This exam is **open-book**. Study materials and your own notes are allowed. Pocket calculators are allowed.

Write your **name and student number** on the first page (below this box). In case any of the pages get detached from the rest, also write your student number in the header of each sheet of paper.

Each question is marked with a number of percentage points (for example, 10 %), and they add up to 90 % (10 % is given by default).

Give your answers **on this paper** in the space provided. Handwrite neatly. Design your answer on **scratch paper before** you start writing here, or you may run out of space on this paper! All questions require you to explain your answer. The explanation must be **correct and complete** in order to get all the points for that question. Partial points are possible.

Name and student number: _____

Question 1 *(10 %)*

This is about storing big files on a computing cluster.

In the distributed file system that you read about (the **Google File System, GFS**), which type(s) of **write operation(s)** can a client application perform on files? In other words, what does the GFS support in terms of writing into files?

Say what you know about the write operation(s) supported, in your own words. For example: where in the file can the writing be done? how much data can be written there? by how many client applications at the same time?

Answer:

Question 2 *(11 %)*

Here is a more concrete question, still about storing large data. Take these (approximate) latency numbers that you've seen before ("ns" means nanoseconds):

DRAM reference:	100 ns
Read/write 1 MB sequentially from DRAM:	250,000 ns
Round trip within data center:	500,000 ns
Disk seek:	10,000,000 ns
Read/write 1 MB sequentially from disk:	20,000,000 ns

To keep it simple, we talk here about a single cluster machine. This machine can fit 1 TB of data on its disk.

Say that 1 TB of data is streaming in at the machine. It needs to be written to the disk, into a file. The dataset contains timestamped data points of fixed size, 1 KB. The data points are not sorted by timestamp, but arrive in somewhat random order. You're trying to figure out what's the more efficient way to store this dataset:

1. Start with a file of size zero. Append the new data points, in the random order in which they are in the dataset.
2. Start with a 1-TB file with no content written in. Write each incoming data point at the correct (sorted) location in this file. (Assume it's already clear at which offset in the file each timestamp should go.)

How much time would each of these two solutions take? (You can be a little approximate.) Which one is faster, and roughly by how much?

Answer:

Question 3 *(14 %)*

Here are brief questions about the Spark framework. Answer each question, explaining briefly how you got to your answer.

(a) What type of dependencies (narrow or wide) do the following Spark methods have, and why?

```
map, sample, repartition, sortByKey, coalesce
```

(From the API: `sample` returns a sampled subset of the RDD; `coalesce(numPartitions)` returns a new RDD that is reduced into `numPartitions` partitions.)

Answer:

(b) If a Spark program looks roughly like this sequence of methods:

```
.textFile().flatMap().map().groupByKey().map().sortByKey().saveAsTextFile()
```

can you estimate how many *stages* of execution Spark will create?

Answer:

(c) If, during the execution of a Spark job, one worker machine fails (breaks down or becomes unreachable), does Spark guarantee that the job will still obtain correct results? Why or why not?

Answer:

Question 4 *(13 %)*

Take a typical Spark program that you have worked with (here, a word count):

```
1 rdd = sc.wholeTextFiles("...")
2
3 words = rdd.flatMap(lambda (file, contents): contents.lower().split())
4             .map(lambda word: (word, 1))
5             .reduceByKey(lambda a, b: a+b)
6
7 word_counts = words.collect()
```

Say that the input data is big, so you run this program on a computing cluster with the command `spark-submit --master yarn --deploy-mode cluster program.py`.

Knowing how the Spark framework distributes a job: **where will each line of code above actually execute**, on the cluster? Explain how you arrived at that answer.

You're not given the names of the machines, but you can refer to them by the standard Spark terms *driver*, *master*, *worker machine(s)*, and/or *executor(s)*. So, an answer can be formulated something on the lines: "Line 1 will execute on the driver, because...". It is possible that some line of code is complex and executes in multiple steps (in that case, say how each step might execute, as much as you know).

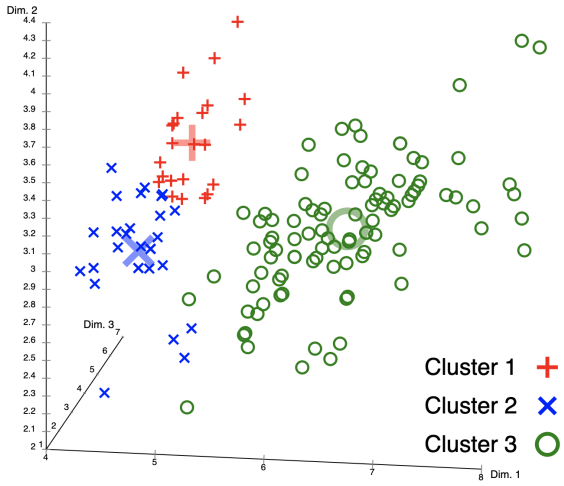
Answer:

Question 5 *(18 %)*

Sketch a distributed Spark solution to a basic data-clustering problem, with big data in input.

The data in input is a large set of N data points in d dimensions (in the picture below, $d = 3$ dimensions are drawn). This data is available in some plain-text files at some HDFS path P, one data point per line. The line has the format $x_1, x_2, ..., x_d$ (x_1 is the value of that data point in the first dimension, etc.).

You are also given a number k: the number of clusters needed (below, $k = 3$ clusters are shown).



The algorithm starts with an initialisation: k random data points from the input data are selected as cluster "representatives". Then, the algorithm proceeds with two basic steps:

1. Assign each data point to a cluster. The cluster chosen is that whose cluster representative is closest to this data point in terms of Euclidean distance. (You can also pick any other distance in d dimensions.)
2. Recalculate the cluster representatives. For any cluster, the new representative is simply the mean point in space (the centroid) for the data points currently assigned to that cluster. (The centroid doesn't need to correspond to a real data point; it can fall anywhere.)

The algorithm iterates these basic steps a small number of times, which is decided by the user. (This clustering method is called "naïve k-means", one of the many variants of k-means clustering.)

Your answer should include: (a) Spark code or pseudocode for this clustering method (the code syntax can be approximate, and any Spark library is fine); (b) A statement saying whether your method has any drawbacks or limitations.

Answer:

Answer (continued):

Question 6 *(12 %)*

As you have learned, Kafka is an open-source stream-processing platform that can broker messages (or events) between producers and consumers. Messages are stored in so-called topics. Data can be replicated and partitioned.

(a) What is the advantage of having multiple partitions for a topic?
Answer:

(b) Does it make sense to have more partitions than brokers? Why so/why not?
Answer:

(c) Does it make sense to have more replicas than brokers? Why so/why not?
Answer:

Question 7 *(12 %)*

Stream-processing engines can provide various guarantees about end-to-end processing in the event that failures such as network outages occur. Spark Structured Streaming offers end-to-end exactly-once semantics under any failure, provided that the data source is "replayable" and that the data sink is "idempotent," which means that operations that are applied multiple times do not change the result.

(a) Explain why it is essential that data sources are "replayable".
Answer:

(b) What role do offsets play in this process?
Answer:

(c) Imagine that you use Kafka as data source for your Structured Streaming query in combination with an idempotent sink. The connection to the Kafka broker is affected by a network outage. As Kafka topics have an offset and are replayable, you will enjoy Spark's guarantees when the connection is restored. However, the provided guarantee may not hold indefinitely. Explain why not.
Answer: