ТИТУЛ

ЗАДАНИЕ

РЕФЕРАТ

Отчет — 30 с., 19 рис., 9 ист.

БОЛЬШИЕ ДАННЫЕ, ВИТРИНЫ ДАННЫХ, ХРАНИЛИЩА ДАННЫХ, APACHE AIRFLOW, ETL-ПРОЦЕСС.

Объектом исследования являются технологии и программное обеспечение, используемое для обработки больших данных и построения витрин данных.

Цель работы — анализ технологий построения систем обработки больших данных и выявление требований к архитектуре такой системы.

В процессе работы был проведен анализ проблематики построения современных витрин данных, рассмотрены вопросы хранения исторических данных и организации ETL-процессов с Apache Airflow.

Актуальность работы обусловлена тенденцией к переходу с проприетарных BI-инструментов на программное обеспечение с открытым исходным кодом. Такая тенденция обусловлена несколькими факторами, такими как ограничение на покупку иностранного ПО из-за международных санкций, высокой стоимостью проприетарных BI-инструментов и т.д.

СОДЕРЖАНИЕ

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ.....5
ВВЕДЕНИЕ.....6
1. Витрины данных.....7
1.2. Проблематика построения и понятие витрин данных.....7
1.2. Версионность витрин данных.....10
2. Apache Airflow.....15
2.1. Введение Apache Airflow.....15
2.2. Планирование и выполнение конвейеров. 19
2.3. Инкрементальная загрузка и обратное заполнение (backfilling). 21
2.4. Рекомендации по разработке задач — атомарность. 23
2.5. Рекомендации по разработке задач — идемпотентность. 25
2.6. Ветвление в Apache Airflow. 27
ЗАКЛЮЧЕНИЕ.....32
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....33

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

Витрина данных — подмножество (срез) хранилища данных, представляющее собой массив тематической, узконаправленной информации, ориентированной, например, на пользователей одной рабочей группы или подразделения.

Хранилище данных — предметно-ориентированная информационная база данных, специально разработанная и предназначенная для подготовки отчётов и бизнес-анализа с целью поддержки принятия решений в организации.

ВВЕДЕНИЕ

В настоящее время в контексте словосочетания “большие данные” наиболее популярны темы, связанные с машинным обучением, нейронными сетями и искуственным интеллектом.

Действительно, искать скрытые закономерности, визуализировать заранее собранные датасеты, применять к ним методы машинного обучения и математической статистики может быть довольно занимательно (и, конечно, же полезно для бизнеса).

Однако, зачастую, даже люди, профессионально занимающиеся информационными технологиями, не осознают всей технической сложности, скрывающейся за «расчетом» обычных витрин данных.

В данной работе освещается основная проблематика расчета витрин данных в современных реалиях, а также использование ПО Apache Airflow для оркестрации расчета витрин данных и ETL-процессов.

1. Витрины данных

1.2. Проблематика построения и понятие витрин данных

Витрина данных (Data Mart) — подмножество (срез) хранилища данных, представляющее собой массив тематической, узконаправленной информации, ориентированной, например, на пользователей одной рабочей группы или подразделения.

Обычно это данные по определенной теме или задаче в компании. Например, витрина с данными о заказчиках для отдела маркетинга может содержать подробные данные по договорам, истории заказов и поставок, оплатах, звонках и адресах доставки [1]. Ничего лишнего, только нужные и актуальные очищенные данные, полученные из других ИС предприятия. Таких витрин даже на одном предприятии может быть множество (рисунок 1).

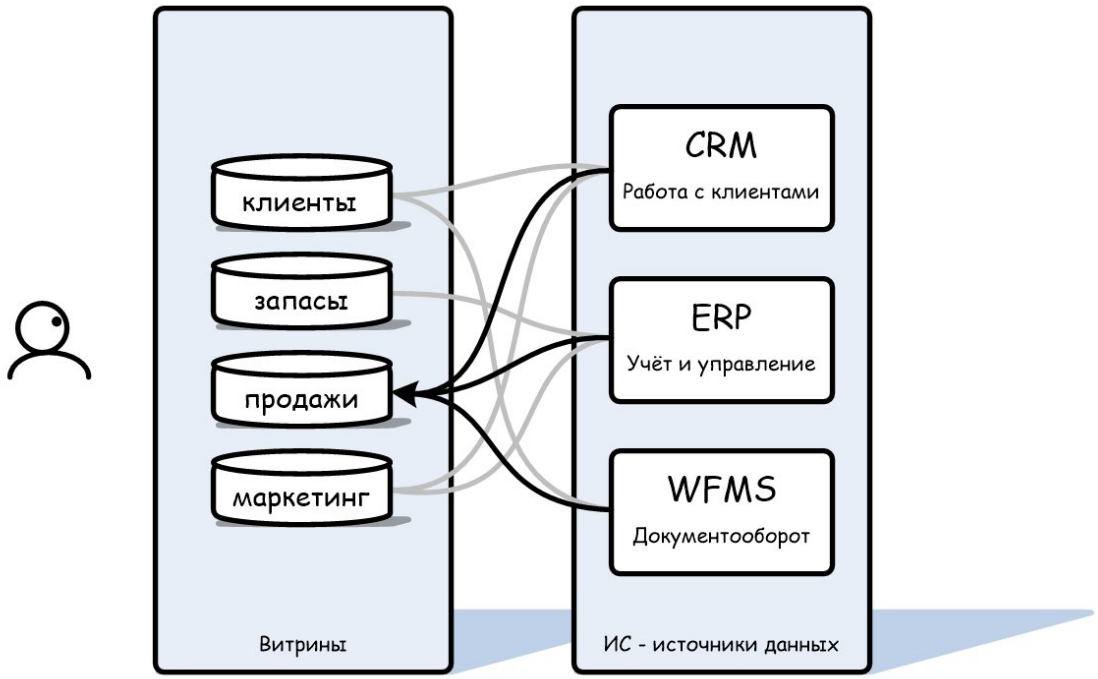

Рисунок 1 — источники и витрины данных

Теперь, попытаемся понять, что из себя представляет витрина данных на более прикладном уровне, для этого пойдем от обратного и перечислим все, что внешне напоминает или ассоциируется с витринами данных, но не является по определению или не удовлетворяет требованиям к эффективности и надежности.

Представления (view). Многим, кто имел дело с СУБД на уровне простых запросов, курсовых и pet-проектов, а затем лишь в виде готовых оберток в приложениях на работе может показаться, что запрос на создание описанных выше сущностей может быть реализован с помощью простых SQL-представлений. Действительно, представление это очень понятная программная реализация данных, однако в реальных условиях она не применима. Основных причин (как минимум) две — невозможность выдержать высокую нагрузку и сложность проектирования. Реальные базы данных, например, такие, как база данных системы управления контентом Drupal (рисунок 2) [2], включают сотни таблиц, а процесс построения витрины на практике захватывает до нескольких десятков таблиц,что затруднительно обработать в рамках одного запроса хотя бы с точки зрения его написания.

Иерархия представлений. Чтобы облегчить составление витрины, можно представить ее как иерархию представлений. Такой подход уже лучше описывает декомпозицию процесса обработки данных в ETL. Однако, в этом случае запрос к витрине все еще будет обрабатываться оптимизатором как единый и будет (вероятно) неоптимален, так как даже современные оптимизаторы имеют довольно ограниченные возможности по сравнению со сложностью ETL-процессов, особенно если по какой-то из таблиц-источников не собрана статистика. Но, даже, если мы построим оптимальный запрос, единовременная загрузка данных из всех источников будет создавать слишком большую нагрузку на оперативную память.

Иерархия материализованных представлений. Итак, мы пришли к тому, что следует «приземлять» результаты вычислений на диск. Но, материализованное представление будет пересчитываться каждый раз, когда изменяются данные хоть в одном источнике. Имея витрину как каскад таких представлений (а сами витрины тоже могут быть зависимы) получим почти постоянные перестроения. Кроме того, объем данных в реальных процессах обычно слишком велик, чтоб пересчитывать всю витрину целиком было в принципе возможно.

