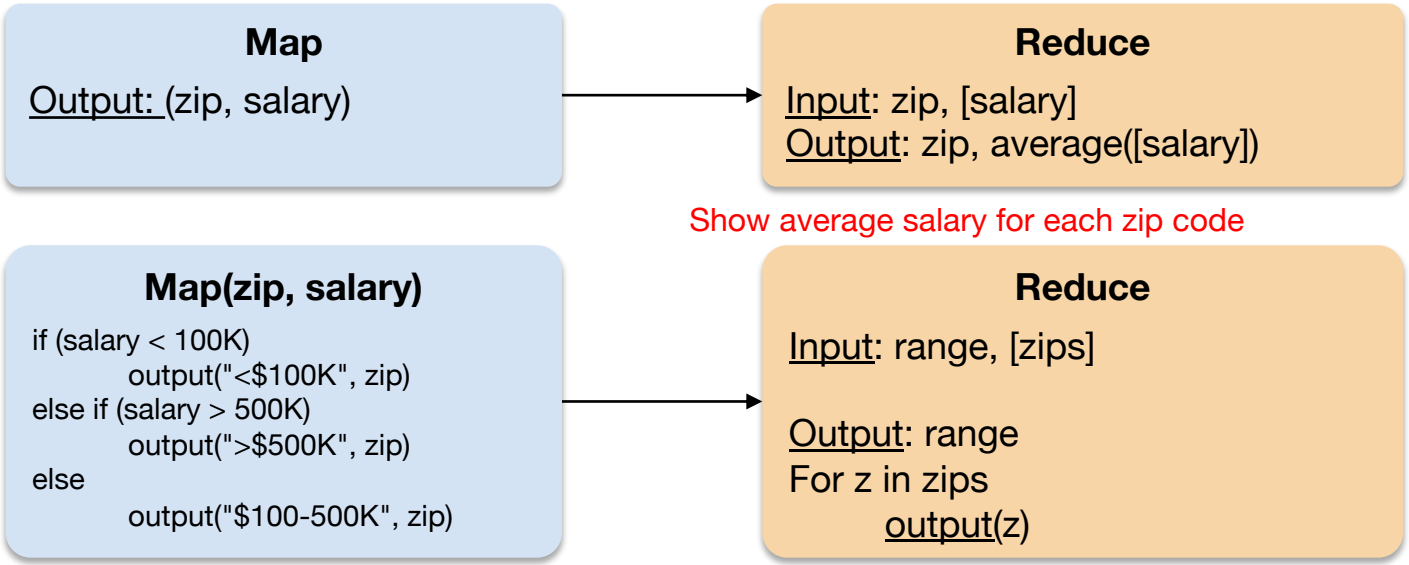
- *Find where page links come from*
- Map: output <target, source> for each link to *target* in a page *source*
- Reduce: concatenate the list of all source URLs associated with a target
  Output <target, list(source)>



Other Examples: Two rounds of MapReduce

Average salaries in regions

- *Show zip codes where average salaries are in the ranges:*
  (1) < $100K (2) $100K … $500K (3) > $500K
- Data is a set of lines: { name, age, address, zip, salary }



Execution Model: *M* Mappers, *R* Reducers

Execution sequence

1. Split input into M shards
2. Launch workers, assign map tasks
3. Mappers read shards, **emit (k, v)** pairs
4. Each mapper partitions output into R buckets: **hash(key) mod R**
5. Reducers fetch their buckets from all mappers
6. Each reducer sorts by key, groups identical keys
7. Reduce function runs once per key
8. Each reducer writes one output file

Locality

Schedule map tasks on the machine that holds the input shard. Moving code is cheap. Moving terabytes is not.

Fault Tolerance

Map and reduce tasks are deterministic and side-effect free.

A failed task is simply re-executed on another worker. No checkpoint infrastructure needed.

Shuffle Cost and Stragglers

Execution sequence

1. Write intermediate data to local disk
2. Partition by reducer assignment
3. Transfer partitions across the network
4. Merge at reducers
5. Sort by key
6. Group equal keys

Stragglers

A job finishes only when its last task finishes

At scale, something is always slow.

Slow tasks are called **stragglers**.
Causes include:

- Contested CPU or I/O on a busy machine
- Slow or degraded disk
- Unusually heavy data partition

Fix: Speculative execution

- If a task runs unusually long, launch a duplicate on another worker
- Use whichever result finishes first

Fault Tolerance

- **Mapper fails** → re-run on another worker
  - Input still in HDFS; intermediate output regenerated
- **Reducer fails** → re-run on another worker
  - Reads from mapper output that is still available
- Master detects failures via heartbeat timeouts
  - Re-queues all tasks on a silent worker

Works because tasks are deterministic and side-effect-free

- Re-execution is safe: *same function + same input = same output*
- A failed mapper may force a re-run of reducers that already pulled from it

Why MapReduce Was Important

Lowered the barrier

Engineers without deep distributed systems expertise could write reliable large-scale jobs.

Established the vocabulary

Partitioning, *shuffle*, *speculative execution*

These patterns influenced every subsequent framework.

Spread through open source

- Apache Hadoop MapReduce
- Amazon EMR
- Google Cloud Dataproc

The ideas became a standard.

MapReduce solved the problem of making large-scale batch computation accessible and reliable

Limitations of MapReduce

Iterative Computation

Each round = new job

Full disk read + shuffle + disk write per iteration.

100-round ML algorithm pays that cost 100 times.

