ANSWERS

A13.7 Functional programming and big data

Functional programming · A level · AQA 7517 4.11.1, Eduqas A500QS 1.4 · about 25 min

What this lesson is about

Volume, velocity and variety, why big data needs distributed processing, and how immutability, statelessness and higher-order functions make MapReduce work.

- What makes data big
- When one machine is not enough
- Why functional programming fits
- MapReduce

Questions

6 marks in all

1. Which three features are used to describe big data? [1 mark]

Tick every answer that is true.

- ☐ **A** Volume
- ☐ **B** Velocity
- ☐ **C** Variety
- ☐ **D** Visibility

> **Answer:** A, B, C. Volume is how much there is, velocity how fast it arrives, and variety how many different forms it takes.

2. Why does processing big data usually have to be distributed? [1 mark]

- ☐ **A** The data is too big to fit on, or be processed by, a single server
- ☐ **B** Relational databases cannot store numbers
- ☐ **C** Functional languages can only run on more than one machine
- ☐ **D** Distributed processing needs no network

> **Answer:** A. When the volume is too great for one machine, the data is split across many that work at the same time.

3. Which features of functional programming make it easier to write correct distributed code? [1 mark]

Tick every answer that is true.

- ☐ **A** Immutable data structures
- ☐ **B** Statelessness
- ☐ **C** Higher-order functions
- ☐ **D** Global variables shared by all machines
- ☐ **E** Updating data in place

> **Answer:** A, B, C. Shared, changeable data is exactly what makes distributed imperative code hard to get right.

4. How does statelessness help when a job is spread over many machines? [1 mark]

- ☐ **A** A pure function gives the same result on any machine in any order, so chunks can be processed in parallel and failed work simply rerun
- ☐ **B** It means the machines do not need any memory
- ☐ **C** It stops the data from being split into chunks
- ☐ **D** It makes every machine run the program one after another

> **Answer:** A. With no state to depend on or change, it does not matter which machine does a piece of work, when, or how many times.

5. In MapReduce, partial results can arrive in any order and in any grouping. Which combining function is safe to use in the reduce stage? [1 mark]

- ☐ **A** Adding counts
- ☐ **B** Subtracting one count from another
- ☐ **C** Taking the mean of two partial means
- ☐ **D** Keeping whichever result arrived first

> **Answer:** A. Addition is associative and commutative, so any grouping and order give the same total. The others give different answers for different splits or orders.

6. In a graph schema, what does an edge represent? [1 mark]

- ☐ **A** A relationship between two nodes
- ☐ **B** A value stored about a node
- ☐ **C** One entity, such as a robot
- ☐ **D** A single timestamped fact

> **Answer:** A. Nodes are the entities, edges the relationships between them, and properties the values stored about a node.

The task: count the fleet's events

Three machines each hold one chunk of a fleet's event log (in the starter). Each line is `<robot>,<event>`, such as `"bot2,bump"`. Count how many times each event happens, MapReduce style: - `to_counts(line)` returns a dictionary with one key, the event, and the value 1: `to_counts("bot2,bump")` is `{"bump": 1}`. - `merge(a, b)` takes two dictionaries of counts and **returns a new dictionary** holding every event in either, with the counts added. It must not change `a` or `b`, so there must be no `d[key] = ...` assignment or `update` anywhere in the program: build the new dictionary with a dictionary comprehension. - `machine(chunk)` is what one machine does: map `to_counts` over the chunk's lines and fold the results together with `merge`, starting from `{}`. - Map `machine` over `CHUNKS`, then fold the three partial results together with `merge`. Print the totals one per line, events in alphabetical order, as `<event> <count>`, such as `bump 4`. Then print `same as one machine: True` if the result equals `machine` applied to all the lines joined into one tuple (it should).

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()
from functools import reduce

CHUNKS = (
    ("bot1,bump", "bot2,tag", "bot1,tag", "bot3,stall"),
    ("bot2,bump", "bot3,tag", "bot1,bump"),
    ("bot3,tag", "bot2,stall", "bot1,tag", "bot2,bump"),
)

counts = {}
for chunk in CHUNKS:
    for line in chunk:
        event = line.split(",")[1]
        counts[event] = counts.get(event, 0) + 1
print(counts)
```

The hint students can ask for: Test merge on its own first with two small dictionaries, including an event that is only in one of them. The keys of the new dictionary are every key in either. Once merge works, machine is one fold, and so is combining the machines.

A solution

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()
from functools import reduce

CHUNKS = (
    ("bot1,bump", "bot2,tag", "bot1,tag", "bot3,stall"),
    ("bot2,bump", "bot3,tag", "bot1,bump"),
    ("bot3,tag", "bot2,stall", "bot1,tag", "bot2,bump"),
)

def to_counts(line):
    return {line.split(",")[1]: 1}

def merge(a, b):
    return {k: a.get(k, 0) + b.get(k, 0) for k in set(a) | set(b)}
```

```
def machine(chunk):
    return reduce(merge, map(to_counts, chunk), {})

totals = reduce(merge, map(machine, CHUNKS), {})
print("\n".join(map(lambda k: k + " " + str(totals[k]), sorted(totals))))

all_lines = reduce(lambda a, b: a + b, CHUNKS, ())
print("same as one machine:", machine(all_lines) == totals)
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.