Cloudera

CDP-3002

CDP Data Engineer- Certification Exam

Latest Version: 7.4

Question: 1

A Data Engineer identifies severe performance degradation due to high shuffle I/O caused by too many small shuffle partitions (default 200) in a Spark application running on a CDE Virtual Cluster. To optimize the job, they must set the number of shuffle partitions to 100.
Which option correctly demonstrates how this performance configuration should be applied to the job?

A. Update the Virtual Cluster configuration using cde-utils.sh add-spark-config-in-virtual-cluster -c 'spark.sql.shuffle.partitions=100', then rerun the existing job definition.
B. Set spark.app.id=100 in the application code, ensuring the driver manages partition count implicitly.
C. Execute the job using cde job run —name my_job —conf spark.driver.cores=100, overriding the setting specifically for this run.
D. Modify the Spark driver configuration to include —conf spark.default.parallelism=100 during job creation.
E. Dynamically adjust the cluster's memory fraction by setting spark.memory.storageFraction=0.9 during job submission to force fewer partitions.

Answer: A

Explanation:
To resolve performance issues caused by an inefficient number of shuffle partitions, the property 'spark.sql.shuffe.partitions' must be tuned. In CDE, configuring this value at the Virtual Cluster level using 'cde-utils.sh add-spark-config-in-virtual-cluster' applies the configuration to all jobs running on that cluster. This is suitable if all jobs require this tuning. Alternatively, it can be applied per job using the '—conf flag with 'cde spark submit' or 'cde job run'. Option A provides a mechanism for tuning cluster operations via the administrative utilities. Option B is incorrect as 'spark.app.id' is not supported in CDE job configurations. Option C uses the wrong configuration key (cores instead of shuffle partitions). Option D uses the wrong tuning parameter for shuffle output partitions. Option E relates to memory storage capacity, not shuffle parallelism.

Question: 2

A data engineer is analyzing the execution flow of a complex Spark DataFrame join operation running on a Kubernetes-based cluster After the code is submitted, Spark progresses through Logical and Physical Planning. At which core stage does Spark use a cost model to compare generated execution paths (e.g., deciding between a Broadcast Hash Join and a Sort Merge Join) and select the single optimal plan to distribute as a set of tasks across the Executors?

A. The Unresolved Logical Plan validation against the Catalog
B. The Catalyst Optimizer performing rule-based optimizations, resulting in the Optimized Logical Plan
C. The final Physical Planning session, selecting the best Physical Plan after running against a cost model
D. The Driver converting the Physical Plan into a Directed Acyclic Graph (DAG) for stage definition

E. The Tungsten engine generating optimized Java bytecode for CPU-bound operations

Answer: C

Explanation:
Spark's planning for structured APIs involves two main phases: logical and physical. The initial logical plan is optimized by the Catalyst Optimizer, resulting in the optimized logical plan. The physical planning phase takes this optimized logical plan and generates multiple possible physical execution plans based on cluster details and configuration (like shuffle partitioning or data distribution). These physical plans are then run against a cost model, and Spark selects the best physical plan (the blueprint for actual execution) to be sent to the cluster. Choosing a specific join strategy (like Broadcast Hash Join) is part of this physical planning decision.

Question: 3

A PySpark job requires the installation of the pandas library to run advanced transformations inside the executor pods on a Cloudera Data Engineering Virtual Clusten The Data Engineer prepares the Python dependency using the recommended mechanism. Given the Kubernetes environment in CDE, which configuration accurately sets up the required virtual environment dependency?

A. Creating a files resource containing pandas.zip, and referencing it via —files flag.
B. Defining spark.executor.pyspark.archive=hdfs://path/to/pandas.tar.gz in the job configuration.
C. Creating a python-env resource by uploading a requirements.txt file specifying pandas, and mounting the resource to the job definition.
D. Building a custom Spark JAR with an embedded Pandas library, and submitting it as a primary job file.
E. Setting the environment variable PYTHONPATH=/usr/lib/pandas in the Kubernetes pod specifications manually.

Answer: C

Explanation:
In Cloudera Data Engineering (CDE), custom Python dependencies for Spark jobs (like 'pyspark' features needing Pandas) are managed using the dedicated 'python-env' resource type. This resource mandates the upload of a 'requirements.txt' file, which CDE uses to build the necessary Python virtual environment for the job execution on the pods. Option A uses the generic 'files' resource, which is intended for application code or supporting files, not managed Python environments. Options B, D, and E describe deployment methods that are not standard or supported procedures for managing Python environments in CDE.

