Exam materials (with answers)

Example exams (we will keep roughly this format of the exam sheet)

General idea. Reading the exam questions, you will see that most are phrased in quite simple words ("How is data stored?", "How do these frameworks differ in speed?"). This is because you won't be asked many questions about the details of the big-data frameworks you know. Instead, you're asked exactly the types of questions you may be asked in a *job interview*, so the types of things that really matter. Then, there is always at least one question where you're asked to write some code, also as you would be asked in an interview. For coding questions, syntax doesn't matter (you can even write pseudocode with a Spark flavour). What matters is the *idea* (how to approach the problem), and the *feasibility* of your code for big data (recall in the feedback on your assignments, my occasional comments that a certain line of code is not suitable for big data).

(You can also get the exam sheets from the "Files" menu on the left.)

Example exams below. Every year, the material taught was slightly different, hence some questions being excluded. You **write your answers on the paper, in the empty spaces**. The size of the empty space roughly matches the size of the expected answer, so you don't have space to write much other than what really matters for the question.

- Exam 2018 (Jan 30, 2018) all questions except 5, 7 (write question 4 in core Spark)
- Exam 2019 (Jan 29, 2019) all questions except 7-9 (8 is a perhaps difficult, but very real, case study for Bigtable/HBase, which you can try--in that year, we spent a bit more time on that topic)
- Exam 2020 (Jan 28, 2020) all questions except 5.1

These were *closed-book exams* (the streaming week had no Kafka, but Twitter Storm instead). Some questions are too simple for an open-book exam (not all; some are still hard enough). They all can be used also as quiz questions, so you should ideally feel comfortable answering all of them; they show you what's important to know about a topic.

Last year the exam was *open-book* and also some topics were different (different Kafka version, different streaming library):

- Exam 2021 (Jan 26, 2021) all questions except 6bc, 7.

Answers (or hints) come below.

Answers Exam 2018 (Jan 30, 2018)

Q1.1: True, concurrently, and atomically, but there's only a special kind of "record appends" (see GFS paper) which provides only certain (not all the desirable) performance guarantees.

Q1.2: Not quite true. Don't confuse the chunk index (computed locally out of file offset) with the chuck handle (a sort of pointer, retrieved from master). The client obviously always knows the chunk index, but that's insufficient to read/write.

Q1.3: Mostly true: but! they are not persisted (in RAM), but polled from chunk servers at boot, and updated via polling.

Q2.1: GFS, Bigtable (or their Apache clones).

Q2.2: For GFS: the store for the namespace, mapping, metadata; also the first 2 steps in the read / write process, and also the chunk management (migration, lease, garbage collection). For Bigtable: the store for the location of the root tablet; clients cache tablet locations, so no centralised communication; also decentralised tablet management.

Q2.3: Shadow master (GFS). 5 Chubby replicas (Bigtable).

Q3.1: Not intentionally. It can easily happen by mistake, though, if there are very many records per key, and your code does something like a groupByKey. An RDD is big. Initially, 1 partition = 1 chunk from disk, which means small partitions.

Q3.2: Spark map operates in batch (inputs 1 batch = sets of tuples, and outputs 1 batch); doesn't necessarily read input from disk, nor save output to buffers; takes a function f as a closure.

Q3.3: A computation on a partition, local inside an executor.

Q3.4: Depends on # cores free on cluster, # and size of the files in input. A good guess: min(# files, chunks, or partitions of input data, # free cores).

Q3.5: It imposes a partitioning on both RDDs, and will likely need to repartition (shuffle) at least one of the inputs.

Q3.6: Pre-partition both RDDs with the same partitioner. This can save time, if the pre-partitioning can be conveniently done somewhere in the program.

Q4: Schematically, do map(URL, text): splits the page into words w and outputs (sorted(w), w). This sorted(w) sorts the characters inside the word, so the word is takes as a list of chars. Then, a reduce by key outputs the set. Bonus points for an added "combiner" in the map.

Q6: Sketch a Bigtable schema that is feasible, and allows those operations (put_tweets, get_topic) to execute reasonably fast. Many solutions possible, with 1, 2, or 3 Bigtables (for example, one for tweets, one for topics, both being kept updated whenever new tweets come, so at put_tweets). You should have the required columns present with some logically named keys. Then, sketch an

implementation for those operations: inside these implementations, all you can use is the Bigtable basic operations: read and write 1 row, given the row key. All the operations must be fast, say, constant time complexity, rather than linear in the # tweets, # users, # hashtags). I definitely needed a separate table for topics to make a fast get_topic (you don't want to have to browse the entire tweet table to compute the topic).

Answers (Jan 29, 2019)

Q1.1: 3 things: (a) the file and chunk namespaces, (b) the mapping from files to chunks, (c) the locations of each chunk's replicas.