Graph Processing

Graph algorithms repeatedly propagate state along edges.

Rebuilding graph structure as key-value data every round is expensive and unnatural.

Multi-Stage Pipelines

Real analytics have 5–10 stages.

Chaining jobs means a full disk round-trip between every stage.

These limitations directly motivated other frameworks

BSP and Pregel

Why Iterative Computation Needs a Different Model

MapReduce approach to iterative computation (e.g. PageRank):

```
Round 1: read from HDFS → map → shuffle → reduce → write to HDFS
```

```
Round 2: read from HDFS → map → shuffle → reduce → write to HDFS
```

```
Round 3: read from HDFS → map → shuffle → reduce → write to HDFS
```

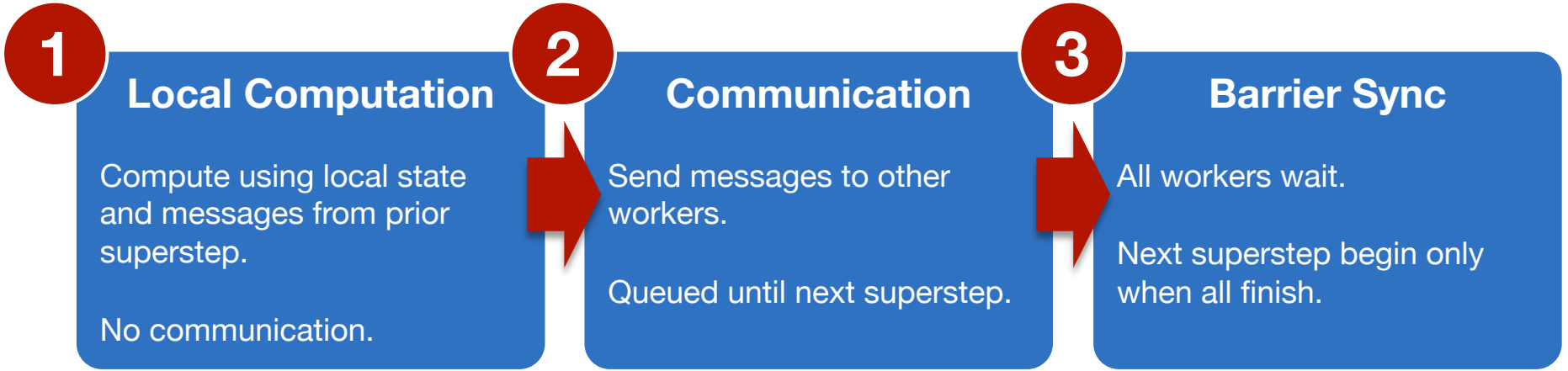
```
... repeated for every iteration ...
```

Each round pays full I/O cost — even though most of the data has not changed

50 iterations: 50 HDFS reads, 50 shuffles, 50 HDFS writes

BSP avoids this by keeping state in memory across rounds.

Bulk Synchronous Parallel (BSP)



→ Repeat until termination condition is met

Cost

Fast workers wait at the barrier.
Load imbalance is fully visible as wasted time.

Benefit

Natural checkpoint boundaries. State in memory across rounds. Deterministic message delivery.

Barriers create clean checkpointing boundaries:

- After N supersteps: save vertex state to stable storage
- Machine failure → rollback to last checkpoint, replay forward
- Much simpler than async recovery: no in-flight message tracking
- Tradeoff: checkpoint write cost vs. replay cost vs. failure rate
- BSP's deterministic delivery makes it easy to identify a consistent rollback point

Graph Algorithms: The MapReduce Problem

Router and network topology

Social networks (friends, follows)

Citation graphs

Program dependency graphs

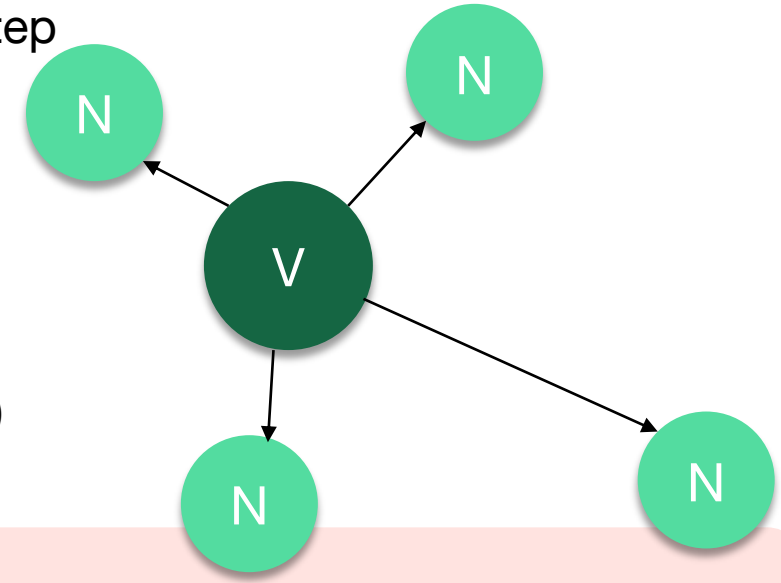
What graph algorithms need

Keep graph structure in memory across iterations. Let vertices exchange messages directly along edges, round by round. Avoid rebuilding from scratch every round.

Pregel: Think Like a Vertex

Each active vertex runs the same function every superstep:

- Receive messages from previous superstep
- Update own state and edge values
- Send messages to neighbors
  (arrive next superstep)
- Vote to halt
  (reactivated on receiving a new message)



Termination

All vertices have voted to halt AND no messages are in transit.
Both conditions must hold simultaneously.

