Airflow / Cloud Composer优化

开始使用

1. 规范文件名。帮助其他开发人员浏览您的 DAG 文件集合。

A。例如）team_project_workflow_version.py

2. DAG 应该是确定性的。

A。给定的输入总是会产生相同的输出。

3. DAG应该是幂等的。

A。多次触发 DAG 具有相同的效果/结果。

4.任务应该是原子的和幂等的。

A。每个任务应该负责一个可以独立于其他任务重新运行的操作。在原子化任务中，部分任务的成功意味着整个任务的成功。

5. 尽可能简化 **DAG**。

A。任务之间的依赖关系越少，越简单的 DAG 往往具有更好的调度性能，因为它们的开销更少。线性结构（例如A -> B -> C）通常比具有许多依赖性的深度嵌套树结构更有效。

6. 在每个文件的顶部和每个函数实现 **Python** 文档字符串约定。

A。Python 文档字符串约定可帮助其他开发人员和平台工程师了解您的 Airflow DAG。

b. 记录 BashOperators 的方式与记录函数的方式相同。对于不熟悉 DAG 中引用的 bash 脚本的开发人员来说，如果没有每个 bash 脚本意图的文档，就很难进行故障排除。

标准化 **DAG** 创建

7. 将所有者添加到您的**default_args**。

A。确定您是否更喜欢开发人员的电子邮件地址/ID，还是通讯组列表/团队名称。

8. 使用 `with DAG() as dag`: 代替dag 无需将 dag 对象传递给每个操作员或任务组。`= DAG()`

9. 在 **DAG ID** 中设置版本。

A。DAG 中发生任何代码更改后更新版本。

b. 这可以防止已删除的任务日志从 UI 中消失、为旧 dag 运行生成无状态任务以及 DAG 更改时的普遍混乱。

C。Airflow 开源计划在未来实现版本控制。

10. 将标签添加到您的 **DAG**。

A。通过标签过滤帮助开发人员导航 Airflow UI。

b. 按组织、团队、项目、应用程序等对 DAG 进行分组。

11.添加**DAG**描述。

A。帮助其他开发人员了解您的 DAG。

12. 在创建 **DAG** 时暂停。

A。这将有助于避免意外 DAG 运行，从而增加 Cloud Composer 环境的负载。

13. 设置 `catchup=False` 避免自动追赶使您的 **Cloud Composer** 环境超载。

14. 设置 **a** `dagrun_timeout`以避免 dags 未完成、占用 Cloud Composer 环境资源或在重试时引入冲突。

15. 通过在实例化期间将 arg 传递给 DAG，确保所有任务默认情况下都相同。`start_date`

16. 对 **DAG**使用静态 。A。动态 start_date 具有误导性，并且在清除失败的任务实例和丢失 DAG 运行时可能会导致失败。`start_date`

17.设置 `retries` 为 `default_arg` **DAG** 级别的应用，并仅在必要时对特定任务进行更细化。

A。重试次数最好的范围是 1-4 次。过多的重试会给 Cloud Composer 环境增加不必要的负载。

```
import airflow
from airflow import DAG
from airflow.operators.bash_operator import BashOperator

# Define default_args dictionary to specify default parameters of the DAG,
such as the start date, frequency, and other settings
default_args = {
   'owner': 'me',
   'retries': 2, # 2-4 retries max
   'retry_delay': timedelta(minutes=5)
}

# Use the `with` statement to define the DAG object and specify the unique
DAG ID and default_args dictionary
```

```
with DAG(
   'dag_id_v1_0_0', #versioned ID
   default_args=default_args,
   description='This is a detailed description of the DAG', #detailed
description
   start_date=datetime(2022, 1, 1),  # Static start date
   dagrun_timeout=timedelta(minutes=10), #timeout specific to this dag



   is_paused_upon_creation= True,
   catchup= False,
   tags=['example', 'versioned_dag_id'], # tags specific to this dag
   schedule_interval=None,
) as dag:
   # Define a task using the BashOperator
   task = BashOperator(
       task_id='bash_task',
       bash_command='echo "Hello World"'
   )
```

18. 定义每个回调函数应该发生什么。(发送电子邮件、记录上下文、消息 Slack 通道等)。根据 DAG，您可能可以什么也不做。

A。成功

湾 失败

c. sla_miss

d. 重试

例子：

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator

default_args = {
   'owner': 'me',
   'retries': 2, # 2-4 retries max
   'retry_delay': timedelta(minutes=5)
}

def on_success_callback(context):
   # when a task in the DAG succeeds
   print(f"Task {context['task_instance_key_str']} succeeded!")

def sla_miss_callback(context):
   # when a task in the DAG misses its SLA
   print(f"Task {context['task_instance_key_str']} missed its SLA!")

def on_retry_callback(context):
   # when a task in the DAG retries
   print(f"Task {context['task_instance_key_str']} retrying...")

def on_failure_callback(context):
   # when a task in the DAG fails
   print(f"Task {context['task_instance_key_str']} failed!")

