Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ **ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ**

КАФЕДРА **КОМПЬЮТЕРНЫЕ СИСТЕМЫ И СЕТИ (ИУ6)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.04.01 Информатика и вычислительная техника**

МАГИСТЕРСКАЯ ПРОГРАММА **09.04.01/07 Интеллектуальные системы анализа, обработки и интерпретации больших данных**

О Т Ч Е Т

по рубежному контролю № 2

Название: Обработка больших данных с Apache Airflow

Дисциплина: Искусство аналитической работы с большими данными

Студент ИУ6-22М _____ С.В. Астахов
(Группа) (Подпись, дата) (И.О. Фамилия)

Преподаватель _____ Д.В. Березкин
(Подпись, дата) (И.О. Фамилия)

Москва, 2024

Содержание

Введение

В предыдущем реферате была освещена роль Apache Airflow в реализации ETL-процессов построения витрин данных.

В данном реферате более детально рассмотрены возможности и специфика разработки с использованием данного ПО, такие как:

- планирование и выполнение конвейеров;
- инкрементальная загрузка и обратное заполнение;
- рекомендации по разработке задач — атомарность;
- рекомендации по разработке задач — идемпотенстность;
- реализация ветвления.

В предыдущем реферате мы уже рассмотрели базовые функции и задачи Apache Airflow. Очевидно, основная идея Airflow довольно проста, однако реализация данного ПО имеет крайне высокое качество и множество небольших полезных функций, которые позволили ему обрести мировую популярность. Давайте рассмотрим детали работы и функции Apache Airflow подробнее [1].

1. Планирование и выполнение конвейеров.

После того как вы определили структуру вашего конвейера (конвейеров) в виде DAG, Airflow позволяет вам определить параметр schedule_interval для каждого графа, который точно решает, когда ваш конвейер будет запущен Airflow. Таким образом, вы можете дать указание Airflow выполнять ваш граф каждый час, ежедневно, каждую неделю и т. д. Или даже использовать более сложные интервалы, основанные на выражениях, подобных Cron.

Чтобы увидеть, как Airflow выполняет графы, кратко рассмотрим весь процесс, связанный с разработкой и запуском DAG. На высоком уровне Airflow состоит из трех основных компонентов (рисунок 1):

- планировщик Airflow – анализирует DAG, проверяет параметр schedule_interval и (если все в порядке) начинает планировать задачи DAG для выполнения, передавая их воркерам Airflow;
- воркеры1 (workers) Airflow – выбирают задачи, которые запланированы для выполнения, и выполняют их. Таким образом, они несут ответственность за фактическое «выполнение работы»;
- веб-сервер Airflow – визуализирует DAG, анализируемые планировщиком, и предоставляет пользователям основной интерфейс для отслеживания выполнения графов и их результатов.