Example: Single-Source Shortest Path

```
if vertex is source and superstep == 0:
    vertex.distance = 0
    for each edge (v → neighbor, weight):
        send(neighbor, weight)

else:
    best = min(incoming_messages)
    if best < vertex.distance:
        vertex.distance = best
        for each edge (v → neighbor, weight):
            send(neighbor, best + weight)
vote_to_halt()
```

Source starts at 0

Propagate initial distances

Take the best candidate

Improved? Update and propagate.

No improvement? Vote to halt
(Reactivated if better path arrives.)

The framework handles scheduling, distribution, and fault recovery invisibly.

Simple example: find the maximum value

Semi-pseudocode:

```
class MaxValueVertex
    : public Vertex<int, void, int> {
  void Compute(MessageIterator *msgs) {
    int maxv = GetValue();
    for (; !msgs->Done(); msgs->Next())
        maxv = max(msgs.Value(), maxv);

    if (maxv > GetValue()) || (step == 0)) {
        *MutableValue() = maxv;
        OutEdgeIterator out = GetOutEdgeIterator();
        for (; !out.Done(); out.Next())
            sendMessageTo(out.Target(), maxv)
    } else
        VoteToHalt();
    }
  }
};
```

Simple example: find the maximum value

Semi-pseudocode:

```
class MaxValueVertex
    : public Vertex<int, void, int> {
  void Compute(MessageIterator *msgs) {
    int maxv = GetValue();
    for (; !msgs->Done(); msgs->Next())
        maxv = max(msgs.Value(), maxv);

    if (maxv > GetValue()) || (step == 0)) {
        *MutableValue() = maxv;
        OutEdgeIterator out = GetOutEdgeIterator();
        for (; !out.Done(); out.Next())
            sendMessageTo(out.Target(), maxv)
    } else
        VoteToHalt();
    }
  }
};
```

find maximum value

Simple example: find the maximum value

Semi-pseudocode:

```
class MaxValueVertex
    : public Vertex<int, void, int> {
  void Compute(MessageIterator *msgs) {
    int maxv = GetValue();
    for (; !msgs->Done(); msgs->Next())
        maxv = max(msgs.Value(), maxv);

    if (maxv > GetValue()) || (step == 0)) {
        *MutableValue() = maxv;
        OutEdgeIterator out = GetOutEdgeIterator();
        for (; !out.Done(); out.Next())
            sendMessageTo(out.Target(), maxv)
    } else
        VoteToHalt();
    }
  }
};
```

send maximum value to all edges

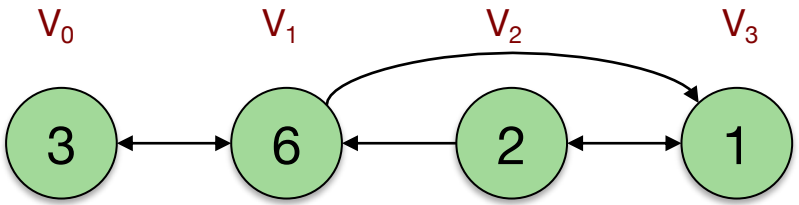
Simple example: find the maximum value

Semi-pseudocode:

```
class MaxValueVertex
    : public Vertex<int, void, int> {
  void Compute(MessageIterator *msgs) {
    int maxv = GetValue();
    for (; !msgs->Done(); msgs->Next())
        maxv = max(msgs.Value(), maxv);

    if (maxv > GetValue()) || (step == 0)) {
        *MutableValue() = maxv;
        OutEdgeIterator out = GetOutEdgeIterator();
        for (; !out.Done(); out.Next())
            sendMessageTo(out.Target(), maxv)
    } else
        VoteToHalt();
    }
  }
};
```