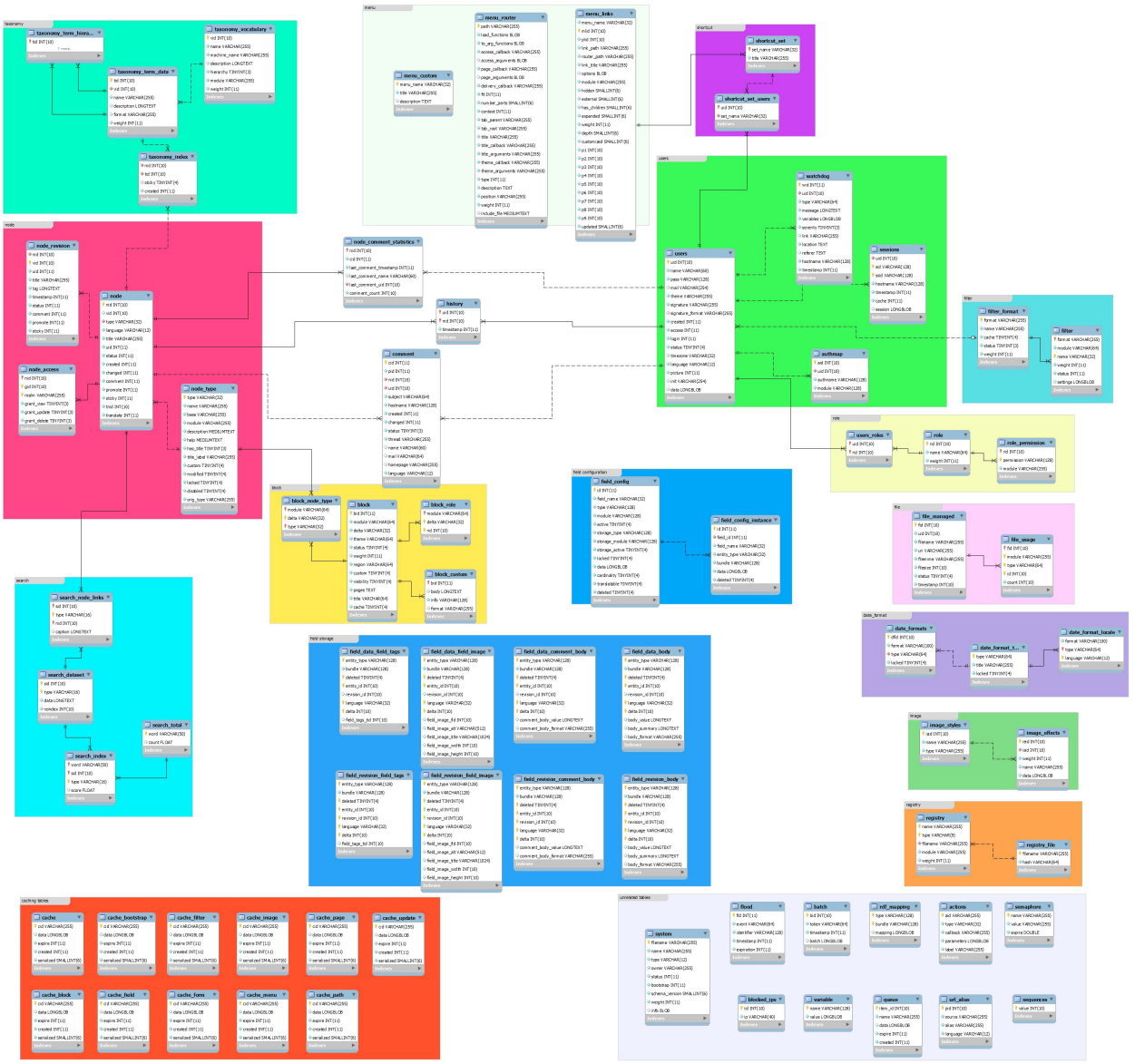

Рисунок 2 — структура БД системы Drupal

Как правило, заказчику не нужно, чтобы витрина постоянно содержала полностью актуальные данные, следовательно мы можем пересчитывать ее по определенному расписанию (в профессиональном сообществе оно называется “регламент”). Кроме того, чтобы минимизировать объем данных в расчетах, как правило, витрина либо строится за один раз лишь на одну отчетную дату (или диапазон дат), например, витрина закупок на определенный день – такие витрины называют “срезными” (они строятся в “разрезе” определенной даты), результирующая таблица наращивается простой дозаписью новых данных. Если бизнес-логика не позволяет организовать такую постройку витрины, например, нам нужна витрина закупок по отделам и типам товаров за все время

наблюдений, то строится т.н. “инкрементальная” витрина. Такая витрина сравнивает версии последних использованных при расчете данных источников с актуальными версиями и производит расчет на основе изменений.

