SAMPLE QUESTIONS - DATA ENGINEERING MODULE

Note: Number of MCQs/Descriptive questions not exactly same as Final exam

For exact details on Final exam questions and points breakup, refer to our email communication.

1. Why do we require parallel and distributed computing?

(a) The clock speed of CPUs has reached an upper bound. So more machines are required rather than faster machines to solve problems.
(b) The number of cores per machine has been increasing. So we need to be able to use them collaboratively to solve larger problems.
(c) It is easier to write parallel/distributed programs rather than sequential programs.
(d) Distributed computing is well suited to solve both small and large problems sizes.
Answer: (a and b)

2. Which of the following statements about GFS are FALSE

(a) GFS is designed for low latency storage and access to data
(b) GFS ensures high reliability by replicating blocks of a file
(c) GFS ensures high reliability by partitioning files into multiple blocks
(d) GFS is well suited to store millions of small HTML pages from a web crawl as separate files
Answer: (a, c and d are FALSE)

3. Which of these statements about Hadoop MapReduce and Spark are TRUE

(a) Conceptually, the shuffle phase in MapReduce and Spark are similar
(b) The map function is Hadoop is similar to the flatMap function in Spark
(c) The reduce function is Hadoop is similar to the reduce function in Spark
(d) Hadoop's Map and Reduce functions are more expressive than Spark's transformations
Answer: (a, b are TRUE)

Please answer in 5-10 lines.

1. Discuss the limitation imposed on strong scaling by Amdahl's law.

Answer: Amdahl's law imposes limitation on the speedup if the problem size is constant and the number of processors increase, for a given application. The parallel speedup is limited by the serial fraction of the application. Even if the serial fraction is small, the strong scaling speed that can be achieve is only 1/(s + (1-s)/p), where s is the serial fraction and p is the number of parallel processors. Even for small values of s=0.05, this speedup value is limited for large values of p, such as speedup=16 for p=100 while the ideal speedup should be 100. So strong scaling speedup (or efficiency) is low when the processors are high.

2. How is Record Append in GFS different from a regular append of data to a file? How is it more useful?

Answer: Record Append is performed internally by GFS, and GFS decides the offset at which the append happens. Regular append is done by the client, with the client directly deciding the position for appending the data.

Record Append allows concurrent clients to perform an append to a file while ensuring that the data is not over-written by each other. This is done using an atomic append operation that is internally coordinated by GFS data nodes. So the key benefit is the ability to perform concurrent append-mutations by clients.

3. Dynamo uses Sloppy Quorum where the number of replica reads (r) and replica writes (w) that must be successful for a given operation is such that (r+w > N), where N is the number of replicas for the key-value pair. Why is this condition required?

Answer: In Dynamo, consistency is not guaranteed by the system. Rather the system maintains sufficient information for the clients to ensure eventual consistency. This is done by always appending a value for a key rather than replace the old value, and maintaining a version for each key.

When writing value to a key, it is ensured that at least “w” replicas for a key have the value appended to them. Similarly, a client will receive values from at least “r” replicas for a given key when it performs a read. By ensuring that r+w>N, sloppy quorum ensures that at least one copy of every write that is perform will be present in one or more of the “r” replicas returned. i.e., there is guaranteed to be an intersection between every write performed and at least one of the read replicas that are returned. This way, the client can review the values from the different replicas and decide which value is the correct (eventually consistent) value.

4. Briefly compare and contrast the role of accumulators and broadcast variables in uni-directional communication between the driver code and the distributed execution of a Spark application.

Answer: Accumulators allows aggregate values to be communicated from the Spark executors to the driver. They let the driver code to receive certain accumulated values as a result of performing an action, without having to fetch the result of the action to the driver code. This way, the potentially large output from the action can be directly saved to disk while still returning some useful debug or logging information to be sent to the driver code.

Broadcast variables allow the driver to communicate a large, read-only value to the executors. Broadcast variables can be reused across multiple transformations and actions. This is much more efficient than using a local variable to store such large values since these variables will be serialized and sent from the driver to the executors each time they are used.

5. CAP theorem states that only two of the three properties of Consistency (A), Availability (A) and Tolerance to Network Partitioning (P) are possible. Why do modern Big Data application prefer A+P and sacrifice C?

Answer: Contemporary big data applications run within Cloud data centers with a large number of servers. As a result the chances of machine failures or the network between the servers being disconnected (partitioned) is high. At the same time, the end users of these applications expect high availability, such as ecommerce users performing addition of items to a cart, and any lack of availability of the application can cause a revenue loss to the business. So, we need to achieve high

availability while at the same time also being tolerant to network partitioning, and hence have to sacrifice strong consistency. Such platforms do strive to achieve weak consistency such as eventual consistency. So there may be temporary inconsistency that should be acceptable, e.g., the number of available items of a product available on an ecommerce website.