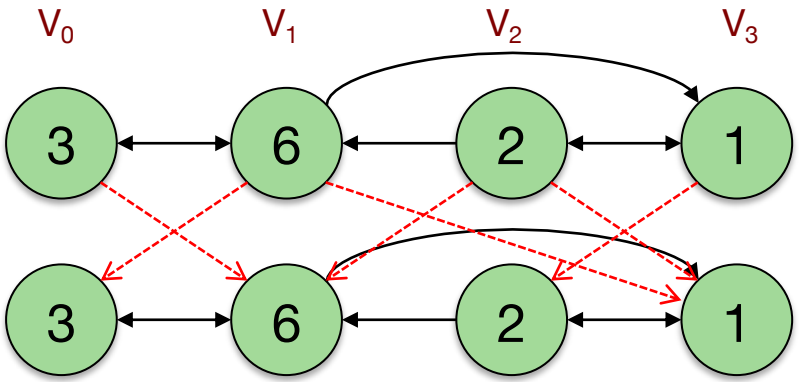
Simple example: find the maximum value



Superstep 0

Active vertex Inactive vertex

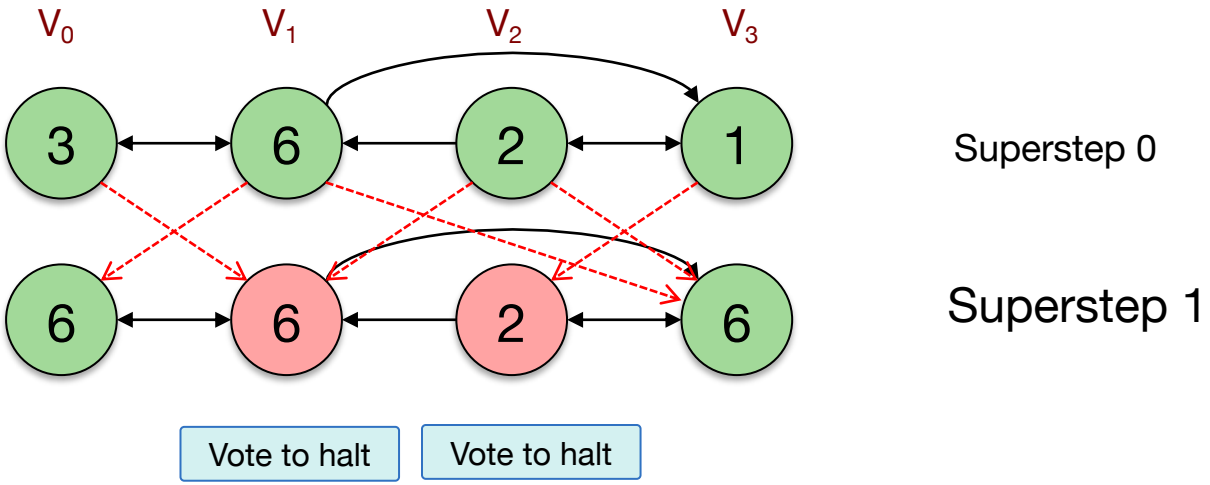
Simple example: find the maximum value



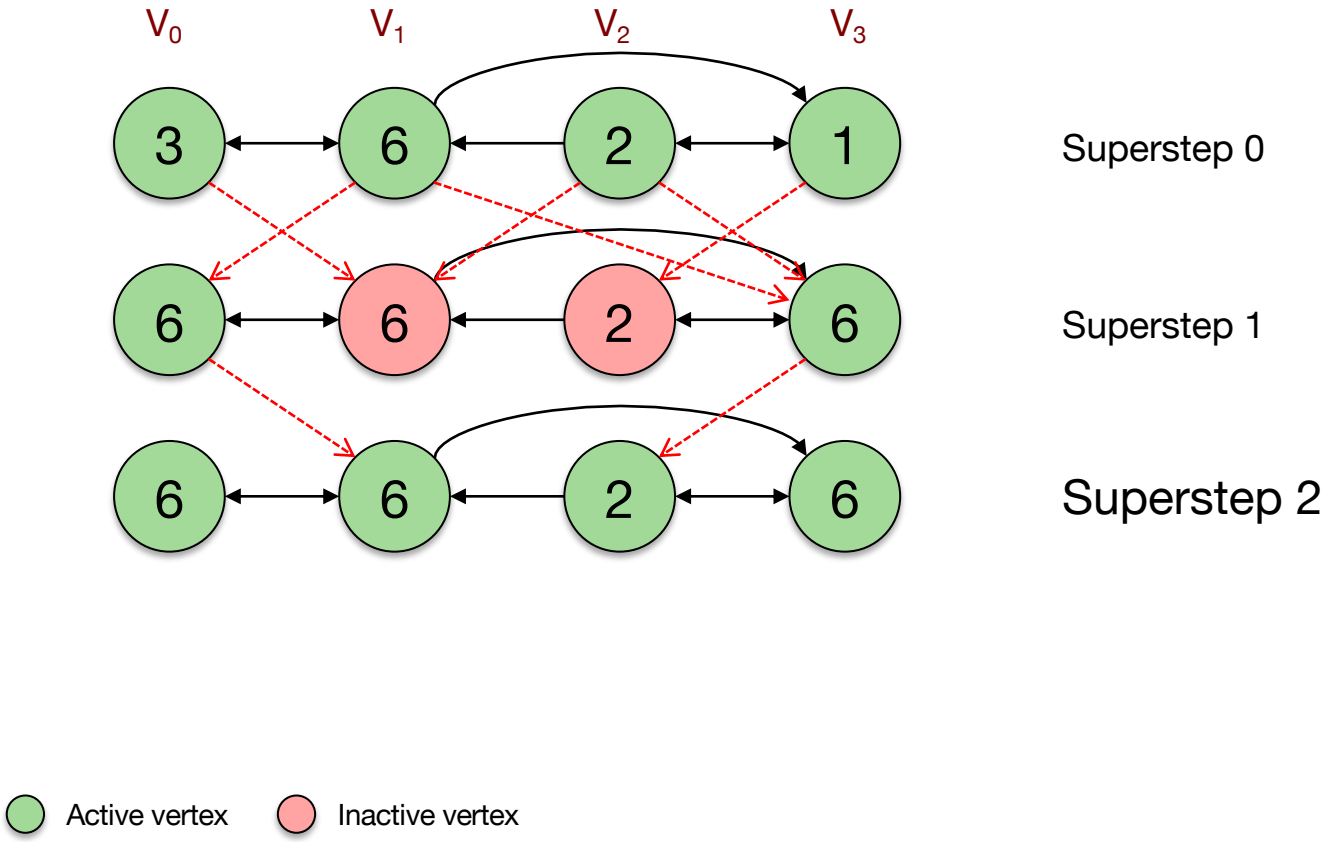
Superstep 0

Active vertex Inactive vertex

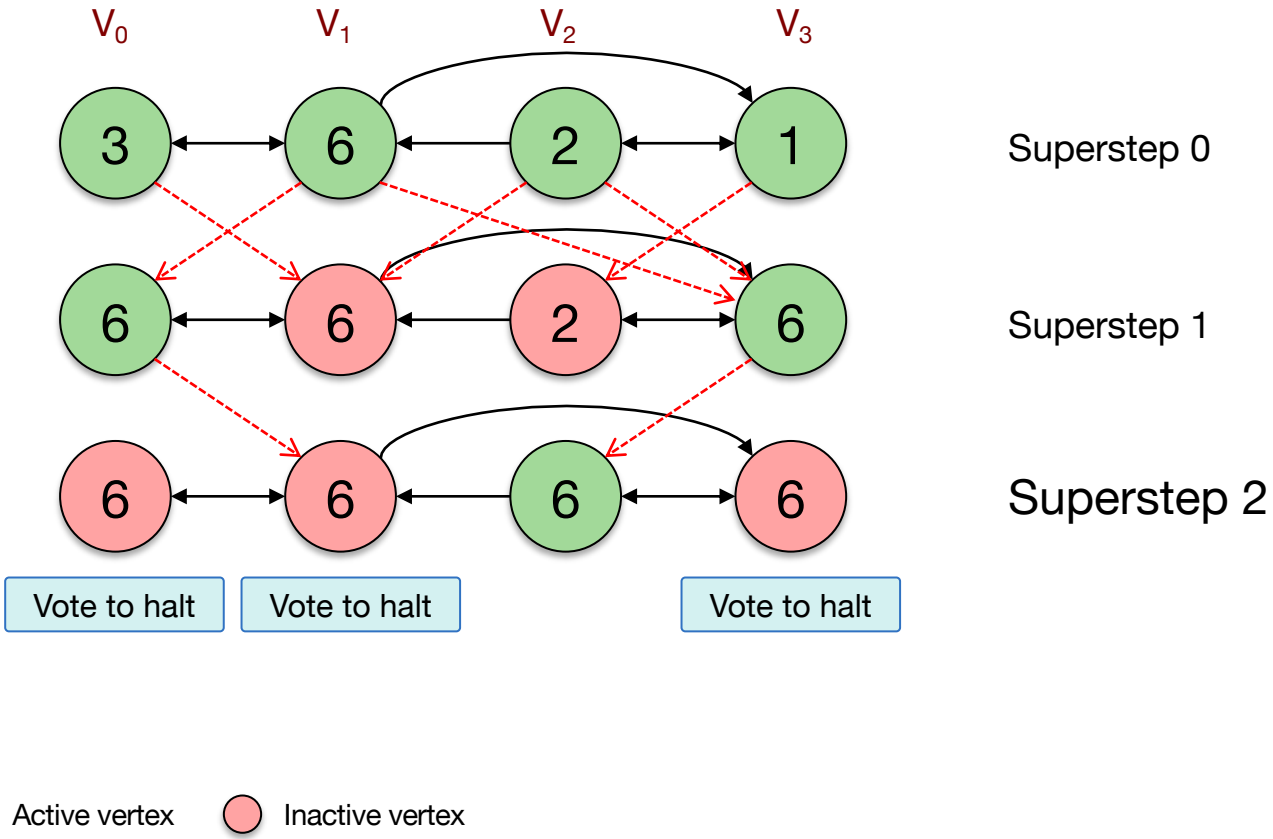
Simple example: find the maximum value