Question: 4

A Data Engineer develops a complex ETL pipeline on a Spark DataFrame, df_transformed, involving multiple wide transformations (joins and aggregations). This single DataFrame is subsequently used three separate times for downstream analytical tasks (e.g., writing to Iceberg tables, running two different reports). To achieve optimal performance tuning and prevent redundant computation of the

complex lineage across the cluster, which Spark operation should be applied after creating df_transformed?

A. Execute df_transformed.collect() to force immediate data transfer to the driver node.
B. Apply df_transformed.cache() to utilize distributed data persistence.
C. Execute df_transformed.repartition(400) to increase downstream parallelism.
D. Use spark.conf.set('spark.lazy.execution', 'false') to bypass lazy execution.
E. Convert the DataFrame back to an RDD using df_transformed.rdd().

Answer: B

Explanation:
Persistence (caching) is a fundamental optimization technique crucial when a DataFrame/Dataset resulting from expensive operations is reused multiple times. Calling 'df_transformed.cache()' (or 'persist()') tells Spark to store the computed data partitions in memory (and potentially disk), avoiding the costly re-execution of the preceding transformations (the lineage) for subsequent reads. This significantly improves job acceleration. Option A forces computation but moves all data to the single driver, potentially causing OOM errors and is antithetical to distributed processing. Option C relates to shuffling/parallelism management but doesn't prevent re-computation. Option D describes a non-existent/incorrect Spark configuration (Spark execution is inherently lazy until an action is called).

Question: 5

A Data Engineer is planning the migration of legacy spark-submit batch jobs to production workflows running via the Cloudera Data Engineering CLI (Spark over Kubernetes). Which TWO statements accurately reflect best practices or operational characteristics for deploying and managing Spark applications in this CDE environment?

A. The deployment mode for Spark jobs in CDE must be manually specified as —deploy-mode=client, ensuring the driver runs on the submitting machine.
B. The CDE CLI command cde job run is preferred over cde spark submit for production execution because it requires the job definition to be permanently defined beforehand, simplifying repeated execution and scheduling.
C. Custom containerized execution environments are exclusively supported via the custom-runtime-image resource type, allowing Docker images to run as both Driver and Executor pods.
D. When setting resource limits, CDE defaults the maximum number of executors to the cluster's CPU core limit divided by the configured —executor-cores setting.
E. Job dependencies, such as application JAR files or external reference configurations, must be managed using the files resource type, ensuring they are mounted to the standard lapp/mount directory on the workload pods.

Answer: B,E

Explanation:
Statement B is correct: 'cde job run' requires a pre-created job definition and is generally suited for production environments where a job is run multiple times, contrasting with 'cde spark submit', which is

better for development testing as it doesn't create permanent definitions. Statement E is correct: The 'files' resource type is used for application code (like JARs or Python files) and external reference configuration files, which are mounted to the '/app/mount' location on the Spark driver and executor pods. Statement A is incorrect: Spark jobs in CDE run using Kubernetes, often implicitly in cluster mode, and requiring '—deploy-mode=client' is usually inappropriate for production Spark jobs. The CDE CLI abstracts the cluster management details. Statement C is partially correct but too restrictive: While 'custom-runtime-image' supports custom containers, python dependencies can also be managed via 'python-env'. Statement D is incorrect: CDE autoscaling sets the executor range based on CPU cores configured for the Virtual Cluster to maximize efficiency, but the specific calculation involving hard limits isn't stated as a guaranteed default, rather the executor range is set to match the CPU core range of the Virtual Cluster by default.

Question: 6

A Data Engineer is configuring a Spark job in Cloudera Data Engineering (CDE) intended to run on a Kubernetes Virtual Cluster. The job processes highly variable datasets, and the engineer aims to minimize costs by only consuming the necessary executor resources. Which pair of configuration flags must be explicitly defined when submitting this job using the cde spark submit command to guarantee resource elasticity within defined bounds?

A. —driver-memory and —driver-cores
B. —executor-memory and —executor-cores
C. —min-executors and —max-executors
D. —conf spark.dynamicAllocation.initialExecutors and —conf spark.dynamicAllocation.enabled=true
E. —conf spark.kubernetes.authenticate.driver.serviceAccountName and —conf spark.kubernetes.namespace

Answer: C

Explanation:
When deploying a Spark job via the CDE CLI using cde spark submit, the minimum and maximum boundaries for resource allocation are controlled by —min-executors and --max-executors. Explicitly setting these flags directly dictates the range within which the Spark application can dynamically allocate executors on the underlying Kubernetes cluster managed by CDE, ensuring the job starts with sufficient resources (minimum) and respects cluster capacity limits (maximum) for cost efficiency.

Question: 7

A data engineering team observes that Spark jobs deployed on their CDE Virtual Cluster (which manages Kubernetes resources) efficiently scale up resources when needed, but hold onto idle executors for too long, delaying scale-down and increasing cloud costs. Which CDE utility command parameter, related to cluster autoscaling, should the engineer decrease to accelerate the return of resources to the cluster pool?