6. Spark Streaming only allows time-based windows and not count-based windows. What are the downsides of such an approach?

Answer: Time based window means that tuples that have arrived within a certain window length, e.g., 3 minutes, will be formed into a window for aggregation. Count based window allows a certain count of events that arrive to be formed into a window, e.g., 25 events. Count based windows are useful when there is a need to form windows of a fixed event size by an application, and also allows the platform to load balance the events per window. If we only support time based windows and the input event rate is not constant, then it will not be able to form windows of a fixed event size using time-based window.

7. Say we have an event stream with the following pairs of (time,value) events in the given order: (1,5), (2,2), (4,6), (5,2), (7,8), (8,3), (9,4), (10,7).

What is the output from performing a sum over the values for the following window operations, only considering whole window lengths.

(a) SlidingTimeWindow(Slide=2, Length=4) [3 points]

(b) BatchCountWindow(Length=3) [2 points]

Answer: (a) (5+2+6), (6+2), (2+8+3), (8+3+4+7)

(b) (5+2+6), (2+8+3)

=== Descriptive ===

1. Discuss the pros and cons of Narrow and Wide Transformations in Spark RDDs on application expressivity and on execution performance.

Answer:

- Narrow transformations operate on each item or partition of the input RDD at a time, and each item in the output RDD has exclusively dependency on a subset of items in the input RDD.
    - From an expressivity perspective, this prevents multiple output items from accessing the same input item for performing operations or aggregation. Here, with 1:1 narrow dependency, each output item depends on exactly 1 input item (e.g., map, flatMap), while others with N:1 narrow dependency have multiple input items (e.g., mapPartition) that result in a single output item. Most narrow transformations only operate on one RDD at a time, with exception being operations like union, cogroup.
    - However, from a performance perspective, narrow transformations have several benefits. The task-level data parallelism can be at the granularity of each output item. These tasks can progress at a different pace as they are independent, avoiding the effect of stragglers. Since the output items have exclusive dependency on one or

more input items, they can exploit pipeline parallelism. Also, the contiguous tasks with narrow dependencies can all be run on the same executor. All these tasks with narrow dependencies can be part of a single stage in the dataflow, reducing the cost of scheduling. There is no data transfer across the network that is required, or costly disk I/O effects of shuffle.

- In wide transformations, multiple output items can be dependent on overlapping sets of input items. Wide transformations require a shuffle stage, where input items have to be duplicated and/moved to one or more executors handling the processing of the output item.
    - From an expressivity perspective, wide transformations allow an all-to-all dependency between the input and output items. So this allows many different forms of grouping and aggregations to be performed, both for transformations and actions. E.g., groupByKey, reduce, distinct, sort, etc. They also allow aggregation across multiple RDD, such as joins and intersections.
    - Since wide dependencies force a shuffle to happen, there are costly from a performance perspective. Shuffle causes all input data to be written to disk and transferred over the network, limiting the benefit of "in-memory" processing by Spark. They force new stages to be created, adding overhead to the scheduling. Tasks in the next stage cannot start before task in the current stage complete, causing stragglers. Pipelining cannot be exploited.

2. Discuss how Spark is able to exploit Data Parallelism, Pipeline Parallelism and Task Parallelism for scalable and distributed execution of applications.

Answer:

- Data parallelism refers to the ability to perform the same operation on different parts of the data by multiple threads/processes. By having multiple partitions per RDD, Spark allows tasks to be created for each partition to perform transformation/action on them. Each task can be executed on different threads of execution to exploit partition-level data parallelism.
- Pipeline parallelism means that if a series of (narrow) transformations are chained together, then the output items from an upstream task can be consumed as the input event by the downstream task, even before the upstream task completes executing on the full partition. These different tasks can be executed by different threads. i.e., the items from a partition can be streamed to downstream task as and when they complete execution, and we can have multiple threads with different tasks executing on different items of the same partition.
- Task parallelism is the ability to execute different tasks that do not have any dependency between them in different threads/processes. Since Spark creates a dataflow graph based on the RDD dependencies, it can identify transformations on different RDDs that are independent of each other and are part of different stages. Tasks in such independent stages can be executed concurrently by different threads/processes.

3. Discuss and contrast the role of publish-subscribe systems, distributed stream processing systems, and Complex Event Processing engines for fast data processing.

Answer:

- Pub-sub systems are responsible for routing of event streams between different publishers of the event and different consumers of the event. The publishers and consumers do not need to know of each other, and the event exchange is done using shared topics. A broker is used as a central entity to coordinate the movement of event streams. There will be two hops of data transfer, from publisher to broker and from broker to subscriber. Publishers, broker(s) and subscribers can be on different distributed machines. Produces can operate in parallel,

and similarly consumers can operate in parallel. Typically, the broker does not perform any other operation on the event other than routing it, and the event may remain opaque other than optionally exposing a key.