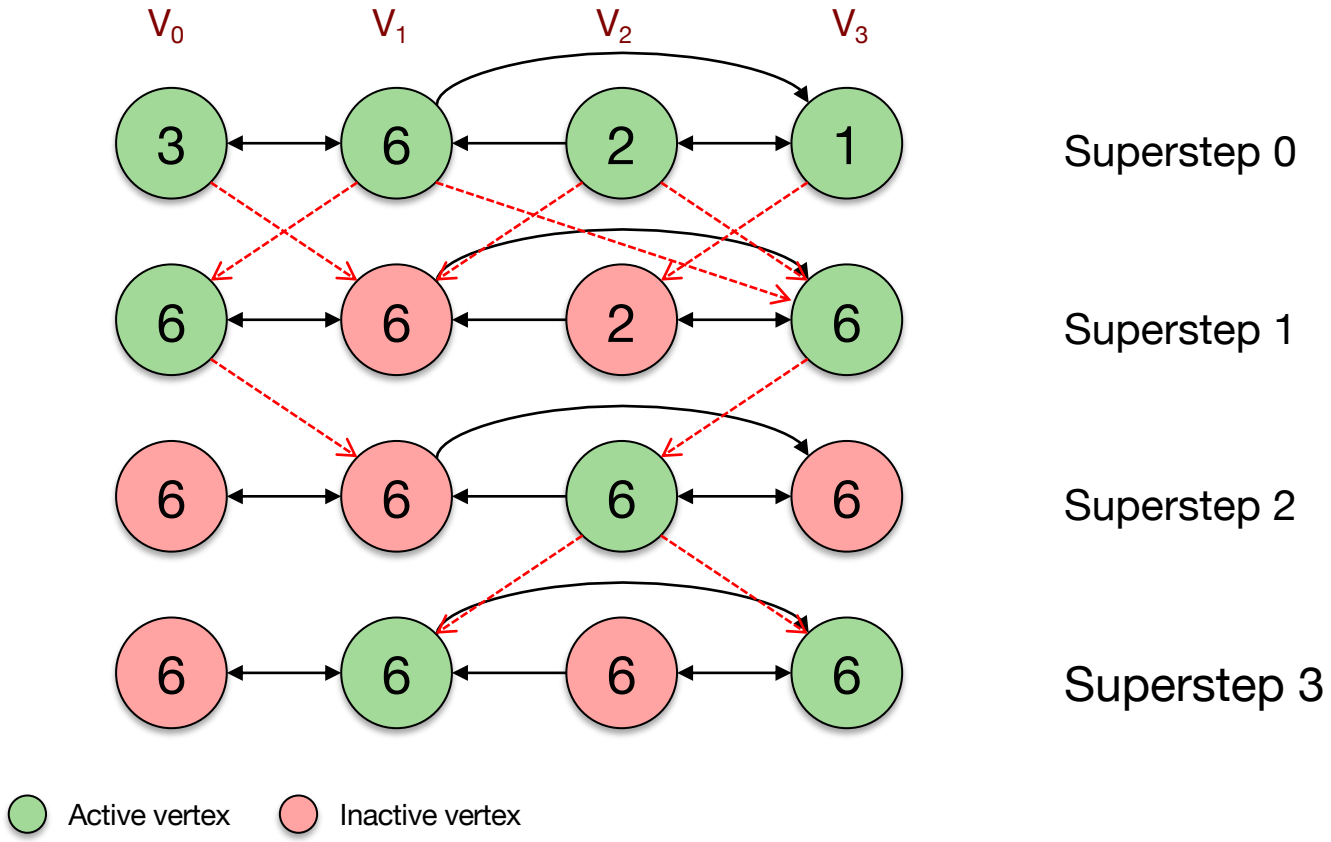
Simple example: find the maximum value



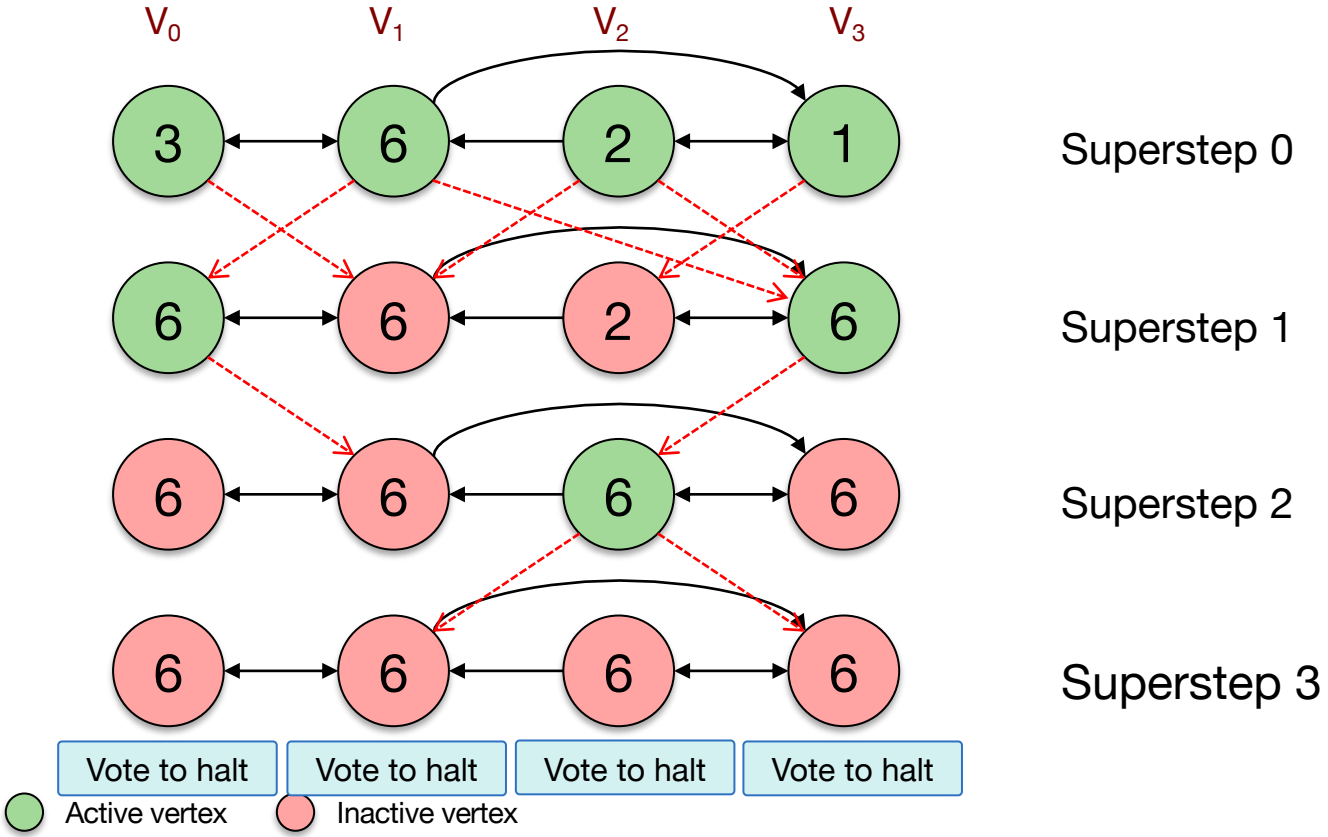
Simple example: find the maximum value



Summary: find the maximum value



Summary: find the maximum value



Pregel vs. MapReduce for Graph Algorithms

MapReduce

- Read full graph from HDFS each round
- Full shuffle to regroup by vertex
- Write updated values to HDFS each round
- Graph structure rebuilt every iteration
- 50 iterations = 50 disk reads + 50 writes

Pregel

- Load graph into memory once
- Messages pass along edges in memory
- Only checkpoint to disk periodically
- Graph structure stays alive across rounts
- 50 iterations = 1 disk read + checkpoints

For iterative graph algorithms: Pregel can be an order of magnitude faster.

