Open source technologies

Monday, May 1, 2023 6:07 PM

<table>
<tr><th>Question</th><th>Answer</th></tr>
<tr><td>Spark actions</td><td>Actions return a value to the Spark application or export data Actions include count, collect, etc.
Operation types on RDDs: transformations and actions
DEF: Spark actions are operations which make some calculation and return the result to the driver or persist it in external storage
The simplest actions are count(), take() and collect()
The driver performs transformations and actions</td></tr>
<tr><td>Spark transformations</td><td>RDD Operations:
- Transformations create new RDDs from existing RDDS (map, filter, join, Lazy operation)
- Actions return a value to the Spark application or export data, Actions include count, collect, etc.
DEF: Spark transformations are operations which read RDDs as inputs and produce RDDs as outputs
Transformations do not mutate input RDDs → they just produce new output RDDs and return a pointer to it
Many, but not all transformations are element-wise i.e., they operate on the elements of the input RDDs in sequence Transformations can operate on one (e.g., filter()), two (e.g., union()) or more input RDDs.</td></tr>
<tr><td>What's common between lineage graphs and lazy evaluations. What are they?</td><td>DEF: The Spark lineage graph is the set of dependencies between RDDs
• The lineage graph is used to re-compute RDDs on demand and to recover lost data if parts of a persisted RDD are lost
Transformations on RDDs are lazily evaluated → Spark will not begin to execute transformations until it sees an action
▪ Instead of immediate execution, Spark does the following:
• Internally record metadata about transformation requests → this in essence means that in-memory RDDs can be regarded as instructions for computing data instead of data itself (which is not materialized immediately)
• Lazy data load, i.e. actual data read and parallelize will be executed when needed to perform an action downstream
▪ Lazy evaluation allows Spark to optimize data processing pipelines inline, transparently to the user, thereby reducing the number of passes over the data</td></tr>
<tr><td>Similarity between map phase and spark</td><td>• Distributed Processing:
designed for distributed processing of large-scale data across a cluster of machines.
• Transformations: In both MapReduce and Spark, the map phase involves applying a transformation to each input record or element independently. This transformation is specified by a user-defined function (mapper) and is applied in a parallel and distributed manner across the dataset partitions.
• Fault Tolerance: Both MapReduce and Spark handle fault tolerance during the map phase. If a node fails during the execution of the map phase, both frameworks can reassign the failed task to another node and continue processing from the intermediate results generated by the completed tasks.</td></tr>
<tr><td>Hadoop</td><td>There are two main elements to the Hadoop framework, namely distributed storage and processing. The distributed storage uses the Hadoop Distributed File System (HDFS) while the processing implements the MapReduce programming model using Yet Another Resource Negotiator (YARN) to schedule tasks and allocate resources.</td></tr>
<tr><td>Kafka vs spark</td><td><table>
<tr><td>Purpose and Functionality:
a distributed streaming platform designed for high-throughput, fault-tolerant, and real-time data streaming. It provides publish-subscribe messaging, durable storage, and stream processing capabilities.</td><td>Apache Spark is a general-purpose distributed computing framework that provides an in-memory data processing engine. It offers high-speed data processing, advanced analytics, and machine learning capabilities.</td></tr>
<tr><td>Data Streaming vs. Data Processing
handling large-scale, real-time data streaming. It enables producers to publish data streams, and consumers can subscribe to those streams, process the data, and store it for further analysis</td><td>Spark is primarily focused on distributed data processing and analytics. It provides a unified platform for batch processing, interactive queries, machine learning, and real-time streaming processing. Spark can consume data from Kafka</td></tr>
</table></td></tr>
</table>