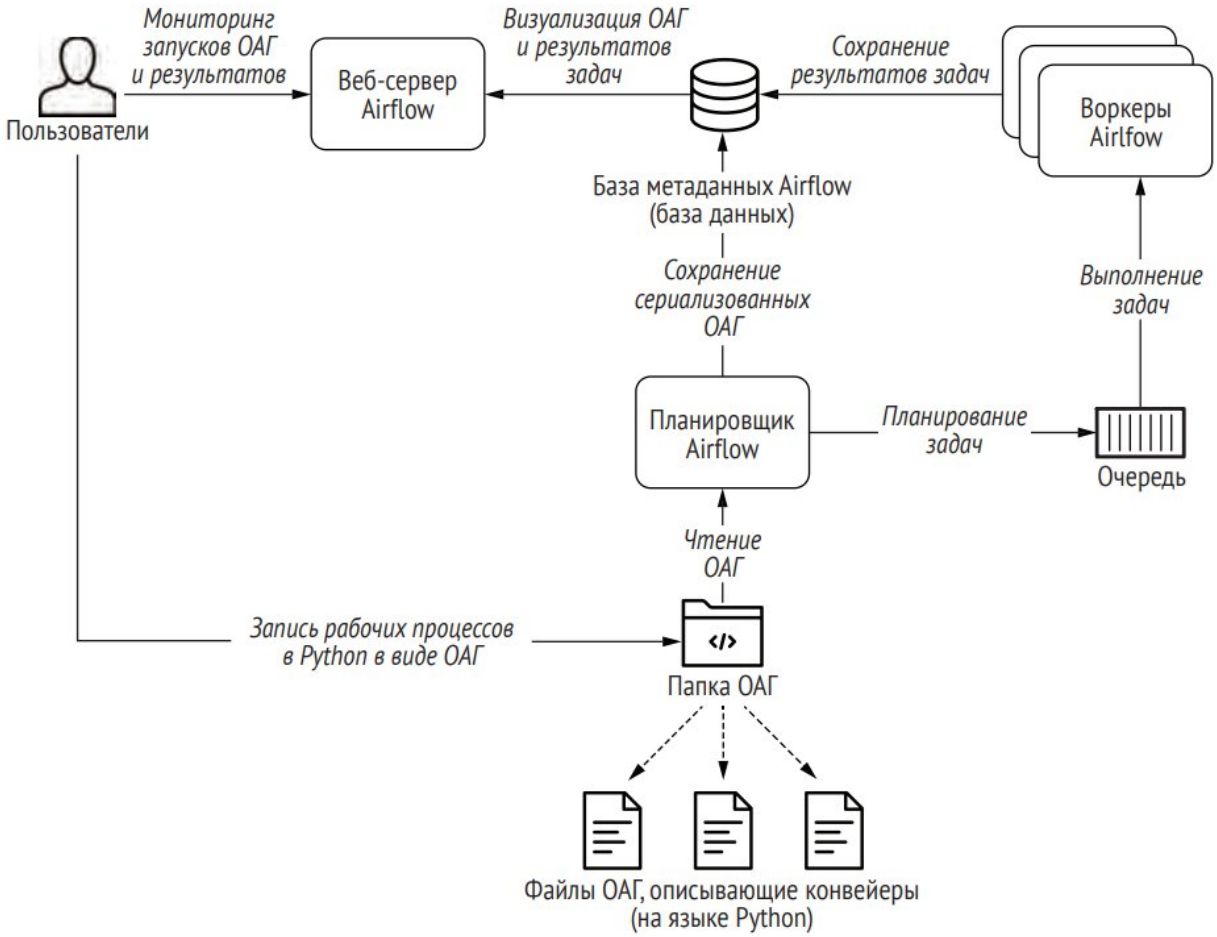

Рисунок 1 — обзор основных компонентов Airflow

Сердцем Airflow, вероятно, является планировщик, поскольку именно здесь происходит большая часть магии, определяющей, когда и как будут выполняться ваши конвейеры. На высоком уровне планировщик выполняет следующие шаги (рисунок 2):

После того как пользователи написали свои рабочие процессы в виде DAG, файлы, содержащие эти графы, считываются планировщиком для извлечения соответствующих задач, зависимостей и интервалов каждого DAG.

После этого для каждого графа планировщик проверяет, все ли в порядке с интервалом с момента последнего чтения. Если да, то задачи в графе планируются к выполнению.

Для каждой задачи, запускаемой по расписанию, планировщик затем проверяет, были ли выполнены зависимости (= вышестоящие задачи) задачи. Если да, то задача добавляется в очередь выполнения.

Планировщик ждет несколько секунд, прежде чем начать новый цикл, перескакивая обратно к шагу 1.