Apache Spark

Lazy dataflow over resilient distributed datasets.
In-memory caching.
Lineage-based fault recovery.

Why Spark Was Needed

MapReduce (multi-stage)	Spark Equivalent
Read from HDFS	Read from HDFS
Stage 1 (map, reduce)	Transform (in memory)
Write to HDFS	Transform (in memory)
Read from HDFS	Transform (in memory)
Stage 2 (map, reduce)	Action → write output
Write to HDFS	

Spark: intermediate data stays in memory between stages.
No disk round-trip between stages.
Transformations pipeline together. Data reused across actions.

Spark Architecture

Driver Program

- Application logic
- Builds execution DAG
- Coordinates the job

Cluster Manager Program

- Allocates resources
- Launches executors
- (YARN / Kubernetes / Standalone)

Workers

- Machines in the cluster that execute tasks assigned by the driver.

Workers

- Processes on workers
- Run tasks in threads
- Cache partitions in memory

Driver → Executors: *orchestration vs. execution*

The driver is a single point of coordination (= single point of failure)

Resilient Distributed Datasets (RDD)

R Resilient

- Lost partitions are recomputed from lineage.
- No explicit replication of intermediate data.

D Distributed

- Data is split across many machines.
- Each partition is a scheduling unit assigned to an executor.

D Dataset

Broadly interpreted:

- text lines, tuples, key-value pairs, parsed objects.
- Elements of any type.

- **Partitioned** across executors — partition is the scheduling unit
- **Immutable** — transformations produce new RDDs, never modify in place
- **Fault-tolerant** through lineage — lost partitions *recomputed*, not replicated
- Optionally **cached** in memory or on disk for reuse across actions

How Spark Partitions Input

File or Directory

- Text files, CSV, JSON, binary formats
- Split into block-aligned partitions

HDFS or S3

- Distributed file system
- Partitions aligned to block boundaries

Database or Store

- Base, Cassandra, JDBC sources
- Partitioned by key range or shard

Another RDD

- Derived from a prior transformation
- Partition count may change on shuffle.

The partition is the scheduling unit.
Logical records inside a partition depend on the input format.

Transformations & Actions

Transformations

- Produce a new RDD from an existing one
- Execute lazily — no computation yet
- Build the lineage graph
- Examples: map, filter, flatMap,
- *groupByKey*, *reduceByKey*, *join*

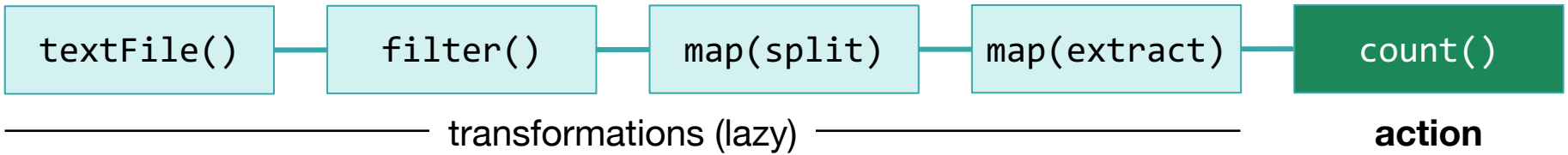
Actions

- Trigger execution of the lineage graph
- Return a result to the driver or write output
- Nothing runs until an action is called
- Examples: count, collect, take(n),
- *reduce*, *saveAsTextFile*, *foreach*

Lazy Evaluation& Lineage Graphs

```
lines    = textFile("logs.txt")           # no execution yet
errors   = lines.filter(starts("ERROR")) # Spark records plan
fields   = errors.map(split_on_tabs)     # Spark records plan
messages = fields.map(extract_message)   # Spark records plan
count    = messages.count()              # ACTION: triggers execution
```

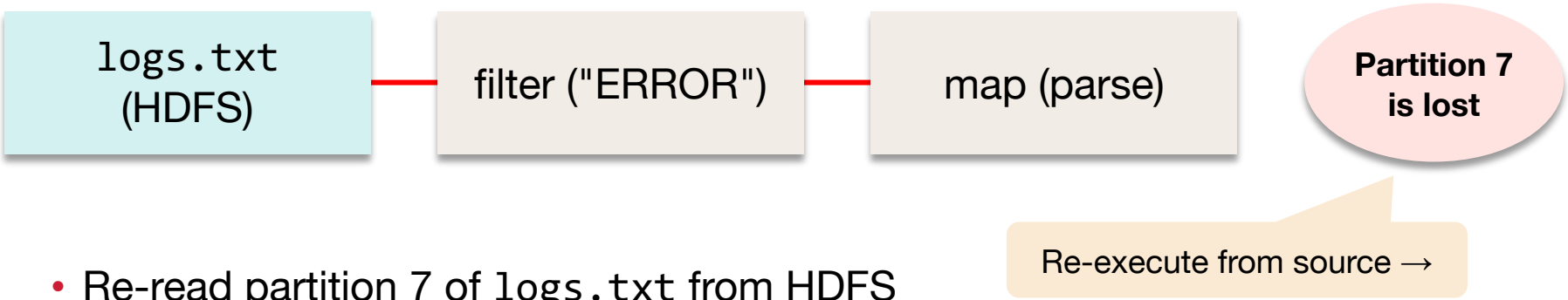
Lineage graph (built during transformations, executed at action):



Transformations build the recipe. An **action** executes it.
This deferred execution enables optimization, pipelining, and lineage-based fault recovery.