A. —unremovable-node-recheck-timeout
B. —scale-down-delay-after-add

C. —scale-down-delay-after-failure
D. —scale-down-unneeded-time
E. —scale-down-delay-after-delete

Answer: D

Explanation:
To speed up the process of scaling down, the user can decrease the value of the —scale-down-unneeded-time option, which specifies the amount of time before a node qualifies for scale-down. The default value for this is 10 minutes. By lowering this value, the cluster autoscaler detects idle nodes (which host released Spark executors) more quickly and proceeds with node removal, aligning with the goal of maximizing resource utilization and reducing cost. While other options manage different scale-down delays, —scale-down-unneeded-time directly controls when an idle resource is marked for deletion.

Question: 8

A performance analysis of a dynamic Spark application on Kubernetes shows that while the application started with minimal resources, scaling requests eventually hit a ceiling imposed by the Virtual Cluster (VC) configuration. The current VC configuration has a maximum capacity set to 50 cores. The user submits a job with the following configuration settings: —executor-cores 4 —min-executors 2 --max-executors 20. If dynamic allocation attempts to scale beyond 20 executors, what is the controlling factor that limits the total resources the Spark application can consume?

A. The spark.driver.memory setting, as the driver prevents resource overallocation.
B. The —max-executors 20 flag defined explicitly during job submission.
C. The Virtual Cluster's absolute Auto-Scale Max Capacity setting of 50 cores, regardless of job configuration.
D. The total number of shuffle partitions defined by spark.sql.shuffe.partitions.
E. The limit defined by the Resource Quota set at the CDP environment level.

Answer: B

Explanation:
The maximum number of executors a specific job is allowed to request, even when utilizing dynamic allocation, is governed by the max-executors flag passed during the job submission or configured in the job definition. If this value is set to 20, the Spark application running within the CDE Virtual Cluster (K8s) will not request more than 20 executors, even if the Virtual Cluster itself has a higher overall resource ceiling (like the 50 cores mentioned, which could theoretically support 12 executors running 4 cores each, but the job limit is the immediate constraint, 20 executors 4 cores/executor = 80 cores total requested, which would still fail if VC max capacity is 50 cores). Assuming the VC capacity is sufficient for the job's theoretical max, the job-specific —max-executors is the tightest constraint imposed by the user on the application's runtime scale.

Question: 9

A Cloudera Data Engineer is designing a DAG for high-frequency incremental data ingestion into Iceberg tables, requiring a check and execution cycle every 5 minutes. The DAG execution logic relies on templating the data interval boundaries. If the DAG uses Python's datetime and timedelta modules for scheduling, what is the precise code snippet required to define this high-frequency schedule?

- `schedule_interval=timedelta(minutes=5)`
- `schedule_interval='/5 '`
- `schedule_interval=datetime.timedelta(minutes=5)`
- `schedule_interval='5 minutes'`
- `schedule_interval=timedelta(hours=0, minutes=5)`

A. Option A
B. Option B
C. Option C
D. Option D
E. Option E

Answer: C

Explanation:
Airflow supports defining the schedule interval using either standard cron expressions (as strings) or Python's `datetime.timedelta` object. If the goal is to use the Python timedelta object, the correct implementation requires importing the `datetime` module and using `datetime.timedelta(minutes=5)` or simply `timedelta(minutes=5)` if `timedelta` is imported directly. Assuming standard imports (e.g., `from datetime import timedelta`, as shown in context examples), Option C is the most formally precise code structure, though A is also often valid based on common imports. However, since Option C explicitly uses `datetime.timedelta`, which is standard, and we must assume the necessary import is available, it is the best representation of defining the interval using Python time objects. Option B uses the cron string equivalent, which is also valid, but the question specifically asks for the code snippet using `timedelta`. Option A depends on importing only `timedelta`, while Option C shows the fully qualified module usage. Given examples commonly import `datetime` and `timedelta`, Option C is the most robust presentation if the intent is to highlight the use of the Python object from the correct module, which is `datetime.timedelta`.

Question: 10

A developer needs to configure a new CDE Spark job through the CLI to run automatically once a day at midnight UTC. The developer sets the job schedule using the following parameters: `--schedule-enabled=true --cron-expression '0 0 ' --schedule-start 2024-06-01T14:00:00Z` Based on Airflow scheduling logic employed by CDE jobs, when will the very first successful job run complete its execution?