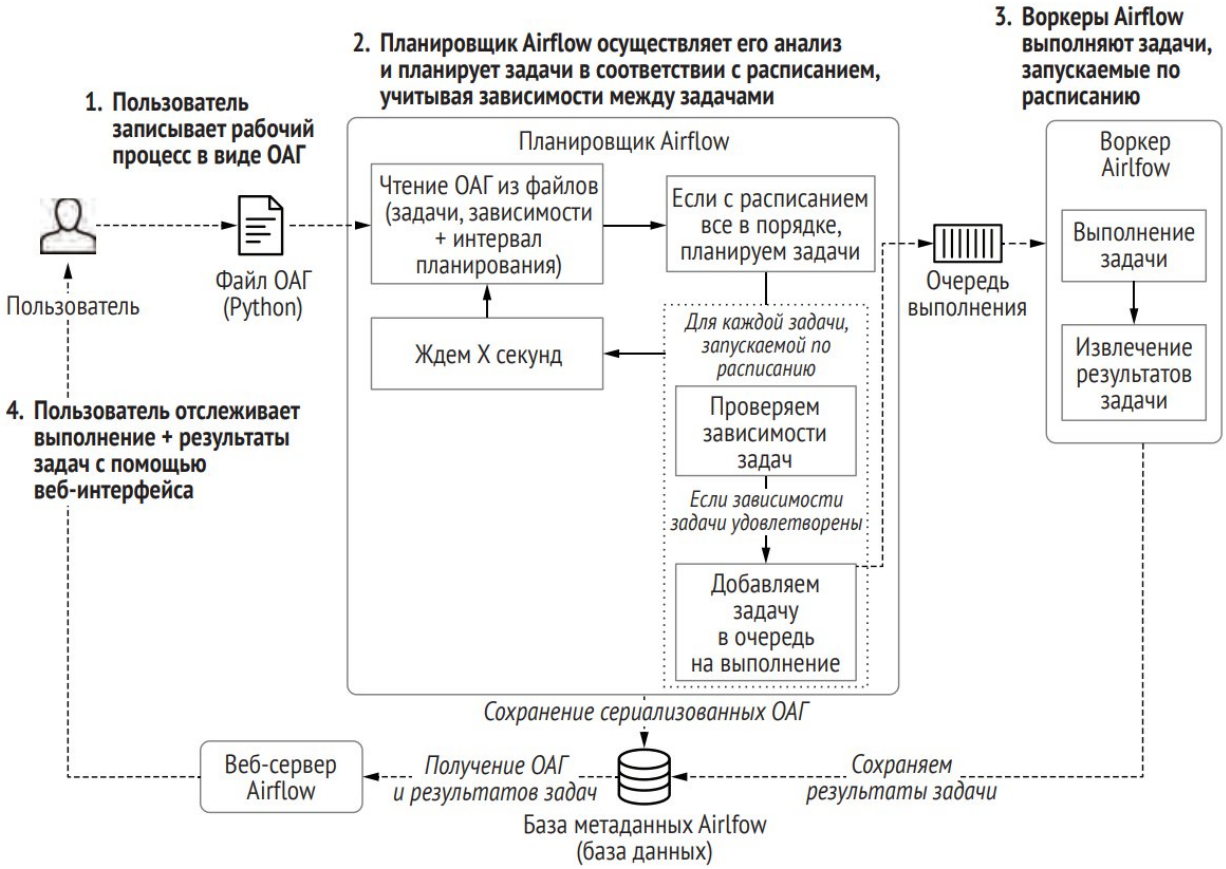

Рисунок 2 — Обзор основных компонентов Airflow (в динамике)

После того как задачи поставлены в очередь на выполнение, с ними уже работает пул воркеров Airflow, которые выполняют задачи параллельно и отслеживают их результаты. Эти результаты передаются в базу метаданных Airflow, чтобы пользователи могли отслеживать ход выполнения задач и просматривать журналы с помощью веб-интерфейса Airflow (интерфейс, предоставляемый веб-сервером Airflow).

2. Инкрементальная загрузка и обратное заполнение (backfilling).

Одно из мощных функций семантики планирования Airflow состоит в том, что вышеуказанные интервалы не только запускают DAG в определенные моменты времени (аналогично, например, Cron), но также предоставляют подробную информацию о них и (ожидаемых) следующих интервалах.

По сути, это позволяет разделить время на дискретные интервалы (например, каждый день, неделю и т. д.) и запускать DAG с учетом каждого из этих интервалов. Такое свойство интервалов Airflow неоценимо для реализации эффективных конвейеров обработки данных, поскольку позволяет создавать дополнительные конвейеры.

В этих инкрементных конвейерах каждый запуск DAG обрабатывает только данные для соответствующего интервала времени (дельта данных), вместо того чтобы каждый раз повторно обрабатывать весь набор данных. Это может обеспечить значительную экономию времени и средств, особенно в случае с большими наборами данных, за счет предотвращения дорогостоящего пересчета существующих результатов.

Эти интервалы становятся еще более мощными в сочетании с концепцией обратного заполнения (рисунки 3-4), позволяющей выполнять новый DAG для интервалов, которые имели место в прошлом[2]. Эта функция позволяет легко создавать новые наборы архивных данных, просто запуская DAG с учетом этих интервалов. Более того, очистив результаты прошлых запусков, вы также можете использовать эту функцию Airflow, чтобы повторно запускать любые архивные задачи, если вы вносите изменения в код задачи, что при необходимости позволяет повторно обрабатывать весь набор данных.

