Group 10 Lab3 report

**Implementation**

1. **Filter**

* First we filter the stream to get the allNames column. Then use flatMapValues to map every topic to a key.
* val topic:KStream[String, String] =
* records.map((date,str)=> (date,str.split("\t")))
* .filter((date,col)=>col.size>23 && col(23)!="") //get allNames column
* .map((date,str)=>(date, str(23).split(";")
* .map(s=>s.split(",")(0)).distinct //get the topic
* .mkString(",")))
* .flatMapValues(topic => topic.toLowerCase.split(",")) //map topic to key
* Then we build a *statestore* using *StoreBuilder* for (topic,topic\_count).
* //Build a persistent state store
* val keyValueStoreBuilder: StoreBuilder[KeyValueStore[String,Long]] =
* Stores.keyValueStoreBuilder(Stores.persistentKeyValueStore("topicstate"),
* Serdes.String,
* Serdes.Long);
* // register store
* builder.addStateStore(keyValueStoreBuilder);
* Finally we call the transform function using the input stream. The transform function takes HistogramTransformer object as a parameter which implements TransformerSupplier interface and returns a Transformer object. The output stream obtained is then published to (“gdelt-histogram”) .
* //Transform each record of the input stream
* val outputStream:KStream[String, Long] =
* topic.transform(new HistogramTransformer(),"topicstate");
* //publish Kafka topic
* outputStream.to("gdelt-histogram")
* The get() method in HistogramTranfsormer class return Transformer object. The transformer consist of three methods: transform(), TopicCount\_increment() and TopicCount\_decrement(). The transform() method calls TopicCount\_increment() first to increment the count of topic and update the statestore. Then the TopicCount\_decrement() is scheduled to execute after 60 minutes only once. The incoming record is scheduled for a task to delete itself after an hour using wall clock time.
* // Should return the current count of the name during the \_last\_ hour
* def transform(key: String, name: String): KeyValue[String, Long] = {
* val count = TopicCount\_increment(name) // Increment the topic count
* //schedule a cron\_job to decrement topic count after 1 hour
* var cron\_job: Cancellable = null
* cron\_job = this.context.schedule(1000 \* 60 \* 60, PunctuationType.WALL\_CLOCK\_TIME, (timestamp) => {
* TopicCount\_decrement(name) // Decrement the topic count after 1 hour
* cron\_job.cancel() // decremenmt once
* });
* return new KeyValue(name,count)
* }
* The TopicCount\_increment() method increments the count of a topic if the topic is already present else initiates count to 1 for a new topic. Similarly, Topic\_decrement() method decrements the count of topic after an hour. It deletes a topic if the count of the topic becomes zero and then updates the consumer using context.forward().

**Questions**

**1. In typical use, what kind of operation would be more expensive, a narrow**

**dependency or a wide dependency? Why?**

* Operations involving wide dependencies are more expensive.
* This is because there exists a one-to-many relationship between the partitions in parent RDD to the partitions in child RDD when considering wide dependency.
* Therefore, transformations involving wide dependencies are slower as they require the data to shuffled over the network.

**2. What is the shuffle operation and why is it such an important topic in**

**Spark optimization?**

* Data is distributed over the network. This data has to be moved from one node to another whenever a “group” operation (or mostly operations with wide dependency) is performed.

This process is called shuffling.

* For example, groupByKey() method involves all the distributed key-value pairs with the same key to be grouped together on a same machine. This causes shuffling.
* If we try to prevent shuffling from happening until it’s totally necessary, we can significantly reduce the running time of our program. Hence, shuffling is an important topic in spark optimization.

