00.B Linguaxes e motores no procesamento distribuído

Este documento integra dúas perspectivas complementarias:

- **as linguaxes usadas historicamente e na actualidade** no procesamento distribuído
- **exemplos equivalentes de Word Count** en distintos motores e linguaxes

O obxectivo é entender que **o que é distribuído é o motor**, non a linguaxe en si.

---

1. Linguaxes usadas no procesamento distribuído

1.1 Java (a linguaxe fundacional)

Uso histórico

- Hadoop MapReduce
- HDFS
- YARN
- HBase
- Hive (UDFs)
- Spark (API nativa)

Por que se usou

- Portabilidade (JVM)
- Boa xestión de memoria
- Madurez en contornos empresariais
- Integración sinxela con sistemas distribuídos

Situación actual

- Segue sendo clave a nivel interno
- Cada vez menos usada directamente polo usuario final

👉 Linguaxe do sistema, non do analista

1.2 Scala (a linguaxe "natural" de Spark)

Uso

- Spark (linguaxe orixinal)
- Flink (API principal)
- Kafka Streams

Por que se usou

- Programación funcional + OO
- Integración directa coa JVM

- Ideal para traballar con DAGs

Situación actual

- Moi potente
- Menos popular en ensino
- Mantense en contornos avanzados

👉 Linguaxe máis potente para Spark, pero non a máis accesible

---

1.3 Python (a linguaxe dominante na actualidade)

Uso

- PySpark
- Hadoop Streaming
- Dask
- Ray
- Flink (API Python limitada)
- ML distribuído

Por que se usa

- Sintaxe simple
- Dominio en ciencia de datos
- Gran ecosistema (pandas, numpy, ML)

Limitacións

- Non é a linguaxe nativa dos motores
- Overhead pola comunicación coa JVM

👉 Linguaxe principal para ensino e análise

1.4 SQL (linguaxe declarativa)

Uso

- HiveQL
- Spark SQL
- Trino / Presto
- BigQuery / Athena
- ClickHouse

Por que é clave

- Familiar para usuarios
- Permite optimización automática
- Ideal para BI e analítica

Situación actual

- Absolutamente central
- Lingua franca do dato

👉 Non é un motor, pero si a interface máis usada

1.5 C++ (alto rendemento)

Uso

- Motores MPP
- ClickHouse
- Motores de ML

Características

- Máximo control de rendemento
- Baixa latencia

👉 Usado internamente, raramente polo usuario final

1.6 Go (infraestrutura moderna)

Uso

- Kubernetes
- Servizos distribuídos
- Infraestrutura de datos

👉 Linguaxe do contorno, non do ETL

1.7 Rust (tendencia emerxente)

Uso

- DataFusion
- Polars
- Motores analíticos modernos

👉 Posible futuro dos motores

1.8 Linguaxes históricas ou residuais

- **Pig Latin**: linguaxe de alto nivel para Hadoop (obsoleta)
- **Scripts Unix (bash, awk)**: valor histórico e pedagóxico

1.9 Resumo cronolóxico

```
2005–2010 → Java
2010–2015 → Java + Pig + Hive
2015–2020 → Scala + SQL
2020–hoxe → Python + SQL
Futuro    → SQL + Python + Rust (interno)
```

1.10 Resumo por perfil

Perfil	Linguaxes
Infraestrutura	Java, Go
Enxeñaría de datos	Python, SQL, Scala
Analítica / BI	SQL
ML distribuído	Python
Motores internos	C++, Rust

---

2. Word Count en distintos motores e linguaxes

2.1 Hadoop MapReduce (Java)

```java
public static class TokenizerMapper
  extends Mapper<Object, Text, Text, IntWritable> {

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(Object key, Text value, Context context)
      throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString().toLowerCase());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken().replaceAll("\\W+", ""));
      if (word.getLength() > 0) context.write(word, one);
    }
  }
}

public static class IntSumReducer
  extends Reducer<Text, IntWritable, Text, IntWritable> {

  public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable v : values) sum += v.get();
    context.write(key, new IntWritable(sum));
```

```
    }
}
```