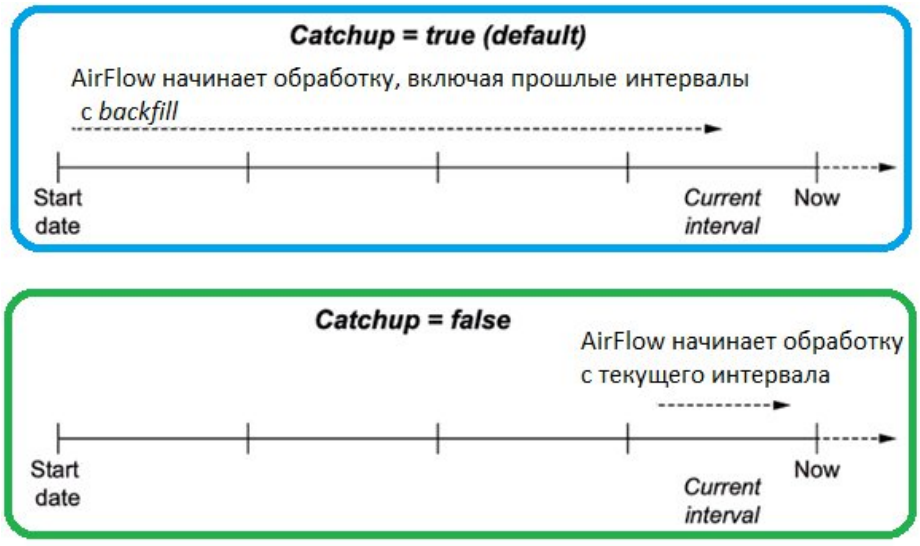

Рисунок 3 — обратное заполнение в Airflow

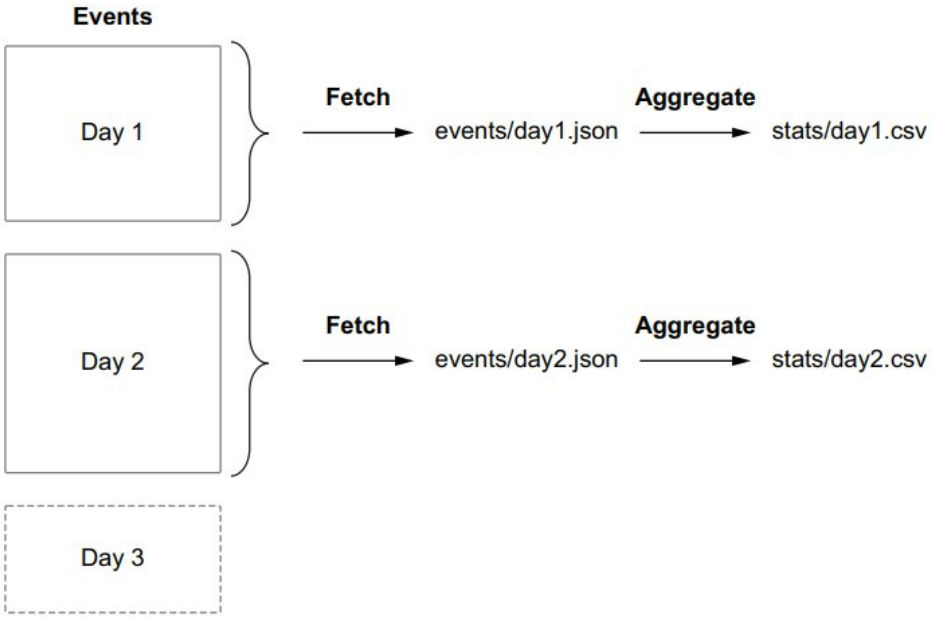

Рисунок 4 — инкрементальная обработка данных

3. Рекомендации по разработке задач — атомарность.

Хотя Airflow выполняет большую часть работы по повторному запуску задач, мы должны убедиться, что наши задачи соответствуют определенным ключевым свойствам для получения надлежащих результатов. Два ключевых свойства: атомарность и идемпотентность [3].

Термин атомарность часто используется в СУБД, где атомарная транзакция рассматривается как неделимая последовательность операций с базой данных, при которой либо выполняются все операции, либо не выполняется ничего.

Аналогично, в Airflow задачи должны быть определены таким образом, чтобы выполняться либо успешно и приводить к какому-либо надлежащему результату, либо выдавать сбой, не влияющий на состояние системы.