**3. In what way can Dataframes and Datasets improve performance both in**

**compute, but also in the distributing of data compared to RDDs? Under**

**what conditions will Dataframes perform better than RDDs?**

* In Dataframes and Datasets, optimizations are done using catalyst optimizer, which provides both rule based as well as cost based optimizations. This increases the performance of queries that developers write. On the contrary, no such optimizations are done with a RDD implementation.
* When it comes to distribution of data, spark dataframe uses off heap memory for serialization. byte code is generated dynamically and there is no need for deserialization when considering small operations. On the other hand, while distributing data into the network, RDD uses Java serialization. This serialization of Java and scala objects incurs an overhead as both structure and data is shuffled between the nodes.
* Dataframes will perform better than RDDs when we are dealing with structured data and when our processing requires filters, maps, aggregations, attribute access on semi-structured data.

**4. Why does Kafka by default not guarantee exactly once delivery semantics on producers?**

* Kafka by default does not guarantee exactly once delivery semantics on producers because exactly-once processing requires a cluster of at least three brokers by default which is also recommended setting for production. User can simply change the “processor.guarantee” config to “exactly\_once” to change the delivery semantic.

**5. Kafka is a binary protocol (with a reference implementation in Java), whereas Spark is a framework. Name two (of the many) advantages of Kafka being a binary protocol in the context of Big Data. (max. 100 words)**

* Two advantages of Kafka being a binary protocol:
  + Event-at-a-time processing: The messages/events are processed as they arrive unlike spark needs to micro-batch the messages.
  + Greater parallelism: Kafka client can connect through TCP to multiple brokers and transfer data in parallel across multiple partitions of the same topic.

**Questions Specific to the assignment**

1. **On average, how many bytes per second does the stream transformer have to consume? How many does it produce?**



|  |  |  |
| --- | --- | --- |
| Segments | RDD Implementation (in sec) | DF/DS Implementation (in sec) |
| 10 | 13.110 | 11.896 |
| 20 | 21.620 | 17.481 |
| 50 | 37.872 | 21.938 |
| 100 | 97.659 | 101.010 |
| 150 | 143.762 | 138.117 |

As we can see from the observations above, the RDD implementation and Dataframe Implementation don’t have much difference in terms of execution time for small number of segments. We expect DF implementation to be faster since it is compiled into execution plan and then executed. Spark can optimize the execution plan to reduce execution time.

1. **Could you use a Java/Scala data structure instead of a Kafka StateStore to manage your state in a processor/transformer ? Why,or why not? (max. 50words)**

* No, Java/ Scala data structure cannot be used since the application is stateful. For the processor/transformer to work correctly the state must be managed in a fault tolerant manner, which is possible through Kafka state store. The state store is frequently used by transformer to store and query data.

**Given that the histogram is stored in a Kafka State Store, how would you extract the top 100 topics? Is this eﬀicient? (max. 75words)**

* We can extract the top 100 topics from Kafka state store by implementing a Kafka consumer which would subscribe the histogram topic (considering it is stored and published). Also, a stream application can be implemented to query the data stored in Kafka state store. Considering, histogram is consumer too, the most efficient way to get the top 100 topics would be to subscribe the same topic as histogram and filter the topic to get the result.

**The visualizer draws the histogram in your web browser. A Kafka consumer has to communicate the current ‘state’ of the histogram to this visualizer. What do you think is an eﬀicient way of streaming the ‘state’ of the histogram to the webserver? (max. 75words)**

* An efficient way of streaming the state of the histogram to the webserver could be Apache Arrow. Apache Arrow would prevent serialization and de-serialization, therefore improving the efficiency. Moreover, it would facilitate communication between other components without worrying about data conversion.

**What are the two ways you can scale your Kafka implementation over multiple nodes? (max. 100words)**

* The two ways to scale Kafka implementation over multiple nodes are:
  + **Partitioning**: A topic can be streamed to multiple partitions. A producer can deliver topic a particular partition of that topic.
  + **Consumer groups**: Partitions can be divided among consumers if they subscribe the same topic and have same number/name.
* **How could you use Kafka’s partitioning to compute the histogram in parallel? (max. 100words)**
* The number of partitions determines the maximum consumer (group) parallelism. The histogram consumer can be organized into consumer group for the given topic such that each consumer within the group reds from a unique partition and the group as whole consumes the whole stream and construct the histogram in parallel.