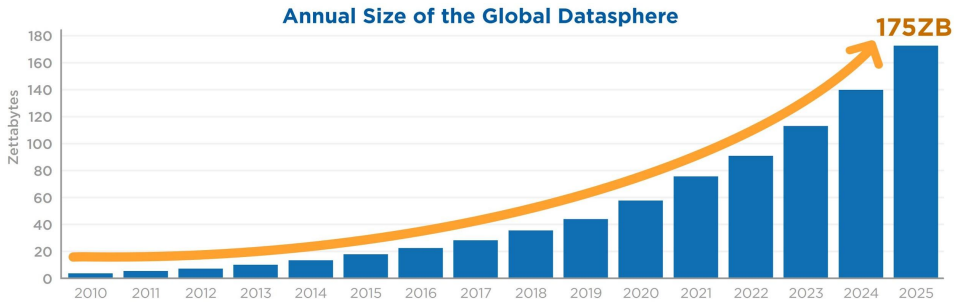
Distributed Technology for Big Data

ISTA 322 - Data Engineering

Parallelism for Big Data Processing

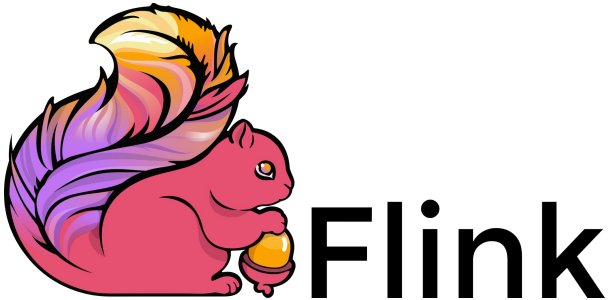
- Why parallelism?
- Scan 100 TB
  - 5 billion webpages
  - Average size of webpage 20k
  - 5 billion * 20k = 100TB
  - At 0.5 GB/sec:
    - ~200,000 sec = ~2.31 days
- Run it 100-way parallel:
  - 2,000 sec = 33 minutes
- 1 big problem = many small problems
  - Trick: make them independent



Data-Parallel Systems

- Data Distribution:
  - The data is split into smaller, manageable pieces that are distributed across multiple processing units for concurrent execution.
- Parallel Execution
  - Different processors work on separate chunks of data simultaneously, accelerating the overall computation process.
- Task Uniformity
  - Each processing unit performs the same operation, but on its own segment of the distributed data.
- Scalability
  - The system's capacity to handle processing can grow by adding more processing units