В качестве примера, допустим, что мы решили добавить к примеру из рисунка 4 нотификацию пользователей по email.

Мы можем записать csv-файл и отправить письма в рамках одного шага Airflow (листинг 1), но тогда мы нарушим атомарность, так как может произойти следующий сценарий

- шаг запустился;
- произошел экспорт данных в файл;

– нотификация не отправилась из-за нестабильной сети.

Листинг 1 — неатомарная задача

```
def _calculate_stats(**context):
    """Calculates event statistics."""
    input_path = context["templates_dict"]["input_path"]
    output_path = context["templates_dict"]["output_path"]

    events = pd.read_json(input_path)
    stats = events.groupby(["date", "user"]).size().reset_index()
    stats.to_csv(output_path, index=False)

    send_stats(stats, email="user@example.com")
```

В данном случае, если мы перезапустим “упавший” шаг, то повторно будет выполнена лишняя работа, на месте которой могли быть куда более сложные вычисления. Кроме того, будет сложнее понять и устранить причины ошибки. Именно поэтому задачу следовало разделить на два отдельных шага (рисунок 5, листинг 2).

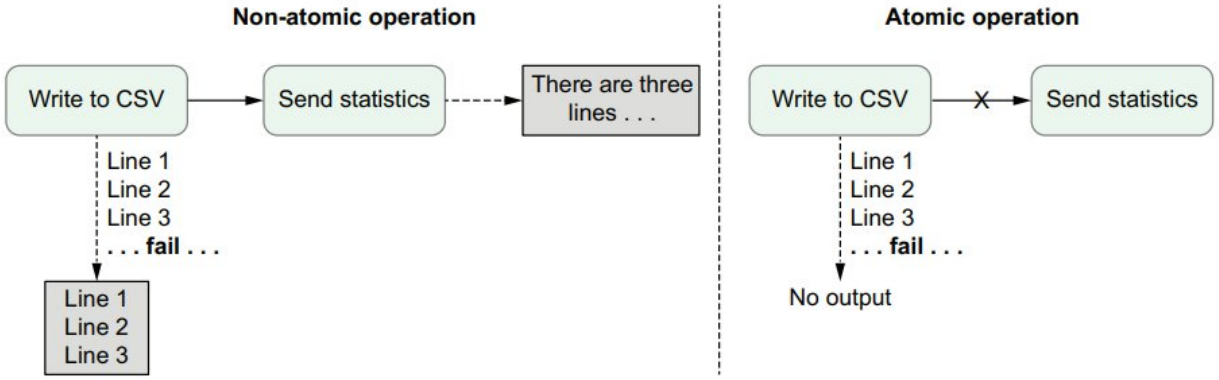

Рисунок 5 — атомарность задач (шагов)

Листинг 2 — атомарная задача

```
def _send_stats(email, **context):
    stats = pd.read_csv(context["templates_dict"]["stats_path"])
    email_stats(stats, email=email)



send_stats = PythonOperator(
    task_id="send_stats",
    python_callable=_send_stats,
    op_kwargs={"email": "user@example.com"},
```

```
    templates_dict={"stats_path": "data/stats/{{ds}}.csv"},
    provide_context=True,
    dag=dag,
)


calculate_stats >> send_stats
```

4. Рекомендации по разработке задач — идемпотентность.

Концепция идемпотентности связана с идеей атомарности и описывает свойство определенных операций в математике и информатике возвращать один и тот же результат при повторных запусках. Идемпотентные операции можно применять несколько раз и каждый раз получать один и тот же выход.

Например, в математике идемпотентными операциями являются сложение с нулём, умножение на единицу, взятие модуля числа, выбор максимального значения, вычисление наибольшего общего делителя, возведение в степень единицы. В информатике идемпотентными являются GET-запросы в протоколе HTTP: сервер возвращает идентичные ответы на одни и те же GET-запросы, если сам ресурс не изменился. Это позволяет корректно кэшировать ответы, снижая нагрузку на сеть.

Для Airflow цепочка задач DAG считается идемпотентной, если повторный запуск одного и того же DAG Run с одними и теми же входными данными дает тот же эффект, что и его однократный запуск. Этого можно достичь, спроектировав каждую отдельную задачу в вашей DAG так, чтобы она была идемпотентной. Другими словами, если повторный запуск задачи без изменения входных данных дает тот же результат, ее можно считать идемпотентной. Разработка идемпотентных DAG и задач сокращает время восстановления после сбоев и предотвращает потерю данных.