- Distributed stream processing systems are used for processing event streams using user-defined logic. Tasks are composed as a dataflow graph of dependencies. The use task logic executes once for each input event to generate 0 or more output events. Output events from upstream tasks and passed as input to downstream tasks for execution. Event streams are routed point-to-point between specific tasks. Applications can exploit task, data and pipeline parallelism, and execute on different machines in a distributed manner. The tasks are typically stateless. There may also be some basic windowing operations possible over the streams for aggregation.
- Complex event processing (CEP) engines are used to perform standing queries on event streams. It can be used to identify temporal patterns within event streams. Events as modelled as tuples with fields/columns exposed, and queries can access these fields within a tuple for operations like filtering, transformation, aggregation, etc. Queries are typically limited to pre-defined operators and do not allow user-defined logic. Queries can be chained together to compose more complex queries. Count/time windows, window-based aggregation and sequence operators are especially useful and unique built-in operators that are provided. These engines typically run on a single machine.

=== Programming (20 points) ===

Give the transformations and actions using Spark RDDs to perform the following tasks.

1. Given an input Key-Value RDD consisting of URL string as key and English text (string) as value, return an inverted index key-value RDD from each word in the text as key to the list of URLs as the value. [4 points]

Answer:

```
KeyValueRDD urlTextRDD;       // Input RDD with URL string as key and text string as value

urlKeywordRDD = urlTextRDD.flatMapValues(v -> v.split(“ “));

keywordUrlRDD = urlKeywordRDD.map((k,v) -> (v,k));

invertedIndexRDD = keywordUrlRDD.groupByKey();

invertedIndexRDD.collect();
```

2. Given an inverted Index key-value RDD from keyword (key) to list of URLs (value), a PageRank key-value RDD from URL (key) to PageRank of that URL (float value) and a query string variable with one or more keywords, return the URLs having the top-10 largest Page Rank values containing all the keywords in the given query string, and with the URLs sorted in descending order by PageRank value. [8 points]

Answer:

```
KeyValueRDD keywordUrlsRDD; // Input RDD with keyword string as key and string[] URL list as value
```

```
KeyValueRDD urlPagerankRDD; // Input RDD with URL string as key and
Pagerank float as value

String query; // Input query string with list of keywords to search for

queryKeys = query.split(“ “); // get list of query keywords

matchKeyURLRDD = keywordUrlsRDD.filter(k.equals(queryKeys[0])); // get RDD
with keyworks and URLs matching keyword 1

matchURLRDD = matchKeyURLRDD.flatMap((k,v) ->
Arrays.asList(v).iterator()); // get RDD with only URLs matching keyword 1

for(int i=1; i < queryKeys.length; i++) {

      // get RDD with only URL matching keyword i

      matchOtherURLRDD = keywordUrlsRDD.filter(k.equals(queryKeys[i]))

            .flatMap((k,v) -> Arrays.asList(v).iterator());

      // get RDD with URLs that match all keywords from 0 to i

      matchURLRDD = matchURLRDD.intersection(matchOtherURLRDD);

}

// Join matching URLs with their page ranks

matchURLNulRDD = matchURLRDD.mapToPair( k -> (k, 0));

matchURLPagerankRDD = matchURLNulRDD.join(urlPagerankRDD).map((k,v) -> (k,
v[1]);

matchPagerankURLRDD = matchURLPagerankRDD.map((k,v) -> (v,k));

// Sort and return URLs with top 10 Pagerank values

top10PagerankURL = matchPagerankURLRDD.sortByKey().top(10);
```

3. Given a key-value RDD from URL (key) to PageRank (float value), and a URL title RDD from URL (key) to webpage title (string value), return an output containing the inner join of the two RDDs by URL and in sorted order of PageRank. [4 points]

Answer:

```
KeyValueRDD urlPagerankRDD; // Input RDD with matching URL as key and
their Pagerank as value

KeyValueRDD urlTitleRDD; // Input RDD with URL as key and their Title as
value

urlRankTitleRDD = urlPagerankRDD.join(urlTitleRDD); // join the RDDs

rankUrlTitleRDD = urlRankTitleRDD.map( (k,v) -> (v[0], (k, v[1]) )  //
Rearrange to have Pagerank as key

rankUrlTitleRDD.sort().collect(); // return sorted rank, title and URL
```

4. Describe how you will implement reduceByKey transformation using a combineByKey transformation. [4 points]

Answer:

Assuming the value if a number,

```
rdd.reduceByKey(v1,v2 : funcAgg(v1,v2))
```

can be implemented using

```
rdd.combineByKey( k : (k, 0),       // init first value to 0
                  v1,v2 : funcAgg(v1,v2),       // merge vals
                  v1,v2 : funcAgg(v1,v2) )      // merge combiners
```