	or downstream processing.	and process it in real-time or perform batch processing on stored data.
	Data Persistence: Kafka: Kafka acts as a highly scalable and fault-tolerant distributed storage system. It stores data streams in topics, retaining the data for a configurable period or based on storage limits.	Spark does not provide built-in data persistence. It relies on external storage systems like Hadoop Distributed File System (HDFS), Apache Cassandra, or cloud storage services to store and access data.
	Stream Processing vs. Batch Processing: Kafka: Kafka focuses on real-time stream processing, enabling continuous ingestion and processing of data as it arrives. It provides support for stream processing frameworks like Apache Flink and Apache Samza.	Spark: Spark supports both real-time stream processing and batch processing. It provides libraries like Spark Streaming and Structured Streaming for real-time processing, and the core Spark engine for batch processing and advanced analytics.
	Integration: Kafka: Kafka integrates well with various data processing frameworks, including Spark, Hadoop, and other stream processing systems. It acts as a reliable and scalable data source or sink for these systems.	Spark: Spark integrates with various data sources and systems, including Kafka. It can consume data from Kafka topics directly into Spark Streaming or Structured Streaming for real-time processing or use Kafka as a source in batch processing workflows.
	Fault Tolerance and Scalability: Kafka: Kafka is designed for high fault tolerance, with built-in replication and distributed architecture. It provides fault tolerance by maintaining multiple copies of data across a cluster of brokers. Kafka scales horizontally by adding more brokers to handle higher data ingestion and consumption rates.	Spark: Spark provides fault tolerance by storing intermediate data in-memory or on disk. It maintains lineage information to reconstruct lost data partitions. Spark can scale horizontally by adding more worker nodes to distribute the processing workload.
Main common thing between spark transformations and mongoDB		
-Difference and impact between lambda and spark streaming	Architecture: Lambda: Lambda architecture is a data processing architecture that combines batch processing and real-time stream processing. It involves separate processing layers for batch and real-time data. Batch processing is typically performed using technologies like Apache Hadoop and MapReduce, while real-time processing is handled by stream processing engines like Apache Storm or Apache Flink. Lambda architecture aims to provide both real-time and accurate results as well as handle large-scale data sets.	Spark Streaming: Spark Streaming is a component of Apache Spark that enables scalable, fault-tolerant, and near-real-time stream processing. It processes data streams in small batches (micro-batches) rather than processing each event individually. Spark Streaming integrates with the core Spark engine, allowing seamless integration with batch processing, machine learning, and other Spark functionalities.
	Data Processing Model: Lambda: In the Lambda architecture, both batch and real-time processing occur independently and produce separate results. Batch processing handles large volumes of data and performs complex transformations, aggregations, and analytics on the entire data set. Real-time processing focuses on low-latency processing of incoming data streams to provide real-time insights and immediate responses.	Spark Streaming: Spark Streaming processes data in mini-batches, which are small, fixed-time intervals of data. It treats real-time data as a series of RDDs (Resilient Distributed Datasets) and applies batch processing operations on these RDDs. This approach allows Spark Streaming to leverage the same programming model and APIs as batch processing, simplifying the development process.

	Processing Latency: Lambda: Lambda architecture introduces inherent processing latency due to the separate batch and real-time processing layers. While batch processing can handle large volumes of data efficiently, the real-time layer introduces some latency before delivering immediate results.	Spark Streaming: Spark Streaming aims to provide low-latency processing by leveraging mini-batches. Although the latency is not as low as pure stream processing engines like Apache Storm or Apache Flink, it offers a good trade-off between processing time and ease of development by utilizing the existing Spark infrastructure.
	Complexity: Lambda: Implementing and maintaining a Lambda architecture can be complex due to the need for managing multiple processing layers, handling data consistency between batch and real-time views, and ensuring proper synchronization of results from different layers.	Spark Streaming: Spark Streaming simplifies the development process by using a unified programming model. Developers can use the same set of APIs for both batch and real-time processing, making it easier to write and maintain code.
	Ecosystem and Integration: Lambda: Lambda architecture is a conceptual design pattern, and the choice of technologies for batch and real-time processing can vary. It can integrate with different components like Apache Hadoop, Apache Storm, or Apache Flink to implement the batch and stream processing layers.	Spark Streaming: Spark Streaming is a part of the Apache Spark ecosystem and tightly integrated with other Spark components. It can seamlessly interact with Spark's batch processing, SQL, machine learning, and graph processing libraries, providing a unified data processing platform.

- Shuffle phase vs rdd	▪ Process: The map process transforms (K1, V1) and generates intermediate results on the local disk ▪ Process: The shuffle process sorts, copies and merges the intermediate outputs on the reduce compute nodes ▪ Process: The reduce process transforms the outputs of the shuffle phase
- MapReduce(general)	
MongoDB	document-oriented distributed NoSQL database 1. Scalability and Performance: MongoDB is built to scale horizontally across multiple servers and handles large amounts of data with ease. It supports automatic sharding, allowing data to be distributed across a cluster of machines. This capability enables high throughput and read/write operations that can scale to meet growing demands. 2. High Availability and Fault Tolerance: MongoDB provides built-in replication and automatic failover. By maintaining multiple copies of data across a cluster, it ensures high availability and data durability. In the event of a node failure, MongoDB can automatically promote a replica to a primary role, minimizing downtime and ensuring continuous operation. 3. Rich Query Language: MongoDB supports a powerful query language with a wide range of query capabilities. It provides support for ad-hoc queries, indexing, aggregation, geospatial queries, and full-text search. The flexible document model allows for complex nested queries and rich data retrieval options.
What is MongoDB for?	Real-Time Analytics Document Storage and Retrieval Easy interface with common languages (Java, Javascript, PHP, etc.) DB tech should run anywhere (VM's, cloud, etc.
What type of data MongoDB use?	Document based
Flink	Apache Flink is a unified stream and batch processing framework Flink programs consist of streams and transformations ▪ The Flink runtime supports iterative algorithms → ML ▪ Fault tolerance via distributed checkpoints