Такой подход (с использованием расписания) позволяет разрабатывать ETL-процессы в виде набора sql-функций или например, spark-скриптов, оркестрируемых каким-либо управляющим механизмом. В теории это могут быть триггеры БД или самостоятельно разрабатываемые скрипты на высокоуровневом языке программирования. Однако, такие инструменты не дают желаемого уровня контроля за успешностью ETL-процессов, их надежностью и т.д. Поэтому, используются либо дорогостоящие BI-инструменты, такие как SAS или Informatica BI. Либо, в сочетании со Spark, СУБД Greenplum и т.п. используется Apache Airflow — открытое программное обеспечение для создания, выполнения, мониторинга и оркестровки потоков операций по обработке данных (см. главу).

1.2. Версионность витрин данных

Подняв тему инкрементальных витрин, мы затронули такой вопрос как версионность данных. Зачастую, клиенту необходимо, чтобы витрина хранила не только актуальные данные, но и историю их изменений (например, для срезных или т.н. “полносрезных” витрин, которые строятся за один запуск ETL-процесса, но данные в них могут меняться из-за изменения источников).

Для хранения истории изменений существует несколько подходов, разработанных в рамках парадигмы SCD (slowly changing dimensions — редко изменяющиеся измерения, то есть измерения, не ключевые атрибуты которых имеют тенденцию со временем изменяться) [3].

Тип 0. Заключается в том, что данные после первого попадания в таблицу далее никогда не изменяются. Этот метод практически никем не используется, т.к. он не поддерживает версионности. Он нужен лишь как нулевая точка отсчета для методологии SCD.

Тип 1. Это обычная перезапись старых данных новыми. В чистом виде этот метод тоже не содержит версионности и используется лишь там, где история фактически не нужна. Тем не менее, в некоторых СУБД для этого типа возможно добавить ограниченную поддержку версионности средствами самой СУБД (например, Flashback query в Oracle) или отслеживанием изменений через триггеры.

Достоинства:

- не добавляется избыточность;
- очень простая структура.

Недостатки:

- не хранит истории.

Тип 2. Данный метод заключается в создании для каждой версии отдельной записи в таблице с добавлением поля-ключевого атрибута данной версии, например: номер версии, дата изменения или дата начала и конца периода существования версии (рисунок 3).

ID	NAME	POSITION_ID	DEPT	DATE_START	DATE_END
1	Коля	21	2	11.08.2010 10:42:25	01.01.9999
2	Денис	23	3	11.08.2010 10:42:25	01.01.9999
3	Борис	26	2	11.08.2010 10:42:25	01.01.9999
4	Шелдон	22	3	11.08.2010 10:42:25	01.01.9999
5	Пенни	25	2	11.08.2010 10:42:25	01.01.9999

Рисунок 3 — таблица с версионностью SCD-2

В этом примере в качестве даты конца версии по умолчанию стоит '01.01.9999', вместо которой можно было бы указать, скажем, null, но тогда возникла бы проблема с созданием первичного ключа из ID,DATE_START и DATE_END, и, кроме того, так упрощается условие выборки для определенной даты (*"where snapshot_date between DATE_START and DATE_END"* вместо *"where snapshot_date>DATE_START and (snapshot_date < DATE_END or DATE_END is null"*).

При такой реализации при увольнении сотрудника можно будет просто изменить дату конца текущей версии на дату увольнения вместо удаления записей о работнике.

Достоинства:

- хранит полную и неограниченную историю версий;
- удобный и простой доступ к данным необходимого периода;

Недостатки:

- провоцирует на избыточность или заведение дополнительных таблиц для хранения изменяемых атрибутов измерения;
- усложняет структуру или добавляет избыточность в случаях, если для аналитики потребуется согласование данных в таблице фактов с конкретными версиями измерения и при этом факт может быть не согласован с текущей для данного факта версией измерения.(Например, у клиента изменились ревизиты или адрес, а нужно провести операцию/доставку по старым значениям);

Тип 3. В самой записи содержатся дополнительные поля для предыдущих значений атрибута. При получении новых данных, старые данные перезаписываются текущими значениями (рисунок 4).

	ID	UPDATE_TIME	LAST_STATE	CURRENT_STATE
1	1	11.08.2010 12:58:48	0	1
2	2	11.08.2010 12:29:16	1	1

Рисунок 4 — таблица с версионностью SCD-3

Достоинства:

- небольшой объем данных;
- простой и быстрый доступ к истории.

Недостатки:

- ограниченная история.