Чтобы понять, как это работает, снова рассмотрим практический пример из дата-инженерии. Предположим, нужно получить данные из базы за определенный день и записать результаты в CSV-файл. Повторное выполнение этой задачи в тот же день должно перезаписать существующий файл, выдавая один и тот же результат при каждом выполнении. Обычно

задачи на запись должны проверять существующие данные, перезаписывать или использовать операции UPSERT для соответствия правилам идемпотентности (рисунок 6).

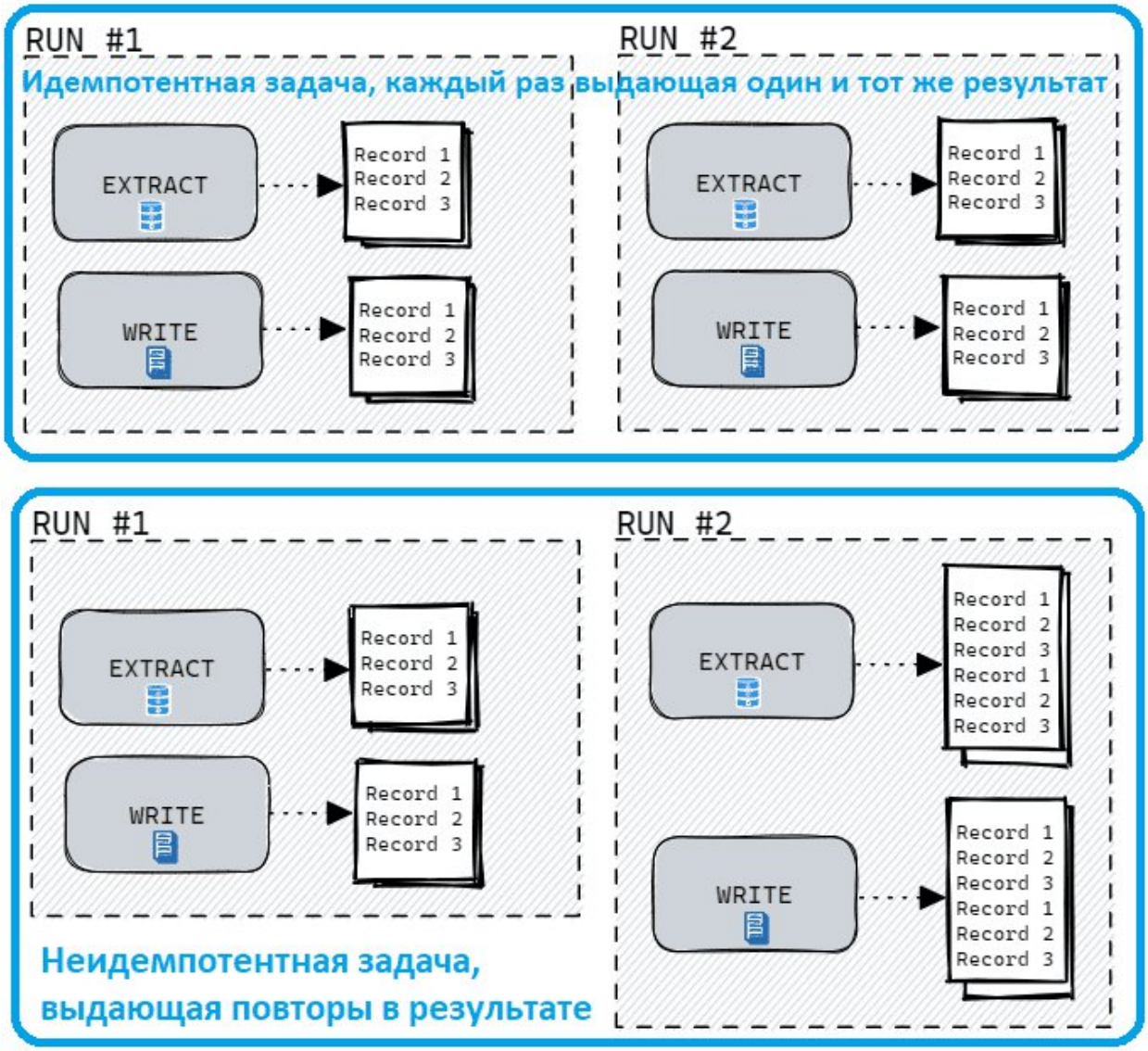

Рисунок 5 — идемпотенстность задач (шагов)