<table>
<tr><td>storm</td><td></td></tr>
<tr><td>Difference between Flink and storm (main thing he wanted to listen - guarantee)</td><td>
<table>
<tr><td>Processing Model:
Flink is a stream processing framework that supports both stream processing and batch processing. It provides a unified API and execution engine for processing both bounded (batch) and unbounded (streaming) data. Flink's processing model is based on the concept of streams, where data is processed as continuous streams, allowing for low-latency and high-throughput processing</td><td>• Storm: Storm is a real-time stream processing system that focuses solely on processing unbounded data streams. It processes data in real-time and supports continuous data ingestion and processing. Storm's processing model is based on the concept of topologies, which consist of spouts (data sources) and bolts (data processing operations).</td></tr>
<tr><td>Fault Tolerance:
• Flink: Flink provides strong fault-tolerance guarantees through its checkpointing mechanism. It allows for exactly-once processing semantics, ensuring that each event is processed exactly once, even in the event of failures. Flink maintains consistent state snapshots to recover and resume processing in case of failures.</td><td>• Storm: Storm provides configurable fault-tolerance mechanisms, but it does not guarantee exactly-once processing semantics out-of-the-box. It provides at-least-once processing semantics by default and relies on external systems, such as Apache Kafka or Apache HBase, for message durability and state storage. Achieving exactly-once semantics requires additional coordination and customization.</td></tr>
<tr><td>Processing Guarantees:
• Flink supports three processing guarantees: at-least-once, at-most-once, and exactly-once. It allows developers to choose the desired level of processing guarantee based on their application requirements.</td><td>• Storm: Storm primarily provides at-least-once processing semantics by default, and achieving exactly-once semantics requires custom coordination and integration with external systems.</td></tr>
<tr><td>Event Time Processing:
• Flink: Flink has built-in support for event time processing, allowing the processing of data based on the time at which events occurred rather than when they are processed. Flink provides mechanisms for handling event time out-of-order events, watermarks, and windowing operations based on event time.</td><td>• Storm: Storm does not have built-in support for event time processing. It primarily operates based on processing time, which is the time at which the data is received and processed.</td></tr>
<tr><td>State Management:
• Flink: Flink provides built-in support for managing and maintaining state as part of its stream processing capabilities. It offers flexible state backend options, including in-memory state, local disk, and external systems like Apache Hadoop or Apache RocksDB.</td><td>• Storm: Storm relies on external systems, such as Apache HBase or Apache Cassandra, for state storage and management. It does not provide built-in state management capabilities.</td></tr>
</table>
</td></tr>
<tr><td>Storm transformations</td><td>Apache Storm is a real time, distributed stream processing platform • Designed as a directed acyclic graph (DAG) data transformation pipeline • Note: real-time is a key characteristic • Note: Storm was the 1st real streaming system (!)

Data source can be: streams = unbounded
Spouts: multiple streams
Bolts= Bolts process input streams and produce new streams: Can implement functions such as filters,</td></tr>
</table>

	aggregation, join, etc Topology: Network of spouts and bolts
cassandra - which database to use for multiple data centres data storage - cassandra or hdfs?	
-- flink guarantee ? Exactly once. then he asked what is exactly once	Exactly-once guarantee in Apache Flink refers to the assurance that every event or record in a stream processing application will be processed and produced exactly once, without any duplicates or omissions, even in the presence of failures or system restarts. This guarantee ensures that the results of the computation are consistent and accurate. Flink achieves the exactly-once guarantee through a combination of mechanisms: 1. Checkpoints: Flink creates consistent checkpoints of the application's state at regular intervals. Checkpoints capture the state of all operators in the application, including the input streams, intermediate state, and output streams. These checkpoints are stored durably, typically in a distributed file system or a highly available storage system. Applications relying on "exactly once" semantics have the advantage of avoiding duplicates and ensuring correctness without the need for additional handling of duplicates or data inconsistencies. In the context of distributed databases, "at least once," "at most once," and "exactly once" refer to different levels of message delivery guarantees in the presence of failures. These terms describe how many times a message or operation is processed and applied to the database. Here's an overview of each guarantee: 2. At Least Once: • At least once guarantee ensures that every message or operation is processed and applied to the database at least once, even in the presence of failures or system restarts. • It allows for the possibility of duplicate messages or operations being processed and applied. • To achieve this guarantee, the system typically relies on techniques such as message acknowledgment, retries, and idempotency. • Applications built on an "at least once" guarantee need to handle potential duplicates and ensure idempotent operations to prevent data inconsistencies. 3. At Most Once: • At most once guarantee ensures that each message or operation is processed and applied to the database at most once. • It means that the system aims to avoid duplicate processing by discarding duplicate messages or operations. • However, there is a possibility of message loss in case of failures or system disruptions. • This guarantee is achieved by techniques like message deduplication or using unique identifiers to identify and discard duplicates. • Applications relying on "at most once" semantics should handle potential message loss or use mechanisms to detect and recover from missing operations. 4. Exactly Once: • Exactly once guarantee ensures that each message or operation is processed and applied exactly once to the database, without any duplicates or omissions. • It provides strong consistency and ensures that the results of the computation are accurate and consistent. • Achieving exactly once semantics is more complex and typically involves additional coordination and mechanisms such as transactional processing, checkpoints, and atomic commits. • It requires coordination between the sender and receiver, often leveraging distributed protocols like two-phase commit or distributed transaction management. • Applications relying on "exactly once" semantics have the advantage of avoiding duplicates and ensuring correctness without the need for additional handling of duplicates or data inconsistencies.
Flink, checkpoints, how we do the checkpoint, why is it needed	checkpoints are a fundamental component of Flink's fault tolerance mechanism, enabling exactly-once processing semantics and ensuring data consistency. They provide the ability to recover from failures and restart the application while maintaining the integrity and accuracy of the data processing. • Flink periodically triggers the creation of checkpoints to capture the state of the streaming