Q1.2: RAM, but not only (or it wouldn't be fault-tolerant): All metadata is kept in the master's memory. The first two types (namespaces and file-to-chunk mapping) are also kept persistent by logging mutations to an operation log stored on the master's local disk and replicated on remote machines. 3% was given if only RAM mentioned.

Q1.3: The data will be consistent (all replicas are identical) but undefined (typically, it consists of mingled fragments from multiple mutations). 3% was given if only consistency was mentioned.

Q2: You can get 1 PB (= 1000 TB = 10^15 B) of file on the disks, in 10^7 unique chunks.
You can only get metadata for 10^6 unique chunks in a RAM of 10 GB; this limits the file size to 100 TB.
You need 10 KB metadata per unique chunk, not per replica (the replicas are the same, except e.g., location, version).

Q3.1: Speed is not needed, and the input data can be loaded in the cluster RAM. Either framework is fine.

Q3.2: The input data is larger than the cluster RAM. Because of this, it's more logical to go for MapReduce, which reads the input data and writes the output data to disk. Note that Spark also can be configured to do disk-based storage rather than the default RAM storage (the answer must state this to be taken as correct).

Q3.3: Speed is required, and the input data can be loaded in the cluster RAM, so Spark is most logical. Also, it's much easier to code ML in Spark (it has built-in libraries for this).

Q4.1: One partition of the parent RDD has multiple partitions of the child RDD depending on it. I saw many answers which confuse a type of narrow dependency (one child partition depends on multiple parent partitions) with a wide. We discussed this in the lecture, and it's emphasised in the paper.

Q4.2: Examples: join (only if not co-partitioned), groupByKey, reduceByKey, sortByKey (even sortBy), repartition (with a given partitioner). There may be some others, less frequently used. 2 examples are enough for all the points. Not coalesce!

Q4.3: A function which determines in which partition a record goes; often used to control data locality (which records stay together).

Q4.4: Hash by key (e.g., groupByKey, reduceByKey). Range partitioning (e.g., sortByKey). User-defined partitioning functions possible (e.g., partitionBy).

Q4.5: Spark keeps persistent RDDs (and also regular RDDs, but only before shuffles) in memory by default, but it can spill them to disk if there is not enough RAM. Users can also request other persistence strategies, such as storing the RDD only on disk.

Q4.6: The typing. RDD records are arbitrary Python types. DataFrame: The records are of a fixed structured type, Row(), with Columns of SQL data types.

Q5.1: In the worst case (when the program must process all data at once), max memory = max(size of input RDD, size of intermediate RDDs). The intermediate RDDs may be much larger than the input (if the program does joins where the data explodes). This max. memory should be less than # executors (configurable up to #CPU cores) x executor memory (configured by either you, or dynamically by Spark up to 8 GB, or set to the default: 1 GB). 2.5% points were given if no awareness of executor memory, nor of the potential for large intermediate RDDs.

Q5.2: (d), flatMap.map.reduceByKey. Explaining why was required. Some confused textFile with wholeTextFiles. The score for this subquestion is unfortunately binary: 0 or 5 (no partial points).

Q6.1: Each value in a cell is an uninterpreted array of bytes (or byte string). It's up to the programmer to know how to unpack that. Note that this isn't caused by Bigtable being stored in SortedStringTables; SSTables suit because the row key is a string!

Q6.2: Describe the 2 levels of "indexing": METADATA and the SSTable block index:
(1) root tablet location is cached (so fast), then lookup in a 3-level tablet location hierarchy gives the right tablet server
(2) once located on the tablet server, the SSTable's block index is loaded in memory, binary-searched, then 1 disk seek gives the right block. Add the time complexity if you can figure it out.

Q6.3: Give a quick summary of how/when data is actually written (to memtable, commit log, SSTables on disk).
The various types of compactions are executed in the background. Incoming read and write operations can continue while compactions occur. Emphasise that it's a sort of deferred write.

Answers Exam 2020 (Jan 28, 2020)

Q1.1: 3 major types of metadata: the file and chunk namespaces, the mapping from files to chunks, and the locations of each chunk's replicas.

Q1.2: The question is only on the _data_ flow! (So, not about the control flow). The master tells the client the identity of the primary and the locations of the other (secondary) replicas. The client pushes the data to all the replicas. A client can do so in any order. The control flow (the write command) is apart from the data flow.

Q1.3: Two cases. If the writes are successful, all clients will see the same data, but it may not reflect what any one mutation has written: typically, fragments from multiple mutations (in other words, the region is undefined but consistent). If there's some failure,

inconsistent/undefined (different clients may see different data at different times), and this problem will be solved later.

