Exam materials

Exam procedure

We have the open-book exam on **Tue 31 Jan**, morning. I booked a 3-hour slot, but you can expect to finish in under 2 hours.

You can bring paper materials with **your own notes**! Avoid other printouts without your own notes: avoid printing code (no way to say whose code that is) or exam answers from other years (they are not your work).

Pocket calculator allowed. No other electronics allowed.

On the Chromebook, there will be 3 open tabs:

- Anubis (https://anubis.dacs.utwente.nl/) , an exam software made by our students
- papers and slides from Canvas will also be available in a browser tab open to this exact link (https://www.utwente.nl/en/ces/sal/exams/intranet/books/Managing-Big-Data/) (you can also read through beforehand)
- this digital chat will also be available (a Canvas BBB conference) (https://bbb.dacs.utwente.nl/b/buc-yuu-c0v-xgl) (no login needed, only type a name), where you can receive public announcements, and you can also click on the teacher's name and "private chat" any questions (it will help a lot to keep the room quiet; will be started only on the day of the exam)

Examples of exams (with answers)

Last year, the students got this set of exam questions on paper, both in the **regular exam** (Exam version 1 (https://canvas.utwente.nl/courses/10468/files/3397962?wrap=1) (https://canvas.utwente.nl/courses/10468/files/3397962/download?download_frd=1)) and this set in the **resit** (Exam version 2 (https://canvas.utwente.nl/courses/10468/files/3397963?wrap=1) (https://canvas.utwente.nl/courses/10468/files/3397963/download?download_frd=1)). The type of questions will still hold for digital exams.

Answers to Exam version 1 (https://canvas.utwente.nl/courses/10468/files/3397962?wrap=1) (https://canvas.utwente.nl/courses/10468/files/3397962/download?download_frd=1)

Q1.

This talks about "one big file".
[5%] Check disk space. Total disk space is: N*D. You must add the replication factor (call it R, whatever value it may have); you should know you need it. Max file possible as far as disk allows: N*D/R.
[5%] Check master memory (only the master stores metadata!). M/T chunk metadata items

possible to store there. This means max file possible as far as memory allows: C*M/T.
[1%] Altogether: whichever of the above is smaller, so min(ND/R, CM/T).

Q2.

a) Quite some options: *spark-submit* without arguments, or with *--master local*. Even *pyspark* if you paste the program in. Executes where you're logged in. Input data from disk (assuming HDFS, the default for big data) is distributed, comes from anywhere on cluster. Output is "to disk", the question says (so save(), not collect()), so again goes anywhere on cluster (where, decided by the HDFS master, not by the Spark driver), and replicated!

b) Variants of a), definitely with *--master yarn,* so different arguments to *spark-submit* (Lecture 4). Executes distributedly, depending on how many executors were configured. Data always distributed, data locality is maximised.

c) Not many, but a subset of: stdout can be used as log, may run quicker because no network is used (...after the data is brought in though!) since shuffles are local. It's easier to debug and include external libs.

d) Can crunch big data, fast!

Q3.

a) Spark avoids disk I/O for intermediate results: caches them in memory if possible. MR would have to save to/load these results from disk buffers after each Map task, and from DFS (disk) files after each MR iteration!

b) Spark SQL adds a query (code) optimizer, which benefits from DataFrames having a known schema. I expect some words on Catalyst. Spark core is blind to the internal structure of the data, so cannot use it for code optimisation (it does optimise, but less, by its lazy execution). Note that Spark SQL on the other hand loses some time in doing schema inference for the .csv, as you probably saw in the project (unless you reformatted to .parquet).

c) Explain that *actions* trigger executions. Only 1 and 3. Nothing to do with repartitioning (repartition() is a transformation, so lazy) or wide dependencies (those are still transformations).

Q4.

a) GFS, Bigtable. Also Kafka can store, but only for a limited (configurable) time.

b) GFS: record append (a special type of append) to the last chunk of the file, and random writes at a given offset in the file.
Bigtable: write inside a single row (single-row mutation), by row key. It can also iterate (scan) over rows, after it located a start row by row key (this may be used more often for reading than for writing).

c) It's hard to argue for storing this structured data in plain GFS files (possible, but very inefficient: you'd have to build an index, and deletions of emails would need file compactions). Random writes needed (additions and deletions). Email is also read occasionally, so reads will also be needed (but this applies to any data). So Bigtable it is. How would you organise the data?

Mention (or draw, like the Web table): which row key you'd choose, column families, columns. There are many reasonable options. Hard requirements: The *row key* must be sortable for data locality, and unique. The row itself should ideally remain small data, or if not possible a (row x column family) should remain small data. You can use 2 Bigtables. Points for: choice of framework and reasoning (3%), and organisation idea (3%: row key, column families, columns should be mentioned with reasonable values; not very picky, but it has to be able to store all the data).

