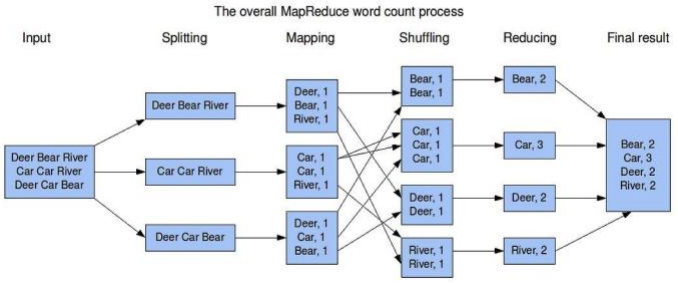
Volume, Velocity, Variety

MapReduce



The Combiner performs a partial aggregation of the same keys at the local Mapper node, reducing the intermediate results transmitted over the network.

1. The mapper read the data from HDFS. A mapper task is started for each HDFS data block. Each record is processed by one map function. The mapper output is sorted and stored on local disks.
2. In order to reduce the mapper output size and save the network I/O cost, a combiner can be designed to perform local aggregation.
3. According to the number of reducers specified, the mapper output is partitioned to several groups, each corresponding to one reducer.
4. The reducer fetches its partition from each mapper and merges the locally sorted data to obtain a global order. Next, each key and its associated values are processed by one reducer function. The output is stored back to HDFS

'mapreduce.map.output.key.field.separator':',',
'mapreduce.partition.keypartitioner.options':'-k1,1',
'mapreduce.partition.keycomparator.options':'-k1,1 -k2,2

Combiner vs. Reducer

1. Combiner is optional, but Reducer is mandatory.
2. Combiner operates on local data.
3. Combiner can only be applied to specific operations like summing and counting.

Combiner ⟺ Reducer

Aggregation operations are idempotent, meaning that multiple aggregations on the same data will not affect the final result (e.g., summation, counting).

Combiner (Primarily uses disk I/O)

Advantages:

- Easy to implement and highly generalizable.
- Effectively utilizes memory resources without requiring significant customization.

Disadvantages:

- Frequent I/O operations result in slower overall performance.
- Only performs local aggregation on the same Mapper node, limiting its efficiency.
- Unsuitable for tasks involving non-idempotent functions

In-mapper Combining (Primarily uses in-memory operations)

Advantages:

- Significantly reduces I/O operations, resulting in better performance.
- Decreases the volume of intermediate data, reducing the load on the network and Reducer.

Disadvantages:

- High memory usage, which can lead to memory overflow in cases with large datasets.
- More complex to implement as it requires careful memory management and custom logic.

Comparison Between MapReduce (MR) and Spark

1. **Data Processing Approach**:
   MR: Uses a disk-based batch processing model, leading to a significant amount of I/O operations.
   Spark: Employs a memory-based processing model where most data is handled in memory, significantly reducing I/O overhead.
2. **Fault Tolerance Mechanism**:
   MR: Implements fault tolerance by restarting tasks from scratch. If a task fails, the system reassigns it to another node for execution.
   Spark: Uses RDD Lineage to achieve efficient fault tolerance, akin to "save points" in games, allowing only the failed part of a task to be recomputed.
3. **Ease of Programming**:
   MR: Operates at a lower level, requiring developers to manually write Mapper and Reducer functions, which can be complex.
   Spark: Offers higher-level APIs like DataFrames (DF), making it much easier and faster to develop.
4. **Additional Features**:
   MR: Limited to basic distributed computing tasks (Map -> Reduce).
   Spark: Supports a wide range of functionalities, including machine learning (MLlib), SQL processing (SparkSQL), graph computation (GraphX), and stream processing (Spark Streaming).
5. **Performance**:
   MR: Disk-based, resulting in slower processing speeds.
   Spark: Memory-based, offering significantly faster performance.

Spark

1. RDD/DataFrame operations are categorized into transformation and action, and the transformation operations are lazily evaluated.
2. When reaching an action, a job will be created, and the real execution begins at this point.
3. Each transformation operation that requires data shuffling becomes the boundary of stages. Spark create a DAG based on the stages.
4. Within each stage, a task will be run for each partition on an executor, and the tasks run in parallel

RDD（Resilient Distributed Datasets）

Immutable, Lazy Evaluation, Cacheable, Parallel Processing, Typed

```
fileRDD = sc.textFile(inputFile)
courseRDD = fileRDD.map(lambda line: line.split(':')[1]).flatMap(lambda x: x.split(';')).map(lambda x: (x.split(',')[0], x.split(',')[1]))
resRDD=courseRDD.mapValues(lambda x:(int(x),1)).reduceByKey(lambda a, b:(a[0]+b[0], a[1]+b[1])).mapValues(lambda x:x[0]/x[1]).sortByKey()
```

Transformation

Transformations create a new RDD from an existing one without immediately executing the operation. They are **lazy**, meaning they only build a logical execution plan and are executed when an **Action** is triggered. (map, filter, flatMap, groupBy, reduceByKey, sortByKey)

Action

Actions trigger the actual computation of RDD transformations and return the result to the Driver Program or save it to an external storage system. (collect, count, take, reduce, saveAsTextFile)

Types of RDD Dependencies

Narrow Dependency

Each partition of the child RDD depends on at most one partition of the parent RDD. No shuffle occurs; transformations are localized to the same node. (map, filter, flatMap)

Wide Dependency

Each partition of the child RDD depends on multiple partitions of the parent RDD.Data redistribution (shuffle) is required, which can significantly impact performance. (groupByKey, reduceByKey, sortByKey)

Lineage Graph

The Lineage Graph shows the dependencies between RDDs, illustrating how each RDD is derived from its parent RDD(s). It is used to recompute lost data in case of failure, ensuring fault tolerance.

Limitations of RDD

1. **Lack of Optimization**
   RDD computations are treated as a **black box**, meaning Spark cannot fully understand the logic inside user-defined functions (e.g., `map`, `filter`).
2. **Unknown Data Schema**
   RDDs do not have a predefined schema, so Spark lacks information about the structure of the data.
3. **Verbose and Complex Code**
   RDD-based code can be verbose and less intuitive, especially for complex operations like joins or aggregations.
4. **Performance Issues**
   Without schema information and optimizations, RDDs often involve more computation and memory overhead, leading to slower performance compared to structured APIs.

Advantages of DF API

1. **Intuitive and Declarative**
   DF provide a high-level abstraction similar to tables in a relational database.
2. **Optimized Execution**
   DF operations are optimized through Spark's **Catalyst Optimizer**, which generates efficient execution plans.
3. **Known Schema**
   DF have a defined schema (column names and types), enabling Spark to enforce data correctness and improve query performance.
4. **Improved Performance**

DF leverage Spark's **Tungsten Execution Engine**, which optimizes memory and CPU usage, leading to faster execution.

DF

```
fileDF = spark.read.text(inputFile)
student=fileDF.select(split(fileDF['value'],':').getItem(0).alias('sid'),
split(fileDF['value'], ':').getItem(1).alias('courses'))
scDF = student.withColumn('course', explode(split('courses', ';')))
scDF2=scDF.select(split(scDF['course'],',').getItem(0).alias('cname'),
split(scDF['course'], ',').getItem(1).alias('mark'))
avgDF = scDF2.groupBy('cname').agg(avg('mark')).orderBy('cname')
```

DSMS

real-time, continuous, infinite

Sampling from a Data Stream

1. Fixed proportion sampling (sample)
2. Random sample of fixed size (Reservoir Sampling) (takeSample)
3. Queries over sliding windows

Datar-Gionis-Indyk-Motwani Algorithm (DGIM)

The DGIM algorithm provides an **approximate count** of 1s in the sliding window, and the accuracy is inherently limited by the algorithm's use of buckets. Each bucket merge introduces a potential loss of granularity, leading to an error margin of up to **50% of the smallest bucket's size**.

Increasing accuracy requires more buckets, which increases **memory usage** and may diminish the algorithm's efficiency.

Filtering a data stream (Hashing)

First Cut

If B[h(x)] = 0, we can be sure that it does not belong to S

If B[h(x)] = 1, we can only say that it may belong to S (Hash Collision)

This may lead to false positives.

Throwing Darts

$(1 - e^{-\frac{mk}{n}})^k$

m : bit array size (mod x). k: Number of hash function n : Size of the input array.

Bloom Filter

Multiple Hash Functions and AND Condition

Finding frequent elements

```
maj_index = 0
count = 1
for i in range(len(A)):
    if A[maj_index] == A[i]:
        count += 1
    else:
        count -= 1
    if count == 0:
        maj_index = i
        count = 1
return A[maj_index]
```

```
count = 0
for i in range(len(A)):
    if A[i] == cand:
        count += 1
if count > len(A)/2:
    return True
else:
    return False
```

Boyer-Moore Voting Algorithm

Heavy Hitter Problem

The Heavy Hitter problem involves identifying elements in a data stream or dataset that appear more frequently than a specified fraction of the total data. These elements are referred to as heavy hitters.

Misra-Gries Algorithm

For each new element:

1. If it is already among the monitored k-1 elements, increment its count.
2. If the number of monitored elements is less than k-1, add the new element to the monitored set with an initial count of 1.
3. Otherwise, decrement the count of all monitored elements, and remove any element whose count becomes 0.