application. The frequency of checkpoints can be configured based on the desired level of fault tolerance and application requirements.

- When a checkpoint is triggered, Flink initiates a coordinated process across all operators in the application to save their state. This process includes storing the state of input streams, intermediate state, and operator-specific state.

1. Recovery and Restart:

- In case of failures or system restarts, Flink can recover the application from the latest successfully completed checkpoint.
- When a failure occurs, Flink restores the application's state from the most recent checkpoint, including the operator states and the positions in the input streams.
- After recovery, the application resumes processing from the restored state, ensuring that the data is processed consistently and accurately.

Why are Checkpoints Needed?

- Fault Tolerance: Checkpoints play a vital role in achieving fault tolerance in Flink. By capturing the state of the application, including input data, intermediate state, and operator state, checkpoints enable the recovery of the application from failures or system restarts.
- Exactly-Once Processing: Checkpoints are necessary to achieve exactly-once processing semantics. By creating consistent snapshots of the application's state, Flink ensures that events or records are processed and produced exactly once, even in the presence of failures.
- Data Consistency: Checkpoints provide data consistency by capturing a consistent view of the application's state. It guarantees that the computation is based on a known and stable set of data, which is crucial for accurate and reliable results.
- Incremental Processing: Checkpoints also enable incremental processing by allowing Flink to process only the new or changed data since the last checkpoint, rather than reprocessing the entire data set. This optimization improves processing efficiency and reduces latency.

- Docker
- Docker vs virtual machine

automate the deployment, scaling, and management of applications using containerization.
Docker enables containerization, which involves encapsulating an application and its dependencies into a standardized container image
A container image contains everything needed to run the application, including the code, runtime, libraries, and system tools.
Docker simplifies the deployment process by packaging applications into containers. With Docker, you can create a container image that includes the application and all its dependencies.
Using Docker, you can easily replicate and scale containers to handle increased workloads
Developers can create Docker images that contain the exact dependencies and configurations required for an application to run
Docker simplifies dependency management by encapsulating dependencies within containers

• Docker: Docker uses containerization to run applications. Containers share the host system's kernel, but they are isolated from each other and have their own user space. This lightweight approach allows for faster startup times and better resource utilization.	• Virtual Machine: VMs, on the other hand, simulate an entire operating system (OS) environment. Each VM runs its own OS instance and has dedicated resources, including CPU, memory, storage, and network. VMs are typically larger in size and require more resources to run.
Performance: • Docker: Docker containers have less overhead compared to VMs since they share the host OS kernel. This makes containerized applications more lightweight and efficient, resulting in faster startup times and better performance.	• Virtual Machine: VMs have more overhead due to the need to run a full OS on each virtual instance. This can lead to slower startup times and slightly reduced performance compared to Docker containers.

Tell about Docker, what does it do, what's container, does it replace OS?

fault tolerance in kafka, replica	
-Logstash(input types)	
- Hbase, hbase vs hdfs	Hbase column-oriented db A table has multiple column families and each column family can have any nb of columns
HDFS, parts in architecture	
- can we use spark for High speed computing: facebook uses cloud or HSC	
- what is better cloud services or physical servers(answer depends they didn't accept)	
- - high speed computing vs cloud computing	
Slurm: Is it cloud based or hpc?	
-SLURM(general)	
Partitioning Replication Cluster management	
cloud computing and flarma(something like this)	