# Create a DAG and set the callbacks
with DAG(
```

```
    'dag_id_v1_0_0',
    default_args=default_args,
    description='This is a detailed description of the DAG',
    start_date=datetime(2022, 1, 1),
    dagrun_timeout=timedelta(minutes=10),

    tags=['example', 'versioned_dag_id'],
    is_paused_upon_creation= True,
    catchup= False,
    schedule_interval=None,
    on_success_callback=on_success_callback, # what to do on success
    sla_miss_callback=on_sla_miss_callback, # what to do on sla miss
    on_retry_callback=on_retry_callback, # what to do on retry
    on_failure_callback=on_failure_callback # what to do on failure
) as dag:

    def example_task(**kwargs):
        # This is an example task that will be part of the DAG
        print(f"Running example task with context: {kwargs}")

    # Create a task and add it to the DAG
    task = PythonOperator(
        task_id="example_task",
        python_callable=example_task,
        provide_context=True,
    )
```

19. 使用 任务组 来组织任务。

例子：

```
# Use the `with` statement to define the DAG object and specify the unique DAG ID and default_args dictionary
with DAG(
   'example_dag',
   default_args=default_args,
   schedule_interval=timedelta(hours=1),
) as dag:
   # Define the first task group
   with TaskGroup(name='task_group_1') as tg1:
       # Define the first task in the first task group
       task_1_1 = BashOperator(
           task_id='task_1_1',
           bash_command='echo "Task 1.1"'
       )
```

20. 运算符变量名称和**task_id** 参数应该匹配。

```
task_id_operator_variable_name = BashOperator(
           task_id='task_id_operator_variable_name',
           bash_command='echo "Task 1.1"'
       )
```

21. 将 **SLA** 添加到您的任务中。

A。通过在任务级别设置 SLA，当任务运行时间超过预期时获取通知。

```
task_id_operator_variable_name = BashOperator(
           task_id='task_id_operator_variable_name',
```

```
            bash_command='echo "Task 1.1"',
            sla=timedelta(minutes=2),
        )
```

减少 Composer 环境的负载

22. 使用 **Jinja 模板/宏**而不是 python 函数。
A。Airflow 的模板字段允许您将环境变量和 jinja 模板中的值合并到 DAG 中。这有助于使您的 DAG 幂等（意味着多次调用不会更改结果）并防止调度程序心跳期间执行不必要的函数。
b. Airflow 引擎默认传递一些可在所有模板中访问的变量。

与最佳实践相反，以下示例基于日期时间 Python 函数定义变量：

```
# Variables used by tasks
# Bad example - Define today's and yesterday's date using datetime module
today = datetime.today()
yesterday = datetime.today() - timedelta(1)
```

如果此代码位于 DAG 文件中，则这些函数会在每个调度程序心跳上执行，这可能会导致性能不佳。更重要的是，这不会产生幂等 DAG。您无法在过去的日期重新运行之前失败的 DAG 运行，因为 datetime.today() 相对于当前日期，而不是 DAG 执行日期。

实现此目的的更好方法是使用气流变量，如下所示：

```
# Variables used by tasks
# Good example - Define yesterday's date with an Airflow variable
yesterday = {{ yesterday_ds_nodash }}
```

23. 避免创建您自己的额外气流变量。

A。元数据数据库存储这些变量并需要数据库连接来检索它们。这可能会影响 Cloud Composer 环境的性能。请改用环境变量或 Google Cloud Secrets。

24. 避免按照完全相同的时间表运行所有 **DAG**（尽可能分散工作负载）。

A。与气流宏或 time_delta 相比，更喜欢使用 cron 表达式来安排时间间隔。这允许更严格的时间表，并且更容易分散全天的工作负载，从而使您的 Cloud Composer 环境变得更轻松。

b. Crontab.guru可以帮助生成特定的 cron 表达式计划。查看此处的示例。

例子：

