Ingeniería Industrial
FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

INTRODUCTION TO BIG DATA

Juan D. Velásquez
Felipe E. Vildoso

Ingeniería Industrial

FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

CHAPTER 2

Ingeniería Industrial
FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

LECTURE 12

Chapter 2

Ingeniería Industrial
FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

MAPREDUCE EXAMPLES

IMPLEMENTING NUMBER OF PAGEVIEWS OVER TIME

```
function map(record) {
        key = [record.url, toHour(record.timestamp)]
        emit(key, 1)
}
```

```
function reduce(key, vals) {
        emit(new HourPageviews(key[0], key[1], sum(vals)))
}
```

IMPLEMENTING GENDER INFERENCE

```
function map(record) {
        emit(record.userid,        normalizeName(record.name))
}
```

```
function reduce(userid, vals) {
        allNames = new Set()
        for(normalizedName in vals) {
                allNames.add(normalizedName)
        }
```

IMPLEMENTING GENDER INFERENCE

```
maleProbSum = 0.0
for(name in allNames) {
        maleProbSum += maleProbabilityOfName(name)
}
maleProb = maleProbSum / allNames.size()
if(maleProb > 0.5) {
        gender = "male"
}
```

IMPLEMENTING GENDER INFERENCE

```
        else {
                gender = "female"
        }
        emit(new InferredGender(userid, gender))
}
```

IMPLEMENTING INFLUENCE SCORE

The influence-score precomputation is **more complex** than the previous two examples and requires **two MapReduce** jobs to be chained together to implement the logic.

The idea is that **the output of the first MapReduce job is fed as the input to the second MapReduce job**.

IMPLEMENTING INFLUENCE SCORE

The code is as follows:

```
function map1(record) {
        emit(record.responderId, record.sourceId)
}
```

IMPLEMENTING INFLUENCE SCORE

```
function reduce1(userid, sourceIds) {
        influence = new Map(default=0)
        for(sourceId in sourceIds) {
                influence[sourceId] += 1
        }
        emit(topKey(influence))
}
```

IMPLEMENTING INFLUENCE SCORE

```
function map2(record) {
        emit(record, 1)
}
```

```
function reduce2(influencer, vals) {
        emit(new InfluenceScore(influencer, sum(vals)))
}
```

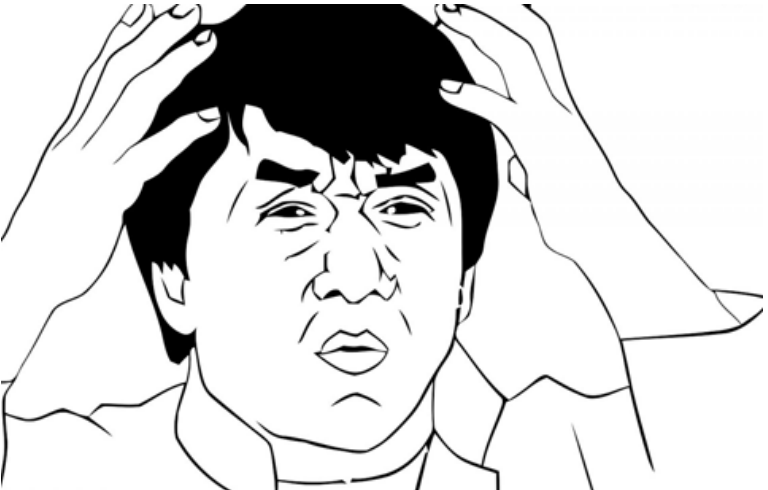
Ingeniería Industrial
FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

LOW-LEVEL NATURE OF MAPREDUCE

LOW-LEVEL NATURE OF MAPREDUCE

Unfortunately, although MapReduce is a great primitive for batch computation—generic, scalable, and fault-tolerant—it doesn't lend itself to particularly elegant code.

You'll find that **Map-Reduce programs written manually** tend to be **long**, **heavier**, and **difficult to understand**.



MULTISTEP COMPUTATIONS ARE UNNATURAL

The influence-score example showed a computation that required two MapReduce jobs.

Running a MapReduce job requires more than just a mapper and a reducer, **it also needs to know where to read its input** and **where to write its output**.

That is the difficult part.

You'd need to **clean up the intermediate output** to prevent it from using up valuable disk space for longer than necessary.

JOINS ARE VERY COMPLICATED TO IMPLEMENT MANUALLY

Let's implement a join via MapReduce.

Suppose you have two separate datasets:

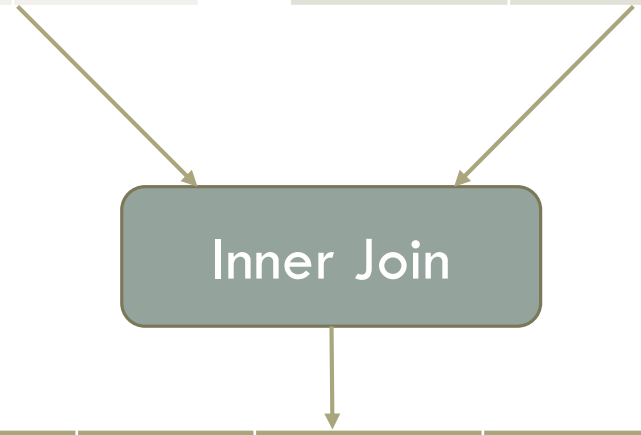
- One containing records with the fields id and age.
- Another containing records with the fields user_id, gender, and location.

You wish to compute, for every id that exists in both datasets, the age, gender, and location

This operation is called an **inner join**.

id	age
3	25
1	71
7	37
8	21

user_id	gender	location
1	m	USA
9	f	Brazil
3	m	Japan



id	age	gender	location
1	71	m	USA
3	25	m	Japan

INNER JOIN

To do a join via MapReduce, **you need to read two independent datasets in a single MapReduce job**, so the job needs to be able to distinguish between records from the two datasets.

MapReduce frameworks typically provide context as to where a record comes from, so we'll extend our pseudo-code to include this context.

INNER JOIN

```
function join_map(sourcedir, record) {
        if(sourcedir=="/data/age") {
                emit(record.id, {"side" = "l", "values" = [record.age]})
        } else {
                emit(record.user_id,
                        {"side" = "r",
                         "values" = [record.gender, record.location]})
        }
}
```

INNER JOIN

```
function join_reduce(id, records) {
        side_l = []
        side_r = []
        for(record : records) {
                values = record.get("values")
                if(record.get("side") == "l") {
                        side_l.add(values)
                }
```

INNER JOIN

```
                else {
                        side_r.add(values)
                }
        }
        for(l : side_l) {
                for(r : side_r) {
                        emit(concat([id], l, r), null)
                }
        }
}
```

INNER JOIN

Although this is not a terrible amount of code, it's still quite a bit of grunt work to get the mechanics working correctly.

There's complexity here: **determining which side of the join a record belongs to is tied to specific directories**, so you have to tweak the code to do a join on different directories.

MapReduce **forcing** everything to be in terms of **key/value** pairs feels **inappropriate for the output of this job**, which is just a list of values.

INNER JOIN

This is only a simple two-sided inner join joining on a single field.

Imagine joining on multiple fields, with five sides to the join, with some sides as outer joins and some as inner joins.

Obviously we don't want to manually write out the join code every time, so you should be able to specify the join at a higher level of abstraction.

LOGICAL AND PHYSICAL EXECUTION TIGHTLY COUPLED

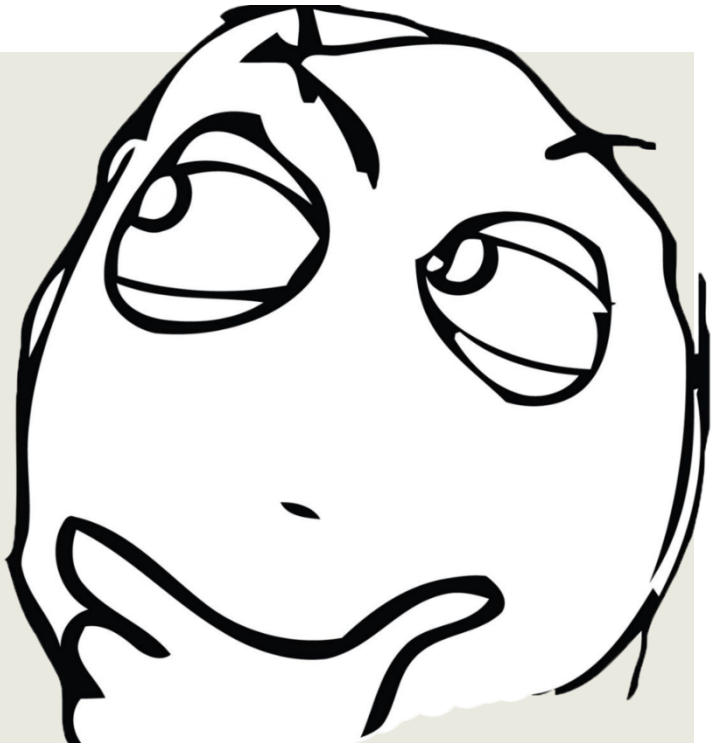
Let's extend the word-count example to filter out the words ***the*** and ***a***, and have it emit the **doubled** count rather than the count.

```
EXCLUDE_WORDS = Set("a", "the")
function map(sentence) {
    for(word : sentence) {
        if(not EXCLUDE_WORDS.contains(word)) {
            emit(word, 1)
        }
    }
}
```

LOGICAL AND PHYSICAL EXECUTION TIGHTLY COUPLED

And the reduce function:

```
function reduce(word, amounts) {
        result = 0
        for(amt : amounts) {
                result += amt
        }
        emit(word,result * 2)
}
```



LOGICAL AND PHYSICAL EXECUTION TIGHTLY COUPLED

That code works, but it seems to be **mixing together multiple tasks into the same function**.

Good programming practice involves separating independent functionality into their own functions.

You could split this code so that each MapReduce job is doing just a single one of those functions.

LOGICAL AND PHYSICAL EXECUTION TIGHTLY COUPLED

But a MapReduce job implies a specific physical execution:

- First, a set of mapper processes runs to execute the map portion, then disk and network I/O happens to get the intermediate records to the reducer.
- Second, a set of reducer processes runs to produce the output.

LOGICAL AND PHYSICAL EXECUTION TIGHTLY COUPLED

Modularizing the code would create more MapReduce jobs than necessary, making the computation hugely inefficient.

So you have a tough **trade-off** to make—either weave all the functionality together, engaging in bad software-engineering practices, or modularize the code, leading to poor resource usage.

LOGICAL AND PHYSICAL EXECUTION TIGHTLY COUPLED

Modularizing the code would create more MapReduce jobs than necessary, making the computation hugely inefficient.

So you have a tough **trade-off** to make—either weave all the functionality together, engaging in bad software-engineering practices, or modularize the code, leading to poor resource usage.

In reality, you shouldn't have to make this trade-off at all and should instead **get the best of both worlds**…

Ingeniería Industrial

FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

QUESTIONS?

SEE YOU ON THE NEXT CLASS!