Fault Tolerance Through Lineage

```
messages = textFile("logs.txt").filter().map()
```



- Re-read partition 7 of `logs.txt` from HDFS
- Re-apply filter → re-apply map → recover partition 7
- Only the missing partition is recomputed, not the full RDD
- Works because transformations are deterministic
- **For long lineages:** use `checkpoint()` to write RDD to disk, shortening recovery

Lazy Evaluation& Lineage Graphs

```
lines    = textFile("logs.txt")
errors   = lines.filter(starts("ERROR"))
messages = errors.map(parse_message)
messages.cache()            # keep in memory after first computation

mysqlCount = messages.filter(contains("mysql")).count()
phpCount   = messages.filter(contains("php")).count()
# Second and third actions reuse cached messages — no re-read/re-parse
```

Cache when:

- RDD reused in 2+ actions
- Iterative algorithms (ML, graph)
- Interactive/exploratory analysis
- Shared intermediate datasets

Watch out for:

- Memory pressure can evict partitions
- Evicted partitions are recomputed
- Only cache if reuse > recomputation cost

Examples: RDD Transformations

Transformation	Description
map(f)	Pass each element through a function func
filter(func)	Select elements of the source on which func returns true
flatmap(func)	Each input item can be mapped to 0 or more output items
groupByKey([numtasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, seq[V]) pairs
reduceByKey(func, [numtasks])	Aggregate the values for each key using the given *reduce* function
sortByKey([ascending], [numtasks])	Sort keys in ascending or descending order
join(otherDataset, [numtasks])	Combines two datasets, (K, V) and (K, W) into (K, (V, W))

Examples: RDD Actions

Action	Description
count()	Returns the number of elements. Forces evaluation.
collect()	Returns all elements to the driver. Dangerous for large datasets.
first()	Returns the first element. Useful for quick inspection.
take()	Returns the first n elements. Safer than collect() for inspection.
reduce(f)	Aggregates all elements with f. Must be commutative + associative.
saveAsTextFile(f)	Writes output to storage. One file per partition.
foreach(f)	Applies f for side effects on each element. Runs on executors.

Example: Error Log Analysis

```
lines    = textFile("app.log")
errors   = lines.filter(starts("ERROR"))
fields   = errors.map(split_on_tab)
messages = fields.map(extract_message)
messages.cache()

mysqlCount = messages.filter(contains("mysql")).count()
phpCount   = messages.filter(contains("php")).count()
topTerms   = messages.flatMap(tokenize)
                     .map(term → (term, 1))
                     .reduceByKey(add)       # shuffle
                     .sortBy(count desc)     # shuffle
                     .take(20)
```

Build lineage (no execution)

Register caching intent

Action #1: execute + cache messages

Action #2: reuses cached messages

Narrow: flatMap + map

Wide: shuffle (stage boundary)

Wide: global sort (stage boundary)

The framework handles scheduling, distribution, and fault recovery invisibly.

Spark Beyond Core RDDs

Spark SQL

- DataFrames with named, typed columns
- SQL queries
- Catalyst optimizer for query planning

Spark MLlib

- Machine learning on distributed data
- Classification, regression, clustering, CF, pipelines

Spark GraphX

- Directed graph model with properties
- Graph algorithms running on the Spark engine.

Spark Streaming

- Derived from a prior transformation
- Partition count may change on shuffle.

Distributed Machine Learning

What Machine Learning Changes

	Data Processing	ML Training
Passes over data	One or a few	Many (one per epoch)
State across rounds	None	Model parameters updated each round
Partitioned object	Input dataset	Data (and/or model)
Output	Transformed dataset	Trained model
MapReduce fit	Good for batch	Poor — iterative, stateful
Spark fit	Good to excellent	Good for data prep; limited for training

Data Parallelism

Parameter Server

- A central server holds model parameters.
- Workers push gradients to the server and pull updated parameters.
- Server becomes a bottleneck as the number of workers grows.

All Reduce

- All workers participate in computing the aggregate with no central coordinator
- All-Reduce is the dominant approach in modern deep learning frameworks.
- Scales better than parameter server.

Model Parallelism

- Data parallelism requires the full model to fit on one device
- Large language models may have hundreds of billions of parameters
  - *Too large for a single GPU*
- **Model parallelism**: split the model itself across devices

Pipeline Parallelism

- Model split into layer groups
- Each device holds consecutive layers
- Micro-batching keeps all devices busy.
- Lower communication overhead

Tensor Parallelism

- Individual weight matrices split across devices
- Distributed matrix multiply.
- Frequent activation exchanges required
- Higher communication cost

Large-scale training uses all three simultaneously:
data parallelism across groups, **pipeline** across layers, **tensor parallelism** within layers

Where Frameworks Fit

- **MapReduce**: restrict the structure
  - Rigid two-phase model makes the runtime responsible for the cluster
- **BSP / Pregel**: make rounds explicit
  - Supersteps with barriers organize iterative state and message passing
- **Spark**: defer execution
  - Lazy dataflow, in-memory caching, and lineage eliminate the need to write intermediate results to disk
- **Distributed ML**: match the framework to the workload
  - Data prep in Spark; training in purpose-built runtimes

The End