На практике реализовать принцип идемпотентности поможет использование шаблонных полей в Airflow, чтобы извлекать значения в DAG через переменные среды и шаблоны Jinja — текстового шаблонизатора для Python. По сравнению с использованием функций Python использование шаблонных полей помогает поддерживать идемпотентность пользовательских DAG и гарантирует, что функции не выполняются при каждом такте планировщика. А для некоторых сценариев, таких как инкрементная фильтрация записей идемпотентность DAG позволяет добиться корректных

результатов. Например, в DAG, который выполняется каждый час, каждый запуск должен обрабатывать только записи за эти 60 минут, а не весь набор данных. Когда результаты каждого запуска DAG представляют лишь небольшую часть общего набора данных, сбой в одном подмножестве данных не помешает успешному завершению остальных запусков DAG. И если DAG являются идемпотентными, дата-инженер может повторно запустить конвейер обработки только для тех данных, где произошел сбой, а не для повторной обработки всего датасета.

Реализовать такую идемпотентность можно двумя способами:

- дата последнего изменения, когда каждая запись в исходной системе имеет столбец со временной отметкой последнего изменения записи. Запуск DAG ищет записи, которые были обновлены в течение определенных дат из этого столбца. В нашем примере с ежечасным запуском DAG он будет обрабатывать записи, которые попадают в период между началом и концом каждого часа. Если какой-либо из запусков завершится неудачно, это не повлияет на другие запуски.
- последовательные идентификаторы записей. Если дата последнего изменения недоступна, для добавочных загрузок можно использовать последовательность или инкрементный идентификатор. Эта отлично работает, когда исходные записи только добавляются и никогда не обновляются.

5. Ветвление в Apache Airflow.

Зачастую при проектировании ETL-процессов в Airflow необходимо реализовывать ветвления в таких ситуациях, как, например [4]:

- переход от старого хранилища данных к новому — в зависимости от даты построения, например, срезного объекта, берутся источники из старого, либо нового хранилища;
- запуск основной логики ETL-процесса (sql-операторов или spark-операторов) только в случае доступности инфраструктуры, отсутствия параллельного запуска того же процесса и т.д. (например, для

ежемесячных отчетов существует практика нескольких запусков DAG в течении месяца с проверкой, что успешных запусков ранее в течении месяца не было, это делается для обхода технических проблем в инфраструктуре, чтобы не перезапускать “упавшие” из-за технических работ на кластере DAG вручную);

- обработка ошибок (рассылка уведомлений, откат изменений).

Для реализации ветвления в airflow существует целый набор branch операторов:

- BranchPythonOperator на основе PythonOperator, который принимает функцию Python в качестве входных данных и возвращает список идентификаторов задач, которые будут запущены после его исполнения;
- ShortCircuitOperator, который также принимает вызываемый объект Python, возвращая True или False в зависимости от логики. Если возвращено значение True, DAG продолжит работу, а если возвращено значение False, все нижестоящие задачи будут пропущены;
- BranchSQLOperator — активирует ветку в зависимости от того, возвращает ли SQL-запрос True или False;
- BranchDayOfWeekOperator — активирует ветку в зависимости от того, равен ли текущий день недели заданному параметру week_day;
- BranchDateTimeOperator — активирует ветку в зависимости от того, попадает ли текущее время в промежуток между target_lower и target_upper.

Рассмотрим пример реализации ветвления в Airflow. Допустим, мы переключаемся со старой ERP-системы на новую и в зависимости от даты запуска DAG необходимо выбрать какая из них будет источником данных (рисунок 6).