Data-Parallel Frameworks & Technologies



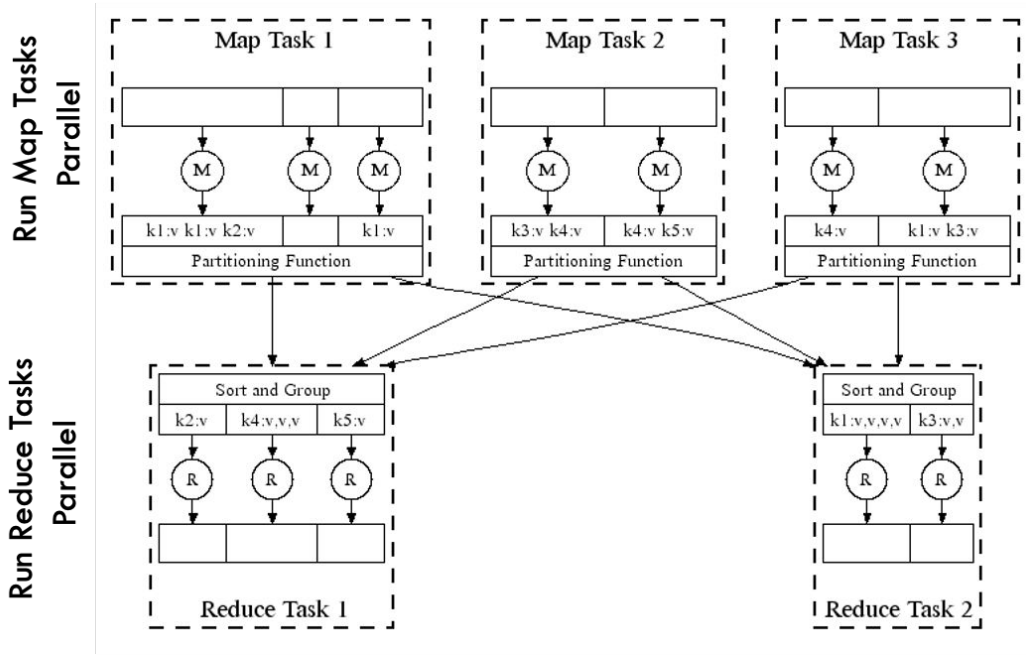
Map-Reduce Programming Model

Map Stage: Data Partitioning & Mapping

- Split data into partitions for distribution across processing units.
- Each unit processes its partition, mapping data to key-value pairs.

Shuffle Stage: Data Communication & Key Grouping

- Shuffle phase ensures all key-value pairs with the same key are sent to the same processing unit.

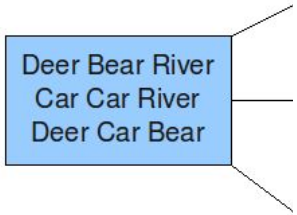


Reduce Stage: Aggregation & Summary

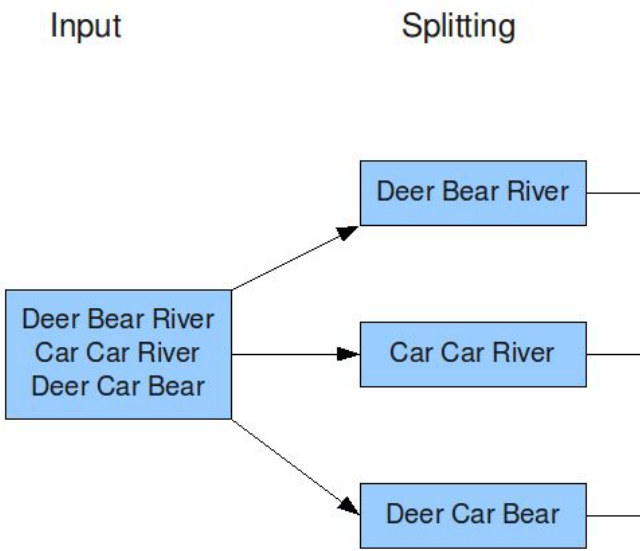
- The reduce function is applied to each group of values, aggregating them into a final result.