Spacing Save Algorithm

For each new element:

1. If it is already among the monitored k-1 elements, increment its count.
2. If the number of monitored elements is less than k-1, add the new element to the monitored set with an initial count of 1.
3. Otherwise, Replace the element with the smallest count in the table with a new element and set the count of the new element to the value of the original count plus 1 (record possible errors).

Lossy Count

1. Divide the Stream into Buckets (Partitions) (Each buckets have $1/\varepsilon$)
2. Count Frequencies in Each Partition
3. Eliminate Low-Frequency Elements (Decrement all counters by 1 and drop elements with counter 0)
4. Combine Counts Across Partitions

Jaccard Distance & Jaccard Similarity

$$sim(C_1, C_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|} \quad d(C_1, C_2) = 1 - sim(C_1, C_2)$$

Shingle

Split a long document into multiple small, consecutive segments (Shingles), with the length defined by the parameter k.

- If k is too small, it results in many meaningless similarities.
- If k is too large, the Shingle set becomes overly sparse, making it difficult to find similar parts.

Min Hashing

Compress the large Shingle set into a shorter signature vector while preserving the similarity between sets. This reduces the data size while ensuring that similarity information is retained.

Row Hashing

Use hash functions instead of permutations.

Locality-Sensitive Hashing

1. Place each column of the signature matrix into multiple buckets.
2. If the signatures of two documents are hashed into the same bucket, they become candidate pairs for further similarity comparison.

Increasing b (number of bands):

- Each band's r (rows per band) becomes smaller.
- This generates more candidate pairs, increasing the probability of finding similar documents.
- However, it also increases the chance of false positives (non-similar pairs mistakenly identified as similar).

Decreasing b /Increasing r :

- Fewer candidate pairs are generated, making the results more precise.
- However, this can lead to false negatives (missing actual similar pairs).

$$P = 1 - (1 - s^r)^b \qquad r * b = total\ line \quad s: sim$$

Distance Measure

$$\text{Euclidean}(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2} \quad \text{Manhattan}(p,q) = \sum_{i=1}^{n}|p_i - q_i| \quad \text{similarity}(p,q) = \frac{p \cdot q}{\|p\| \cdot \|q\|}$$

$$d[i][j] = \begin{cases} d[i-1][j-1] & \text{if } s1[i] = s2[j] \\ 1 + \min(d[i-1][j], d[i][j-1], d[i-1][j-1]) & \text{otherwise} \end{cases}$$

BFS

```
class Mapper
    method Map(nid n, node N)
    d ← N.Distance
    Emit(nid n,N.AdjacencyList)                    //Pass along graph structure
    for all nodeid m ∈ N.AdjacencyList do
        Emit(nid m, d+1)                           //Emit distances to reachable nodes
```

```
class Reducer
    method Reduce(nid m, [d1, d2, . . .])
    d_min ← ∞
    M ← ∅
    for all d ∈ counts [d1, d2, . . .] do
        if IsNode(d) then
            M.AdjacencyList ← d                    //Recover graph structure
        else if d < d_min then                     //Look for shorter distance
            d_min ← d
    M.Distance ← d_min                             //Update shortest distance
    Emit(nid m, node M)
```

Pregel

Pregel divides the computation process into multiple steps (Superstep = Iteration):

1. Message Passing (State Update)
2. Local Computation
3. Synchronization

```
Active node                    Neigh node
For all active node do         Distance update
    Send msg to all neigh      If distance changed:
Status <- active                   Status <- active
```

PageRank

$$R(v) = (1-\beta)\frac{1}{N} + \sum_{u \in InNeighbors(v)} \frac{R(u)}{OutDegree(u)}$$

$$r(1) = 0.8\left(\frac{1}{6}\cdot r(1) + \frac{1}{2}\cdot r(4) + r(6) + \frac{1}{5}\cdot r(2)\right) + \frac{0.2}{6}$$
$$r(2) = 0.8\left(\frac{1}{6}\cdot r(1) + \frac{1}{3}\cdot r(3) + \frac{1}{2}\cdot r(5)\right) + \frac{0.2}{6}$$
$$r(3) = 0.8\left(\frac{1}{6}\cdot r(1) + \frac{1}{5}\cdot r(2) + \frac{1}{2}\cdot r(5)\right) + \frac{0.2}{6}$$
$$r(4) = 0.8\left(\frac{1}{6}\cdot r(1) + \frac{1}{5}\cdot r(2)\right) + \frac{0.2}{6}$$
$$r(5) = 0.8\left(\frac{1}{6}\cdot r(1) + \frac{1}{5}\cdot r(2) + \frac{1}{3}\cdot r(3)\right) + \frac{0.2}{6}$$
$$r(6) = 0.8\left(\frac{1}{6}\cdot r(1) + \frac{1}{5}\cdot r(2) + \frac{1}{3}\cdot r(3) + \frac{1}{2}\cdot r(4)\right) + \frac{0.2}{6}$$