Тип 4. История изменений содержится в отдельной таблице (рисунки 5-6): основная таблица всегда перезаписывается текущими данными с

перенесением старых данных в другую таблицу. Обычно этот тип используют для аудита изменений или создания архивных таблиц (как я уже говорил, в Oracle этот же 4-й тип можно получить из 1-го используя flashback archive).

ID	NAME	POSITION_ID	DEPT
1	Коля	21	2
2	Денис	23	3
3	Борис	26	2
4	Шелдон	22	3
5	Пенни	25	2

Рисунок 5 — таблица с версионностью SCD-4

ID	NAME	POSITION_ID	DEPT	DATE
1	Коля	21	1	11.08.2010 14:12:13
2	Денис	23	2	11.08.2010 14:12:13
3	Борис	26	1	11.08.2010 14:12:13
4	Шелдон	22	2	11.08.2010 14:12:13

Рисунок 6 — таблица с версионностью SCD-4

Достоинства:

– быстрая работа с текущими версиями.

Недостатки:

– разделение единой сущности на разные таблицы.

Гибридный тип/Тип 6(1+2+3). Тип 6 был придуман Ральфом Кимболлом как комбинация вышеназванных методов и предназначен для ситуаций, которые они не учитывают или для большего удобства работы с данными (рисунок 7). Он заключается во внесении дополнительной избыточности: берется за основу тип 2, добавляется суррогатный атрибут для альтернативного обзора версий(тип 3), и перезаписываются одна или все предыдущие версии(тип 1).

VERSION	ID	NAME	POSITION_ID	DEPT	DATE_START	DATE_END	CURRENT
1	1	Коля	21	2	11.08.2010 10:42:25	01.01.9999	1
1	2	Денис	23	3	11.08.2010 10:42:25	01.01.9999	1
1	3	Борис	26	2	11.08.2010 10:42:25	11.08.2010 11:42:25	0
2	3	Борис	26	2	11.08.2010 11:42:26	01.01.9999	1

Рисунок 7 — таблица с версионностью SCD-6

Примечание. На практике, конкретно в витринах данных используются 1, 2 и 6 типы. В хранилище, с которым работаю я, тип 1 реализован на встроенном уровне для любой витрины, т.к. мы работаем с append-only таблицами. Для срезных объектов для каждого PK выбирается строка с максимальным version_id, для удаленных строк она содержит специальный маркер удаления. Для инкрементальных таблиц поверх этого реализуется тип 3, при этом в приемник обычно складываются записи с effective_to = 9999, а сбор консистентной истории изменений реализуется на уровне представления на основе комбинации effective_to, lead(effective_from) и сортировки по verison_id (таблицы 1-2).

Таблица 1 — исходные данные в приемнике витрины

user_id	zarplata_amt	eff_from	eff_to	version_id	deleted_flg
Vasya	45.000	2001	2999	1	[]
Vasya	60.000	2005	2999	2	[]
Vasya	null	2007	2999	3	[v]

Таблица 2 —результирующие данные в представлении витрины

user_id	zarplata_amt	eff_from	eff_to
Vasya	45.000	2001	2004
Vasya	60.000	2005	2006

2. Apache Airflow

2.1. Введение Apache Airflow

В главе 1 было указано, что одним из удобных способов оркестрации ETL-процессов является использование Apache Airflow. Рассмотрим этот инструмент.

Apache Airflow — открытое программное обеспечение для создания, выполнения, мониторинга и оркестровки потоков операций по обработке данных.

По сути, Apache Airflow является набором из python-библиотеки, веб-сервисов и веб-интерфейса для разработки, исполнения и мониторинга сценария выполнения задач. Зависимости между задачами описываются в виде направленного ациклического графа (DAG — directed acyclic graph), задача может представлять из себя, например, bash-скрипт, python-скрипт, обращение к БД на языке SQL, spark-скрипт или пустой оператор (используется для более удобной визуализации сложных зависимостей). Набор операторов может быть расширен пользователем, Airflow очень универсален.

Рассмотрим пример. Допустим, наш пользователь хочет получать отчет и фото с новых запусках космических ракет, используя API thespacedevs.com. DAG из задач, необходимых для реализации такова сценария представлен на рисунке 8 [4].

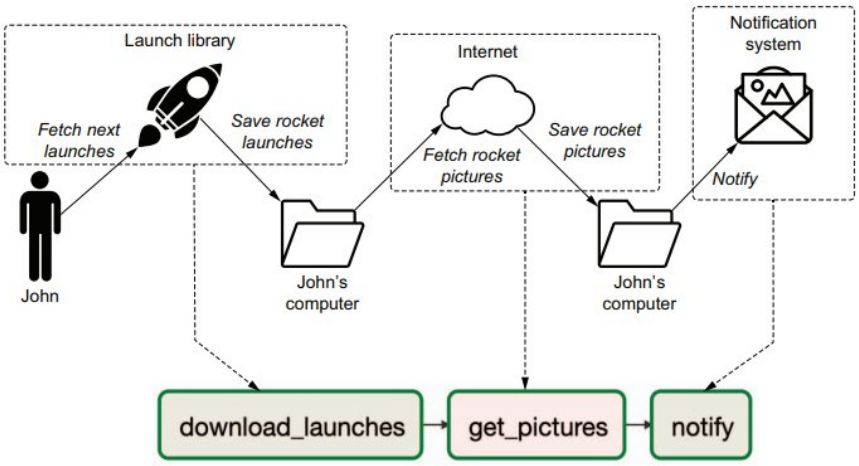

Рисунок 8 — Airflow DAG

Исходный код подобного DAG приведен в листинге 1.

Листинг 1 — исходный код DAG

```
import json
import pathlib
import requests
import requests.exceptions as req_exec
from datetime import datetime


from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator


# отчетная дата
default_args ={
    'start_date': datetime(2021,7, 10)
}



# функция загрузки картинок
def _get_pictures():
    #ensure directory exists
    pathlib.Path('/tmp/images').mkdir(parents=True, exist_ok=True)
    #download all pictures in launches.json
    with open("/tmp/launches.json") as f:
        launches = json.load(f)
        image_urls = [launch["image" ] for launch in launches["results"]]

        for image_url in image_urls:
            try:
                response = requests.get(image_url)
                image_filename = image_url.split("/") [-1]
                target_file = f"/tmp/images/{image_filename}"
                with open(target_file, "wb") as f:
                    f.write(response.content)
                print(f"Downloaded {image_url} to {target_file}")
            except req_exec.MissingSchema:
                print(f"{image_url} appears to be invalid URL.")
            except req_exec.ConnectionError:
                print(f"Could not connect to {image_url}")



with
DAG('download_rocket_launches',schedule_interval=None,default_args=default_args) as
dag:


    # шаг 1. Загрузка списка запусков
    download_launches=BashOperator(
        task_id='download_launches',
```

```
        bash_command="curl -o /tmp/launches.json -L
'https://ll.thespacedevs.com/2.0.0/launch/upcoming/'",
    )


    # шаг 2. Загрузка фото запусков
    get_pictures=PythonOperator(
        task_id='get_pictures',
        python_callable=_get_pictures
    )


    # шаг 3. Сообщение в консоль
    notify=BashOperator(
        task_id='notify',
        bash_command = 'echo "There are now $(ls /tmp/images/ | wc -l) images."'
    )


    # зависимости
    download_launches >> get_pictures >> notify
```

Примеры графического интерфейса Airflow приведены на рисунках 9-12 [5].

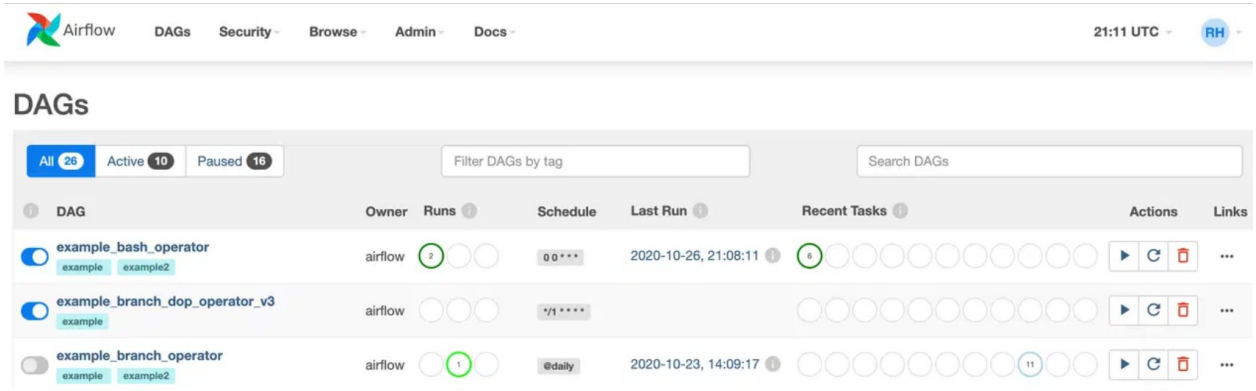

Рисунок 9 — список DAG

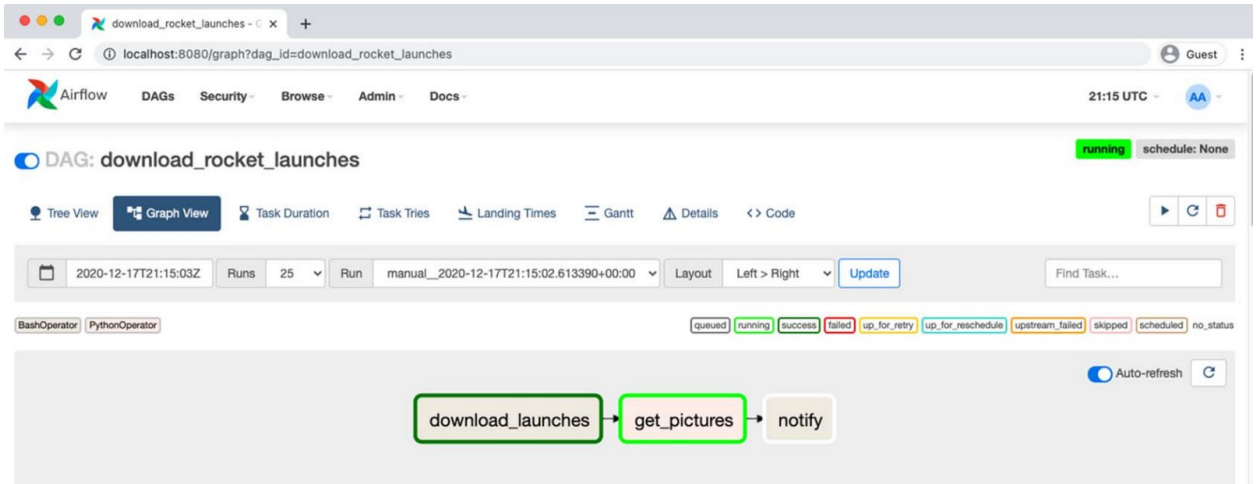

Рисунок 10 — визуализация DAG

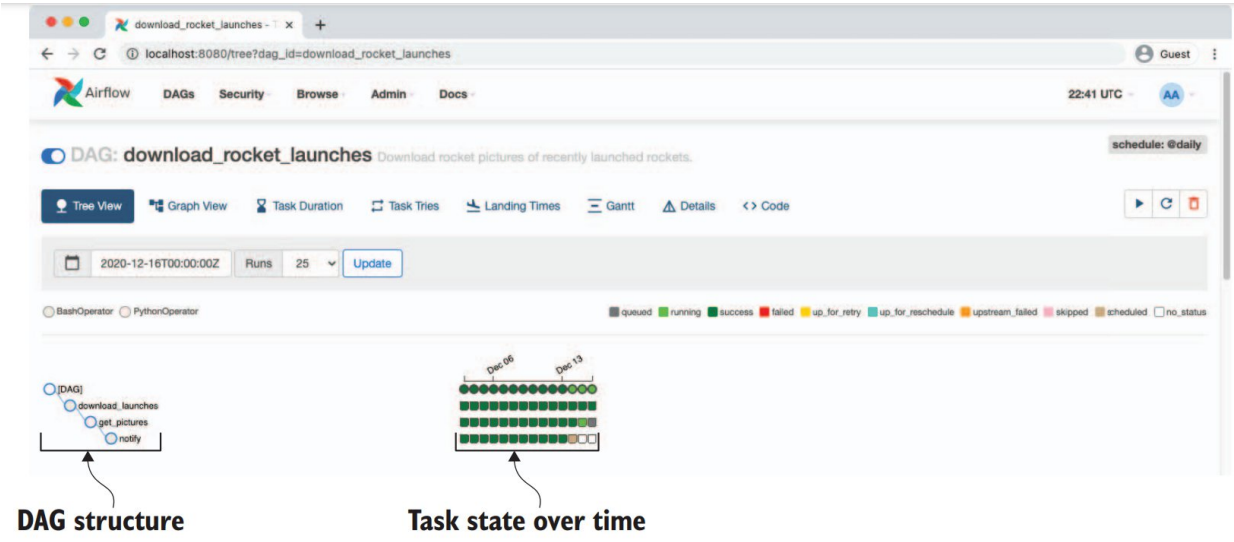

Рисунок 11 — история запусков DAG

```
*** Reading local file: /opt/airflow/logs/download_rocket_launches/notify/2020-12-17T21:15:02.613390+00:00/1.log
[2020-12-17 21:15:30,917] {taskinstance.py:826} INFO - Dependencies all met for <TaskInstance: download_rocket_launches.notify 2020-12-17T2
[2020-12-17 21:15:30,923] {taskinstance.py:826} INFO - Dependencies all met for <TaskInstance: download_rocket_launches.notify 2020-12-17T2
[2020-12-17 21:15:30,923] {taskinstance.py:1017} INFO -
--------------------------------------------------------------------------------
[2020-12-17 21:15:30,923] {taskinstance.py:1018} INFO - Starting attempt 1 of 1
[2020-12-17 21:15:30,923] {taskinstance.py:1019} INFO -
--------------------------------------------------------------------------------
[2020-12-17 21:15:30,931] {taskinstance.py:1038} INFO - Executing <Task(BashOperator): notify> on 2020-12-17T21:15:02.613390+00:00
[2020-12-17 21:15:30,933] {standard_task_runner.py:51} INFO - Started process 1483 to run task
[2020-12-17 21:15:30,937] {standard_task_runner.py:75} INFO - Running: ['airflow', 'tasks', 'run', 'download_rocket_launches', 'notify', '2
[2020-12-17 21:15:30,938] {standard_task_runner.py:76} INFO - Job 6: Subtask notify
[2020-12-17 21:15:30,969] {logging_mixin.py:103} INFO - Running <TaskInstance: download_rocket_launches.notify 2020-12-17T21:15:02.613390+0
[2020-12-17 21:15:30,993] {taskinstance.py:1230} INFO - Exporting the following env vars:
AIRFLOW_CTX_DAG_OWNER=airflow
AIRFLOW_CTX_DAG_ID=download_rocket_launches
AIRFLOW_CTX_TASK_ID=notify
AIRFLOW_CTX_EXECUTION_DATE=2020-12-17T21:15:02.613390+00:00
AIRFLOW_CTX_DAG_RUN_ID=manual__2020-12-17T21:15:02.613390+00:00