MapReduce - Get word count

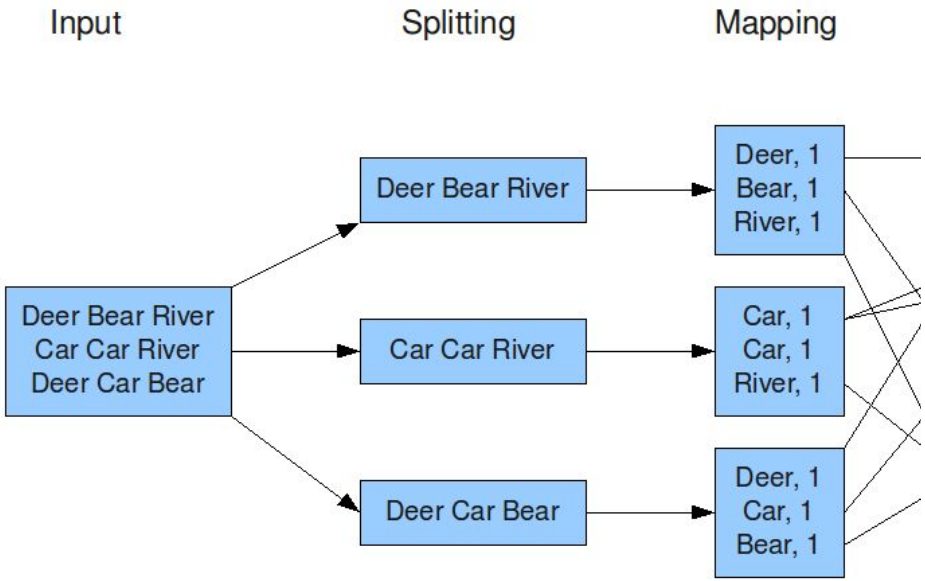
Input



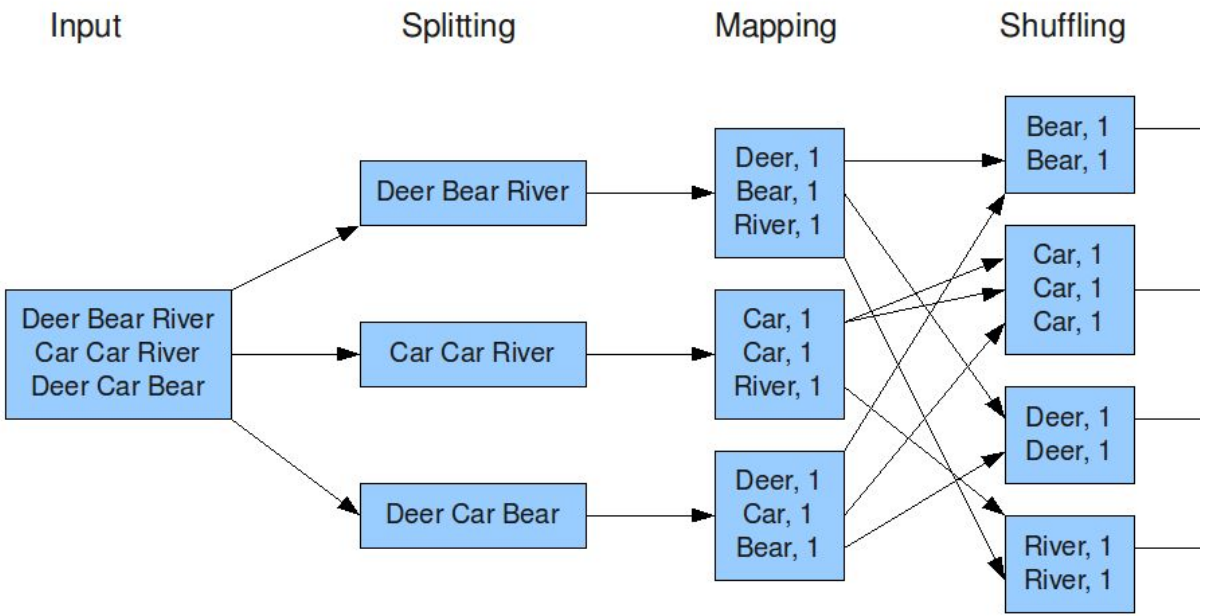
MapReduce - Splitting



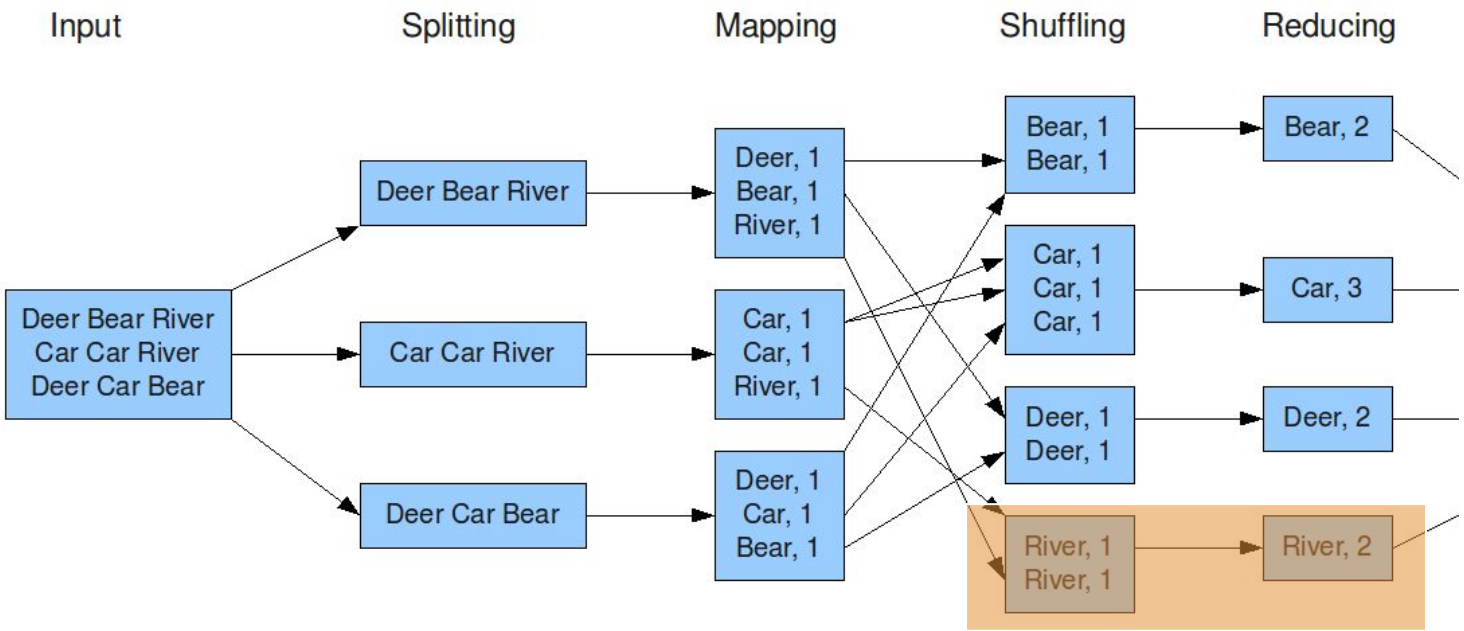
MapReduce - Mapping



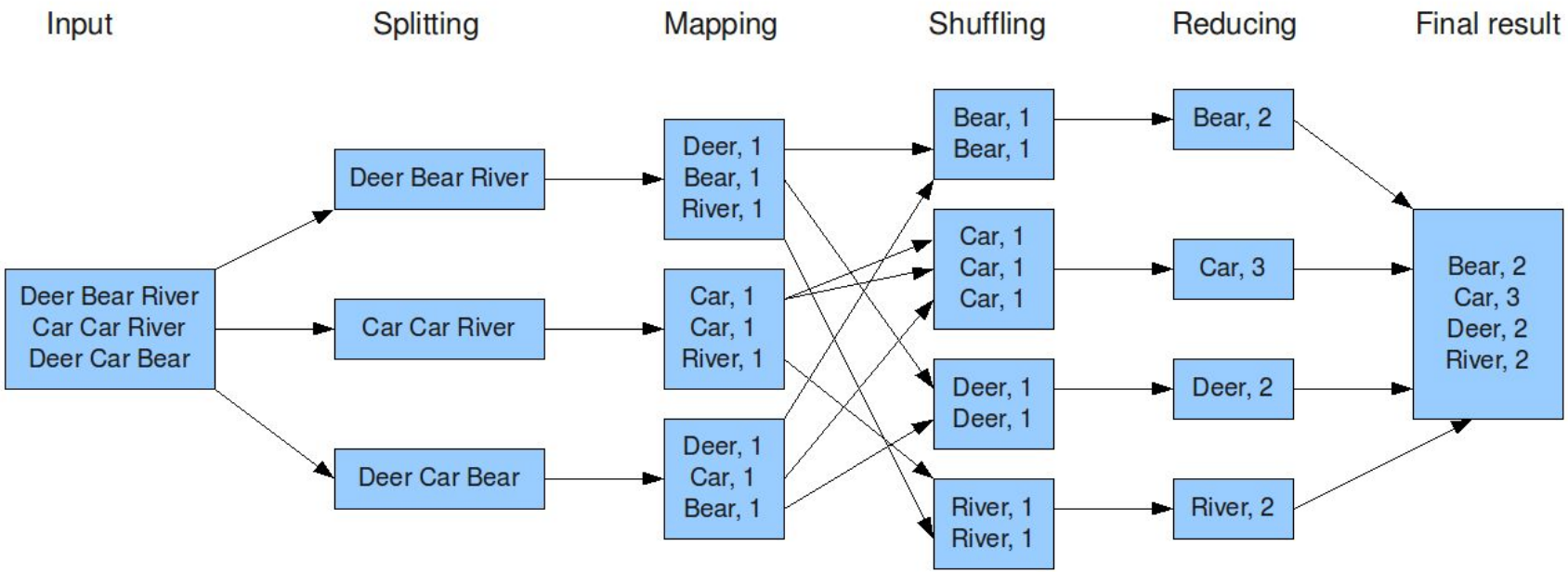
MapReduce - Shuffling



MapReduce - Reducing



MapReduce


Input
Splitting
Mapping
Shuffling
Reducing
Final result
Deer Bear River
Car Car River
Deer Car Bear
Deer Bear River
Car Car River
Deer Car Bear
Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1
Bear, 1
Bear, 1
Car, 1
Car, 1
Car, 1
Deer, 1
Deer, 1
River, 1
River, 1
Bear, 2
Car, 3
Deer, 2
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2

Large Tabular Data: ICU Service

	ROW_ID	SUBJECT_ID	HADM_ID	ICUSTAY_ID	DBSOURCE	FIRST_CAREUNIT	LAST_CAREUNIT	FIRST_WARDID	LAST_WARDID	INTIME	OUTTIME	LOS
1	366	269	106296	206613	carevue	MICU	MICU	52	52	2170-11-05 11:05:29	2170-11-08 17:46:57	3.2788
2	367	270	188028	220345	carevue	CCU	CCU	57	57	2128-06-24 15:05:20	2128-06-27 12:32:29	2.8939
3	368	271	173727	249196	carevue	MICU	SICU	52	23	2120-08-07 23:12:42	2120-08-10 00:39:04	2.0600
4	369	272	164716	210407	carevue	CCU	CCU	57	57	2186-12-25 21:08:04	2186-12-27 12:01:13	1.6202
5	370	273	158689	241507	carevue	MICU	MICU	52	52	2141-04-19 06:12:05	2141-04-20 17:52:11	1.4862

```
map(("ROW_ID","SUBJECT_ID","HADM_ID","ICUSTAY_ID","DBS
OURCE","CAREUNIT","FIRST_WARDID","LAST_WARDID","INTIME
","OUTTIME","LOS"))->[(CAREUNIT, LOS)]

Reduce((CAREUNIT, LOS1), (CAREUNIT,LOS2))-> (CAREUNIT,
LOS1+LOS2)
```

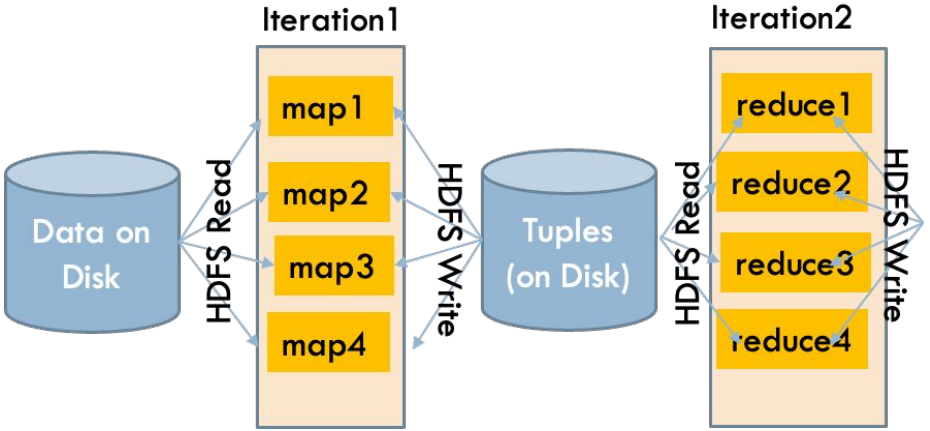
Hadoop MapReduce & Hadoop Distributed File Systems

Hadoop MapReduce

- Open source implementation of the Google MapReduce programming model

Hadoop Distributed File System

- Distributed file system design
- Splits files into blocks distributed across nodes
- Facilitates parallel processing by storing data close to where it is processed



Integration for Efficiency

- HDFS stores data across clusters for redundancy and locality
- MapReduce processes data on the Hadoop cluster
- Intermediate results are written back to HDFS between stages

Hadoop MapReduce Pros & Cons

Advantages of Hadoop MapReduce

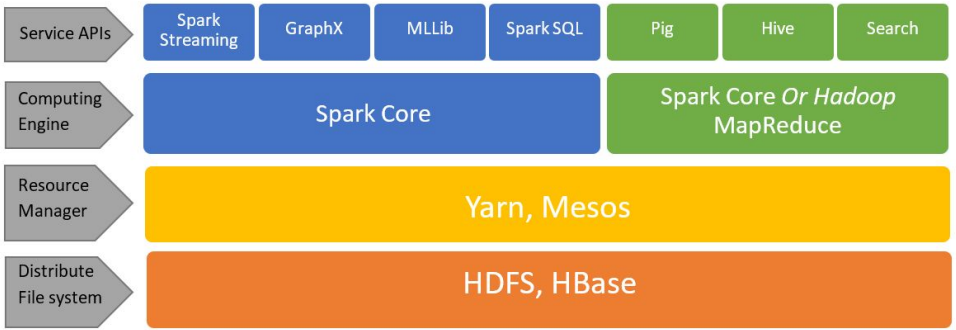
- Scalability: Easily processes petabytes of data across hundreds of nodes
- Flexibility: Handles various types of data, structured or unstructured
- Cost-Effective: Utilizes commodity hardware, reducing infrastructure costs
- Fault Tolerance: Automatically handles node failures with data replication
- High Throughput: Efficiently manages large volumes of data with parallel processing
- Community Support: Benefits from a robust and active open-source community

Two major limitations of Hadoop MapReduce:

- Difficulty of programming directly in MapReduce
  - Many problems aren’t easily described as map-reduce
- Performance bottlenecks due to disk I/O, and serialization
  - Persistence to disk typically slower than in-memory work

Apache Spark - Revolutionizing Data Processing

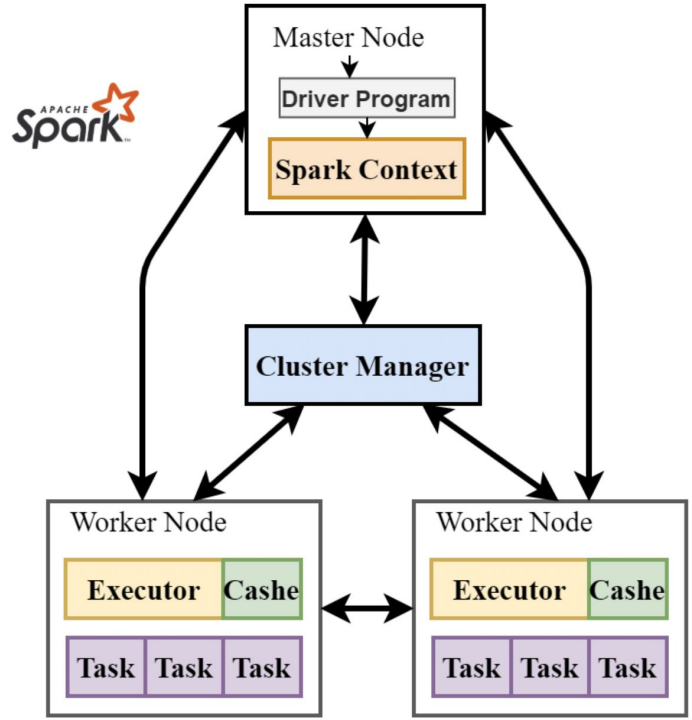
- Speed: Executes tasks up to 100x faster than MapReduce with in-memory processing.
- Ease of Use: Offers high-level APIs in Java, Scala, Python, and R.
- Advanced Analytics: Supports SQL queries, streaming data, machine learning, and graph processing.
- General Purpose: A comprehensive engine for a wide range of tasks.
- Resource Management: Compatible with Hadoop YARN, Apache Mesos, or can run standalone.
- Active Community: One of the largest open-source communities in big data.