Q2.1: Bigtable, organised with a suitable row key. On GFS you'd have to make an index yourselves to find the individual records, or otherwise read an entire chunk to find a single record. Full grades in Q2 were given for good written motivations to your answers (so nothing that looks like a guess); the easiest answer is for you to simply compare the two options.

Q2.2: "Add new" is an append in DFS, and is adding a new row or column (depends on the table design) in Bigtable. A DFS has a simple record append with no search whatsoever, so makes more sense. Bigtable is not too bad either, but will be a little slower due to memtable operations and compactions (batched and delayed, so on average fast!).

Q2.3: Both would work: DFS is faster (you read entire chunks backwards from the end of the file just until the dates become too old); Bigtable is good only with a suitably designed(!) row key based on time. Many answers confused Bigtable row keys with Bigtable timestamps, strangely. Cell contents have timestamps by default, not rows. (Cell timestamps are not relevant here, since the new data is completely new emails, not versions of old emails).

Q3.1: Transformations are lazy operations that define a new RDD, while actions launch a computation to return a value to the program or write data to external storage. Note that transformation have either narrow or wide dependencies. Also note that actions do not always result in small data: it is allowed to do rdd.collect() on a big RDD (which may lead to a crash), or rdd.save() which dumps big data to disk.

Q3.2: Actions: first 2.

Q3.3: You have that RDD's lineage graph. Stages are built when the program runs an action. Each stage contains as many pipelined transformations with narrow dependencies as possible. The boundaries of the stages are the shuffle operations required for wide dependencies: draw the boundary right at this shuffle.

Q3.4: DF: has schema, records are of type Row, have typed named Columns; typing is SQL. RDD: untyped records (any Python types, possibly different between records).

Q4.1: The client (not Chubby!) traverses a tree hierarchy (3 levels) of tablet location info. Chubby file -> root metadata tablet -> metadata tablets -> user tablets. (If you know: Row key in metadata tablets = Bigtable identifier+end row.) Many answers here were offtopic, so read the question carefully.

Q4.2: Tablet is stored in SSTable file. SSTable contains a sequence of blocks (64KB). A block index (stored at the end of the SSTable) is used to locate blocks; the index is loaded into memory when the SSTable is opened. A lookup can be performed with a single disk seek: find the block by performing a binary search in the in-memory index, then read the appropriate block from disk.

Q4.3: It's a deferred write, so fast to execute, but takes a while to finalise. On the tablet server: A write goes to the commit log. Its contents are inserted into the memtable (stored in memory, a sorted buffer). When the memtable reaches a threshold, it is converted to an SSTable and written to GFS (minor compaction). Periodically: a merging compaction in the background; it reads the contents of a few SSTables and the memtable, and writes out new SSTables.

Q4.4: The root metadata tablet can hold 2^17 records. There can thus be at most 2^17 metadata tablets, each of 2^17 records, so in total 2^34 records. Each can point to one data tablet (of 128 MB=2^27 B), so all data tablets in total can hold 2^61 B, or 2 exabytes.

Q5.2: A time series of RDDs of the same type, essentially (computed by the same lines of code, through time). In more words: In each time interval, the records that arrive are stored reliably across the cluster to form an immutable, partitioned dataset. This is then processed via deterministic parallel operations to compute other such datasets. Each series of datasets forms one D-Stream.

Q5.3: There are 3 if you remember to include the result of the read operation. 2 if you forget that (you still get some points). A slide in Lecture 6 essentially answers this with a picture.

Q6: Schematically, for **Q6.1**: Load v first in memory on all machines (collect() or broadcast() or just Python open()). One map() reads a triple (i,j,Mij) and simply transforms it into (i, Mij x vj). One reduceByKey() then adds up all values for a given key i, and thus gets res[i]. Done!
For **Q6.2**: split v into smaller vectors, and explain how to view M accordingly as small squares or stripes, so its parts can be multiplied with the small vectors. From there, as above, finished with an extra reduce. In general: Solutions that are sequential in nature, to any extent, obviously don't quite suit big data (the more for-loops I see, the lower the grade). DataFrame _Rows_ or RDD _records_ CANNOT store big data! unless you argue the sparseness of M. (Interested in the whole story of "big" PageRank computation with MapReduce-style logic? see http://infolab.stanford.edu/~ullman/mmds/ch5.pdf, section "5.2 Efficient Computation of PageRank". Many more details there: M is sparse, etc. but no code.)

Exam Answers 2021 (Jan 26, 2021)

Q1: An easy job _if_ you read the GFS paper (first 2 sections), which builds a story around the idea "data mutations may be writes or record appends". That's it. Describe a bit what a random write and record append are, from the point of view of a file (random write: small write anywhere in the file). How much data can be written there? No hard limit except record size and cluster capacity, but in practice random writes are small and appends are large. A mention of concurrency is expected here, and ideally also a sentence as to whether writing works reliably when multiple clients write at the same location at the same time (data is always written, except at failure of the operation, but writes can be written in different orders in different replicas, which may overwrite data; even appends can be inconsistent!).