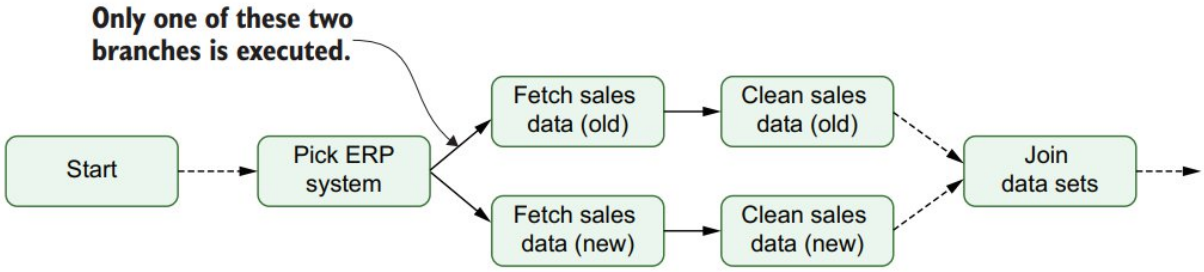

Рисунок 6 — DAG с ветвлением

Реализуется такое ветвление программным кодом, приведенным в листинге 3.

Листинг 3 — ветвление в AIrflow

```
def _pick_erp_system(**context):
    if context["execution_date"] < ERP_SWITCH_DATE:
        return "fetch_sales_old"
    else:
        return "fetch_sales_new"

sales_branch = BranchPythonOperator(
  task_id='sales_branch',
  provide_context=True,
  python_callable=_pick_erp_system,
)

sales_branch >> [fetch_sales_old, fetch_sales_new]
```

Кроме этого, стоит учесть, что по умолчанию, все шаги DAG имеют правило запуска “all_success”, то есть будут запущены, только если все входящие шаги были завершены успешно. Поэтому, чтобы шаг объединения датасетов не был пропущен, необходимо выставить для него правило запуска “none_failed” (входящие шаги или пропущены или завершены успешно, но не с ошибкой) [5].

Кроме этих двух значений параметра существуют следующие:

- all_success (по умолчанию): все вышестоящие задачи завершены успешно;
- all_failed: все вышестоящие задачи находятся в состоянии failed или upstream_failed;

- all_done: все вышестоящие задачи завершены своим выполнением;
- all_skipped: все вышестоящие задачи находятся в пропущенном состоянии;
- one_failed: по крайней мере, одна вышестоящая задача завершилась ошибкой (не ожидает выполнения всех вышестоящих задач);
- one_success: по крайней мере, одна исходная задача выполнена успешно (не требуется ждать, пока будут выполнены все исходные задачи);
- one_done: по крайней мере, одна исходная задача выполнена успешно или завершилась неудачей;
- none_failed: все вышестоящие задачи не завершились ошибкой или upstream_failed - то есть все вышестоящие задачи завершились успешно или были пропущены;
- none_failed_min_one_success: все вышестоящие задачи не завершились неудачей или не завершились ошибкой восходящего потока, и по крайней мере одна вышестоящая задача была выполнена успешно;
- none_skipped: ни одна из вышестоящих задач не находится в пропущенном состоянии, то есть все вышестоящие задачи находятся в состоянии success, failed или upstream_failed;
- always: зависимостей нет вообще, запускает задачу в любое время.

На этом знаомство с базовым функционалом, необходимым для работы с Apache Airflow может быть закончено.

Заключение

В данном реферате были детально рассмотрены возможности и специфика разработки с использованием Apache Airflow, такие как:

- планирование и выполнение конвейеров;
- инкрементальная загрузка и обратное заполнение;
- рекомендации по разработке задач — атомарность;
- рекомендации по разработке задач — идемпотенстность;
- реализация ветвления.

Что позволяет оценить степень важности данного ПО при построении ETL-процессов и получить представление о процессе их проектирования.

Список использованных источников

1. Harenslak, B. Data Pipelines with Apache Airflow / B. Harenslak, J. de Ruiter. – Shelter Island : Manning Publications Co., 2021. – 482 с. – ISBN 9781617296901.
2. Вперед в прошлое: backfill для DAG в Apache AirFlow // BigDataSchool : сайт. – URL: https://bigdataschool.ru/blog/dag-backfilling-in-airflow.html (дата обращения: 05.04.2024)
3. Атомарность и идемпотентность в Apache AirFlow // BigDataSchool : сайт. – URL: https://bigdataschool.ru/blog/atomicity-and-idempotency-in-airflow.html (дата обращения: 08.04.2024)
4. Как построить логически сложный ETL-конвейер: ветвления DAG в Apache AirFlow // BigDataSchool : сайт. – URL: https://bigdataschool.ru/blog/branching-in-dag-airflow-with-operators.html (дата обращения: 10.04.2024)
5. Trigger Rules // Apache Airflow Docs : сайт. – URL: https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dags.html#concepts-trigger-rules (дата обращения: 12.04.2024)