Q5.

a) Column-family data store. This was so easy, it was free points.

b) Indexing of tablets inside Bigtable
Indexing inside tablet -> single disk access
Serving requests from memcache (fast), lazy compactions
Some more (caching tablet locations, block caching, batched reads) if you go into the dirty details (but these are far less significant).

c) Data tablets replicated
Metadata tablets replicated
Even the commit logs are replicated (all of the above end up in GFS files)
That lock service (called Chubby at Google), keeping the namespace, is also persisted and replicated

(3 out of these 4 are sufficient for max. points here.)

Q6.

a) Load balancing for consumers. The more partitions a topic is divided into, the more concurrent consumers can read. The partition is the smallest unit of parallelism.

b) Kafka only allows 1 consumer (from each consumer group) to read from a partition, so essentially as many consumers as partitions. This is a limitation, so they may have to over-partition (make many partitions) if there are many consumers.

c) There's (some) difference: in Spark Streaming, a partition is limited to a time (batch) interval, so probably contains small data compared to a partition in core Spark. The _sequence_ of Spark Streaming partitions in time is roughly equivalent (in data) with the core Spark partition.

d) Spark Streaming has roughly the same set of transformations from core Spark, now called "stateless". On top of these, it adds another set of "stateful" transformations (maybe give an example related to sliding windows) for operations across multiple time intervals.

Q6.

Partial points go for the thinking on the problem. The closer to real code, the higher the points awarded.

1. Computing G amounts to a simple sum (so plain Python is enough), but you need those fractions f_c first. But that's easy: implement a count for the records per class. This is like word

count in core Spark, or in the SQL lib with an aggregation: D.groupBy(['class']).count().collect(). You get two numbers, which you divide by the size of D, D.count(). That's it.

2. D has to be split now in two datasets, which looks easy: apply a filter to get each part of the split. With the smaller datasets ready, you can simply apply the function from 1. again. The fractions w_a, w_b are just a division of RDD/DF sizes (count()).

It would be nice if this point were solved without copying the splits into new RDDs/DFs. This requires writing more code for the Gini Index, and a slightly higher time complexity.

3. Here, "proposing" splits is the issue. You could do the simplest search possible: take each feature column (there are n), compute the average value of each, and propose that as a split. Many other data statistics probably reasonable.

When finally there is complete code: which types of transformations does it use? Many wide dependencies? Long for loops? If not, then it's efficient.

Answers to Exam version 2 (https://canvas.utwente.nl/courses/10468/files/3397963?wrap=1) (https://canvas.utwente.nl/courses/10468/files/3397963/download?download_frd=1)

Q1.

This is the same as Q1 in Exam version 1 above, just with real numbers instead of variable names.

Q2.

The first 4 are transformations, the rest actions. In order, the dependencies: narrow, possibly wide because of that True flag, narrow, narrow.

Q3.

This is easy if you paid attention when coding, and did a lot of debugging in the shell (pyspark), at every stage of a program. For example, line 1 creates an RDD where a record has type: tuple of 2 strings; line 8 returns a plain Python list.

Q4.

This is like Q4 (c) in Exam version 1 above.

(a) This is structured data, so the obvious idea would be to make a Bigtable out of it (you'd have to roughly explain how). However, *"this data will be processed once a day"*, which means a sale (data point) doesn't have to be read/written quickly, so doesn't appear to need indexing. So, you can store it with less effort in big files (semistructured data: csv, json), ideally one file per day to help the processing.

(b) The pictures are pretty large: up to a chunk size. You could store each in one file, then build a database which helps to find these files: a Bigtable holding the coordinates, timestamp, and file path. For the record, images in general can also be stored inside the Bigtable, as a stream of bytes inside a cell (these images are a little large for that, but the block size of the Bigtable can be configured). The assignment doesn't say how often and how quickly the images must be located (often and quickly, like in Google Maps?), but this makes sense in any case.

Q5.

You simply need to choose a row key (a unique string or number), and probably two column families (keywords, pages) with very many columns. Many students weren't able to represent a Bigtable in any form (as a table, or as a map).

Q6.

This one's theory. Kafka and Spark Streaming don't really overlap in functionality: they do different things altogether, so a lot can be said here. Spark doesn't store data except until the end of the program (Kafka does provide a form of storage, and an orderly fashion to provide only the relevant part of the data to a consumer application).

Q7.

This you essentially got as the KNN assignment already! You could try to do regression instead of classification (very similar algorithm, except now all "class" names are replaced with numerical "values". For the new point, compute the most likely value, namely: the average of the values of the k nearest points. (It's unlikely that you'll get a task so close to an assignment though.)