A. Immediately after creation, as 2024-06-01T14T:00:00ż has passed midnight.
At 2024-06-01T23:59:59Z, closing the daily interval of June 1st.
At 2024-06-02T00:00:00Z marking the start of the first scheduled interval.
At 2024-06-02T23:59:59Z, closing the daily interval of June 2nd.
At 2024-06-03T00:00:00Z, running for the period of June 2nd.

Answer: E

Explanation:
Scheduled job runs in CDE (which uses cron expressions for scheduling) start at the end of the first full schedule interval after the specified start date. The schedule interval is 'daily' (o e , which means midnight). The date is June 1st, 2024, at The - -schedule- start first full schedule interval must elapse after the start time. Since the schedule runs daily at midnight (00:00), the first run should process the data for June 1st. If the start date is 14:00 on June 1st, the first full daily interval (which should run at the boundary of June 2nd, 00:00) is considered missed or not fully elapsed from the 14:00 start point if not caught up (assuming catchup is disabled or the logic holds). The rule states the job starts at the end of the first full schedule interval after the start date. If the daily interval is configured to trigger at midnight (00:00): 1. Start Date: 2. The scheduler checks for the next trigger point, This is the end of the interval 2024-06-0100:00 to 2024-06-02 00:00. Because the start date 14:00 is within the first interval, Airflow determines which interval boundary is the first complete one following the effective start time. Following the specific CDE note: If the start date is 14:00, the first scheduled run is triggered at the end of the next day, after 23:59:59. The next full day after June 1st is June 2nd. The run corresponding to the end of June 2nd (which processes June 2nd's data) is triggered at If the had been 00:00, the run would be triggered at the end of the same day start date (June 1st), after 23:59:59. Since the start date is 14:00, the run is pushed to the next boundary (E).

Question: 11

A Data Engineer uses Airflow to orchestrate a nightly Spark ETL job. They define the schedule interval using a specific timedelta object. If the start_date is set to datetime(2024, 1, 1, tz= 'UTC'), which definition ensures the DAG runs exactly once per day, processing the full calendar day's data?

A.
```
schedule_interval=timedelta(hours=24)
```
B.
```
schedule_interval=timedelta(days=0)
```
C.
```
schedule_interval='0 0 '
```
D.
```
schedule_interval=timedelta(days=1)
```
E. Both A and D are technically correct, assuming appropriate Python imports.

Answer: E

Explanation:
Airflow scheduling can be defined using cron expressions or `timedelta` objects. The requirement is to run once per day. A duration of one day can be accurately represented in Python as either 24 hours or 1 day using the `timedelta` object. Option A (`timedelta(hours=24)`) specifies a duration of 24 hours. Option D (`timedelta(days=1)`) specifies a duration of 1 day. Since 24 hours equals 1 day, both A and D achieve the exact same daily scheduling frequency, provided the `timedelta` class is correctly imported (e.g., `from datetime import timedelta`). Option C achieves the same goal but uses cron syntax, which is not the primary focus of the question (which highlights the equivalence of timedelta definitions). Thus, E is the correct summary answer.

Question: 12

A Data Engineer is migrating scheduling configurations from a legacy system to Airflow 2.x within a CDE environment. The goal is to set the DAG schedule parameter (or schedule_interval in older syntax or contexts) to define the frequency of execution. Which of the following data types or values are syntactically valid for defining the scheduling interval in the DAG object definition in Python? (Select all that apply)

A. A standard cron string, such as '0 1 5' (Friday at 1 AM).
B. A Python datetime. timedelta object, representing a specific time duration.
C. The Python string value 'None' , to specify manual trigger only.
D. Airflow preset strings, such as '@daily'.
E. An integer representing the interval in seconds.

Answer: A,B,D

Explanation:

The `schedule_interval` (or modern `schedule` parameter) parameter in an Airflow DAG definition accepts several types of values to define the frequency: Option A: Standard cron expressions are a primary method for defining scheduling intervals. Option B: A `datetime.timedelta` object is explicitly supported and commonly used to define schedules based on time duration. Option C: The Python literal `None` (not the string 'None') is used to define a DAG that runs only manually. The string 'None' is not a valid scheduling value in this context. Option D: Airflow supports several preset strings (macros) that map to standard cron schedules, such as `@daily`, `@hourly`, etc.. Option E: While `timedelta` uses time units, the raw integer seconds value is not a supported type for the `schedule` parameter, which expects a cron string, `timedelta` object, or preset string.

Thank You for Trying Our Product

For More Information – Visit link below:

https://www.examsboost.com/

15 USD Discount Coupon Code:

G74JA8UF

FEATURES

- ✓ 90 Days Free Updates
- ✓ Money Back Pass Guarantee
- ✓ Instant Download or Email Attachment
- ✓ 24/7 Live Chat Support
- ✓ PDF file could be used at any Platform
- ✓ 50,000 Happy Customer