2.2 Hadoop Streaming (Python)

mapper.py

```
#!/usr/bin/env python3
import sys, re
word_re = re.compile(r"[A-Za-zÀ-ÖØ-öø-ÿ0-9']+")
for line in sys.stdin:
    for w in word_re.findall(line.lower()):
        print(f"{w}\t1")
```

reducer.py

```
#!/usr/bin/env python3
import sys
cur, acc = None, 0
for line in sys.stdin:
    w, c = line.rstrip("\n").split("\t", 1)
    c = int(c)
    if cur == w:
        acc += c
    else:
        if cur is not None:
            print(f"{cur}\t{acc}")
        cur, acc = w, c
if cur is not None:
    print(f"{cur}\t{acc}")
```

2.3 Pig Latin

```
lines = LOAD '/ruta/input' USING TextLoader() AS (line:chararray);
words = FOREACH lines GENERATE FLATTEN(TOKENIZE(LOWER(line))) AS word;
grouped = GROUP words BY word;
wc = FOREACH grouped GENERATE group AS word, COUNT(words) AS count;
STORE wc INTO '/ruta/output' USING PigStorage('\t');
```

2.4 Hive (SQL)

```
CREATE EXTERNAL TABLE lines(line STRING)
STORED AS TEXTFILE
LOCATION '/ruta/input';

SELECT word, COUNT(*) AS cnt
FROM (
  SELECT explode(split(lower(line), '\\s+')) AS word
  FROM lines
) t
WHERE word <> ''
GROUP BY word;
```

2.5 Spark (PySpark DataFrame)

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, split, lower, col

spark = SparkSession.builder.getOrCreate()
df = spark.read.text("hdfs:///ruta/input")

wc = (df.select(explode(split(lower(col("value")), r"\\s+")).alias("word"))
        .where(col("word") != "")
        .groupBy("word").count())

wc.write.mode("overwrite").csv("hdfs:///ruta/output")
```

2.6 Spark (Scala RDD)

```
val rdd = sc.textFile("hdfs:///ruta/input")
val wc = rdd.flatMap(_.toLowerCase.split("\\s+"))
            .filter(_.nonEmpty)
            .map(w => (w, 1))
            .reduceByKey(_ + _)
wc.saveAsTextFile("hdfs:///ruta/output")
```

2.7 Spark SQL

```
SELECT word, COUNT(*) AS cnt
FROM (
  SELECT explode(split(lower(value), '\\s+')) AS word
  FROM lines
) t
```

```
WHERE word <> ''
GROUP BY word;
```

2.8 Flink SQL (conceptual)

```
SELECT word, COUNT(*) AS cnt
FROM (
  SELECT LOWER(w) AS word
  FROM lines, LATERAL TABLE(STRING_SPLIT(line, ' ')) AS T(w)
) t
WHERE word <> ''
GROUP BY word;
```

2.9 Trino / Presto (SQL)

```
SELECT word, COUNT(*) cnt
FROM (
  SELECT w AS word
  FROM your_table
  CROSS JOIN UNNEST(regexp_split(lower(line), '\\s+')) AS t(w)
) x
WHERE word <> ''
GROUP BY word;
```

2.10 Dask (Python)

```
import dask.bag as db
import re

b = db.read_text("el_quijote.txt")
words = b.flatmap(lambda s: re.findall(r"[A-Za-zÀ-ÖØ-öø-ÿ0-9']+", s.lower()))
wc = words.frequencies()
print(wc.compute()[:20])
```

2.11 Ray (Python)

```
import ray, re
from collections import Counter

ray.init()
```

```
@ray.remote
def count_words(lines):
    c = Counter()
    for s in lines:
        c.update(re.findall(r"[A-Za-zÀ-ÖØ-öø-ÿ0-9']+", s.lower()))
    return c

lines = open("el_quijote.txt", encoding="utf-8",
errors="ignore").read().splitlines()
chunks = [lines[i:i+5000] for i in range(0, len(lines), 5000)]
parts = ray.get([count_words.remote(ch) for ch in chunks])

total = Counter()
for p in parts:
    total.update(p)

print(total.most_common(20))
```

---

3. Mensaxe clave para o alumnado

> Non escolles a linguaxe porque sexa distribuída,
> senón porque o motor distribuído a executa.