```
schedule_interval="*/5 * * * *", # every 5 minutes.


schedule_interval="0 */6 * * *", # at minute 0 of every 6th hour.
```

25. 除少量数据外，避免使用 **XCom**。

A。它们增加了存储空间并引入了更多与数据库的连接。

b. 如果绝对必要，请使用 JSON 字典作为值。（字典内多个值的一个连接）

26.避免在**dags/Google Cloud Storage**路径中添加不必要的对象。

A。如果必须，请将 .airflowignore 文件添加到 Airflow Scheduler 不需要解析的 GCS 路径。（sql、插件等）

27. 设置任务的执行超时。

例子：

```
# Use the `PythonOperator` to define the task
task = PythonOperator(
```

```
    task_id='my_task',
    python_callable=my_task_function,
    execution_timeout=timedelta(minutes=30),  # Set the execution timeout
to 30 minutes
    dag=dag,
)
```

28.尽可能 使用[可延迟运算符](#) 而不是 [传感器。](#) A。当可延迟操作员知道必须等待时，它可以挂起自己并释放工作人员，并将恢复工作的工作交给触发器。因此，虽然它挂起（延迟），但它不会占用工作插槽，并且您的集群在空闲 Operator 或 Sensor 上浪费的资源将更少/更少。

例子：

```
PYSPARK_JOB = {
    "reference": { "project_id": "PROJECT_ID" },
    "placement": { "cluster_name": "PYSPARK_CLUSTER_NAME" },
    "pyspark_job": {
        "main_python_file_uri":
"gs://dataproc-examples/pyspark/hello-world/hello-world.py"
    },
}

DataprocSubmitJobOperator(
        task_id="dataproc-deferrable-example",
        job=PYSPARK_JOB,
        deferrable=True,
    )
```

29. 使用传感器时，请始终定义 `mode`、`poke_interval`、和 `timeout`。

A。传感器需要 Airflow 工作人员运行。

b. 传感器每 n 秒检查一次（即 poke_interval < 60）？使用`mode=poke`。传感器`mode=poke`将每 n 秒连续轮询并保留 Airflow 工作线程资源。

C。传感器每 n 分钟检查一次（即 poke_interval >= 60）？使用`mode=reschedule`。传感器`mode=reschedule`将在 Poke 间隔之间释放 Airflow 工作线程资源。

例子：

```
table_partition_sensor = BigQueryTablePartitionExistenceSensor(
      project_id="{{ project_id }}",
      task_id="bq_check_table_partition",
      dataset_id="{{ dataset }}",
      table_id="comments_partitioned",
      partition_id="{{ ds_nodash }}",
      mode="reschedule"
      poke_interval=60,
      timeout=60 * 5
  )
```

30. 将处理卸载到外部服务（BigQuery、Dataproc、Cloud Functions等），以最大程度地减少 Cloud Composer 环境的负载。

A。这些服务通常有自己的 Airflow Operator 供您使用。

31. 不要使用子 **DAG**。

A。子 DAG 是旧版本 Airflow 中的一项功能，允许用户在 DAG 内创建可重用的任务组。然而，Airflow 2.0 弃用了子 DAG，因为它们会导致性能和功能问题。

32. 使用**Pub/Sub**实现 **DAG** 到 **DAG** 的依赖关系。

A。以下是多集群/dag-to-dag 依赖关系的示例。

33. 使 **DAG** 加载速度更快。

A。避免不必要的“顶级”Python 代码。在 DAG 之外具有许多导入、变量和函数的 DAG 将为 Airflow Scheduler 引入更长的解析时间，进而降低 Cloud Composer / Airflow 的性能和可扩展性。

b. 在 DAG 内移动导入和函数可以减少解析时间（以秒为单位）。

C。确保开发的 DAG 不会过多增加DAG 解析时间。

例子：

```
import airflow
from airflow import DAG
from airflow.operators.python_operator import PythonOperator

# Define default_args dictionary
default_args = {
   'owner': 'me',
   'start_date': datetime(2022, 11, 17),
}

# Use with statement and DAG context manager to instantiate the DAG
with DAG(
   'my_dag_id',
   default_args=default_args,
   schedule_interval=timedelta(days=1),
) as dag:
```

```
  # Import module within DAG block
  import my_module # DO THIS

  # Define function within DAG block
  def greet(): # DO THIS
      greeting = my_module.generate_greeting()
      print(greeting)