Q2: This is meant to support Q1 with numbers, but can be solved independently without knowing about the DFS. It's about the same as the first practical (which had this for read operations).

There are N = 10^9 data points to be written, each of size 1KB.
Writing 1 MB to disk sequentially takes 20 ms. Seeking for the right position to write one data point takes 10 ms (and actually

writing it takes a bit more, but only 20 microseconds, namely the write time for 1 MB / 1000, so I'd ignore it because it's an entire order of magnitude lower).

Append time: 10^6 * 20 ms = 20000 s =~ 5.5 hours (approx)
Random write time: 10^9 x 10 ms (3 more zeros than the above!) = 10^7 s = 115 days ??

Append on 1 machine is about 500 times faster. Note: you always must write to disk! If you used only the DRAM times (which are negligible in comparison), that's not enough (you still need to add the disk time).

Writing in that sorted fashion is really just impossible: the data streaming in probably outpaces the writing process :(Only appends have a chance of being fast enough!

Q3:

a) 1% for each (map, coalesce generally have 2 cases: map w/o repartitioning, and coalesce with lower numPartitions are the expected ones, so it's OK if you only talk about those cases; may give bonus points for a discussion of all cases though). Answers: nnwwn. Coalesce with a reduction of #partitions needs no shuffle. Reasons given must follow the definition of narrow/wide dependency, not be some random statement about the method.

b) For the inner logic of the program, .flatMap().map().groupByKey().map().sortByKey() , clearly 2 stages ending at wide dependencies. The save is likely a separate stage ending in an action and ending the program. The read with textFile() at the beginning may or may not be a stage, so either answer is acceptable (in the dashboard for distributed jobs, you do see the initial read as a separate stage). Answers between 2 and 4 are reasonable, with an OK explanation.

c) It does guarantee that all data is processed, by recomputing the tasks that were lost, and any prerequisites for that: "If a task fails, we re-run it on another node as long as its stage's parents are still available. If some stages have become unavailable (e.g., because an output from the "map side" of a shuffle was lost), we resubmit tasks to compute the missing partitions in parallel" (quote from the RDD paper.) The exactly-once guarantee from the streaming libs are also relevant here.

Q4: Any lines in the program before the code given (so, creating a Spark context / session, function or global variable definitions) happen on the driver. All lines except collect() run (when they eventually run) distributedly on executors (a more precise word than machines/servers) and don't process anything on the driver (the driver does all the control, little of the execution). With details though:

-- wholeTextFiles in distributed ('yarn', not 'local') mode is not expected to create any network data communication (the code comes to the data, so should all run distributedly on executors, where chunk replicas already reside).
-- map, flatMap: expected to continue running on the same executors as above.
-- reduceByKey is a big deal, a very overloaded line: does some executor grouping+local reducing, then shuffles data on the network, then (probably different) executors finish up the reducing (the executors can change between stages, as this transformation separates stages).
-- collect starts on executors, where the data is in partitions, and communicates it centrally to the driver, where it gets put into a Python list (is centralised in the end).

Q5: The initialisation step can be trivial (since any k points are acceptable under some assumptions, this need not be big-data code, but even plain Python is OK: take the points on the first k lines in the input, for example).

1. Assignment of points to clusters. Give each cluster an identifier. Assume k is small for this to work. With the current k cluster representatives known, .map (or similar) each data point to a new record which includes the data point itself, but also k distances to the k cluster representatives. Then .map (or similar) to keep from those k distances only the smallest one. The identifier of that cluster is the cluster assignment of this point, and should be kept in the record. Essentially like KNN, but with multiple distances.

2. Recalculation of cluster repr. This one sounds easy (just a multidimensional mean() among all points assigned to a cluster), but it's a bit tricky if you go for the wrong methods. for i in range(k):.filter(i).mean() is fine, if ugly and only for tiny k. Anything like rdd.groupBy() is not scalable, because the points for 1 cluster are likely big data, and this method aggregates them in one record. On the other hand, reduceBy() and similar are fine, because they reduce these grouped lists (with the associative function f given) already on executors, for optimisation. Spark SQL aggregation methods are also fine (they're similar to reduce() methods).

Lots of details need to be decided: how the record looks like (some dictionary or list), where the for loop will be (to loop through k clusters, so a short loop if k is small). Limitations: I expect to hear in your answers whether k and d can be a big numbers, and any numerical problems (very large sums?).

Q6.a: Dividing a topic into partitions with distribution over brokers (one broker can hold many partitions of the same topic) means that multiple 'clients' (consumers) can be independent (e.g., one topic can still be consumed by many clients at the same time).

◂ Previous