Spark Architecture

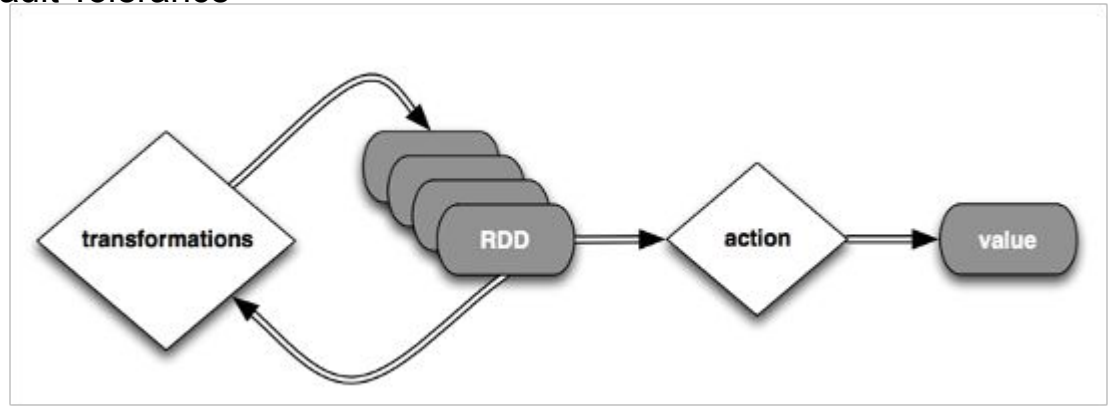
Master Node
- Runs the driver program
- Coordinates Jobs & Distributes Tasks
- Establishes Communication with Cluster
- Manages Resource Allocation

Worker Nodes:
- The Powerhouses of Processing
- Execute Tasks Assigned by Master
- Maintain Data in Memory or Disk



Resilient Distributed Datasets (RDDs)

- A fault-tolerant collection of elements
- RDDs are immutable
- Transformations maps an RDD to another RDD (Map)
  - Transformations are lazy
- Action operation reduces RDD to a value (Reduce)
- Lineage Tracking for Fault Tolerance



```python
from pyspark import SparkConf, SparkContext

# Create a Spark configuration object with settings
conf = SparkConf().setMaster("local").setAppName("My Spark Hello World")

# Initialize a SparkContext with the configuration
sc = SparkContext(conf=conf)

# Create an RDD by parallelizing a small Python list of numbers
numbers = [1, 2, 3, 4, 5]
numbers_rdd = sc.parallelize(numbers)

# Perform a map operation to square the numbers
squared_numbers = numbers_rdd.map(lambda x: x**2)

# Perform a reduce operation to add all numbers together
sum_of_squared = squared_numbers.reduce(lambda n1, n2: n1+n2)

# Print the result
print(sum_of_squared)

# Stop the SparkContext at the end of the application
sc.stop()
```

Spark DataFrame

- Immutable, distributed collection of data
- Organized into named columns, similar to tables in a relational database
- Extends the functionality of RDDs by providing a schema to view and organize data
- Imposes a structure onto distributed data for higher-level abstraction
- Allows use of column names for data operations, enhancing readability and maintainability

```python
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, IntegerType, StringType

# Initialize a SparkSession
spark = SparkSession.builder \
    .appName("Example DataFrame") \
    .getOrCreate()

# Sample data
data = [("James", 34), ("Anna", 20), ("Robert", 50)]

# Define schema of the DataFrame
schema = StructType([
    StructField("Name", StringType(), True),
    StructField("Age", IntegerType(), True)
])

# Create DataFrame
df = spark.createDataFrame(data, schema)

# Show the DataFrame
df.show()

# Stop the Spark session
spark.stop()
```

Spark SQL

- Provides SQL like interface
- Provides a common means to access a variety of data sources, from traditional databases to distributed datasets.
- Employs the same RDDs technology that powers Spark's big data.
- Allows for executing SQL queries across massive datasets in a distributed fashion.

```
from pyspark.sql import SparkSession

# Start a Spark session
spark = SparkSession.builder.appName("Movies in Spark SQL").getOrCreate()

# Load a DataFrame
df = spark.read.option('header', True).csv("movie.csv")

# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("movie")

# Execute a SQL query
sqlDF = spark.sql("SELECT count(*) FROM movie WHERE genres LIKE '%Comedy%'")

# Show the results
sqlDF.show()
```