  # Use the PythonOperator to execute the function
  greet_task = PythonOperator(
      task_id='greet_task',
      python_callable=greet
  )
```

改进开发和测试

34. 实施**"**自检**"**（通过传感器或可延迟操作员）。

A。为了确保任务按预期运行，您可以向 DAG 添加检查。例如，如果某个任务将数据推送到 BigQuery 分区，您可以在下一个任务中添加检查以验证分区是否生成以及数据是否正确。

例子：

```
# ------------------------------------------------------------
  # Transform source data and transfer to partitioned table
  # ------------------------------------------------------------
```

```
  create_or_replace_partitioned_table_job = BigQueryInsertJobOperator(
      task_id="create_or_replace_comments_partitioned_query_job",
      configuration={
          "query": {
              "query": 'sql/create_or_replace_comments_partitioned.sql',
              "useLegacySql": False,
          }
      },
      location="US",
  )

  create_or_replace_partitioned_table_job_error =
dummy_operator.DummyOperator(
      task_id="create_or_replace_partitioned_table_job_error",
      trigger_rule="one_failed",
  )

  create_or_replace_partitioned_table_job_ok =
dummy_operator.DummyOperator(
      task_id="create_or_replace_partitioned_table_job_ok",
trigger_rule="one_success"
  )

  # ------------------------------------------------------------
  # Determine if today's partition exists in comments_partitioned
  # ------------------------------------------------------------
```

```
table_partition_sensor = BigQueryTablePartitionExistenceSensor(
      project_id="{{ project_id }}",
      task_id="bq_check_table_partition",
      dataset_id="{{ dataset }}",
      table_id="comments_partitioned",
      partition_id="{{ ds_nodash }}",
      mode="reschedule"
      poke_interval=60,
      timeout=60 * 5
  )

  create_or_replace_partitioned_table_job >> [
      create_or_replace_partitioned_table_job_error,
      create_or_replace_partitioned_table_job_ok,
  ]
  create_or_replace_partitioned_table_job_ok >> table_partition_sensor
```

35.寻找机会通过Python代码动态生成类似的任务/任务组/DAG。

A。这可以简化和标准化 DAG 的开发过程。

例子：

```
import airflow
from airflow import DAG
from airflow.operators.python_operator import PythonOperator

def create_dag(dag_id, default_args, task_1_func, task_2_func):
   with DAG(dag_id, default_args=default_args) as dag:
```

```
        task_1 = PythonOperator(
            task_id='task_1',
            python_callable=task_1_func,
            dag=dag
        )
        task_2 = PythonOperator(
            task_id='task_2',
            python_callable=task_2_func,
            dag=dag
        )
        task_1 >> task_2
        return dag

def task_1_func():
   print("Executing task 1")

def task_2_func():
   print("Executing task 2")

default_args = {
   'owner': 'me',
   'start_date': airflow.utils.dates.days_ago(2),
}

my_dag_id = create_dag(
   dag_id='my_dag_id',
   default_args=default_args,
```

```
    task_1_func=task_1_func,
    task_2_func=task_2_func
)
```

36. 为 **DAG** 实施单元测试

例子：

```
from airflow import models
from airflow.utils.dag_cycle_tester import test_cycle


def assert_has_valid_dag(module):
    """Assert that a module contains a valid DAG."""

    no_dag_found = True

    for dag in vars(module).values():
        if isinstance(dag, models.DAG):
            no_dag_found = False
            test_cycle(dag)  # Throws if a task cycle is found.

    if no_dag_found:
        raise AssertionError('module does not contain a valid DAG')
```

37. 通过Composer 本地开发 CLI 工具执行本地开发。

A。Composer 本地开发 CLI 工具通过在本地运行 Airflow 环境，简化了 Cloud Composer 2 的 Apache Airflow DAG 开发。此本地 Airflow 环境使用特定 Cloud Composer 版本的映像。

38. 如果可能，请保留暂存 **Cloud Composer** 环境，以便在部署到生产环境之前全面测试完整的 **DAG** 运行。

A。参数化您的 DAG 以更改变量，例如 Google Cloud Storage 操作的输出路径或用于读取配置的数据库。不要将值硬编码到 DAG 内，然后根据环境手动更改它们。

39. 使用Pylint或Flake8等**Python linting** 工具来实现标准化代码。

40. 使用Black或YAPF等**Python** 格式化工具来实现标准化代码。