[2020-12-17 21:15:30,994] {bash.py:135} INFO - Tmp dir root location:
 /tmp
[2020-12-17 21:15:30,994] {bash.py:158} INFO - Running command: echo "There are now $(ls /tmp/images/ | wc -l) images."
[2020-12-17 21:15:31,002] {bash.py:169} INFO - Output:
[2020-12-17 21:15:31,006] {bash.py:173} INFO - There are now 2 images.
[2020-12-17 21:15:31,006] {bash.py:177} INFO - Command exited with return code 0
[2020-12-17 21:15:31,021] {taskinstance.py:1135} INFO - Marking task as SUCCESS. dag_id=download_rocket_launches, task_id=notify, execution
[2020-12-17 21:15:31,037] {taskinstance.py:1195} INFO - 0 downstream tasks scheduled from follow-on schedule check
[2020-12-17 21:15:31,070] {local_task_job.py:118} INFO - Task exited with return code 0
```

Рисунок 12 — логи DAG

Таким образом, на данном примере мы познакомились с основными сущностями Apache Airflow:

1. DAG’и (DAGs) — ключевая сущность Airflow. Это скрипты на Python, которые описывают логику выполнения задач: какие должны быть выполнены, в каком порядке и как часто.
2. Задача (Task) — описывает, что делать. Например, выборку данных, анализ, запуск других систем. Каждая задача — это экземпляр оператора с определенными параметрами. Допустим, есть DAG для загрузки данных из базы. Можно создать задачу для выполнения оператора, который отправит SQL-запрос для загрузки данных. Она

будет содержать информацию о том, какой SQL-запрос нужно выполнить, когда и в каком контексте.

3. Оператор (Operator) — класс Python, который определяет, что нужно сделать в рамках задачи. Есть операторы для выполнения скриптов Bash, кода Python, SQL-запросов. Например, чтобы выполнить скрипт Python для анализа данных, используют PythonOperator.

2.2. 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 13):

- 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.

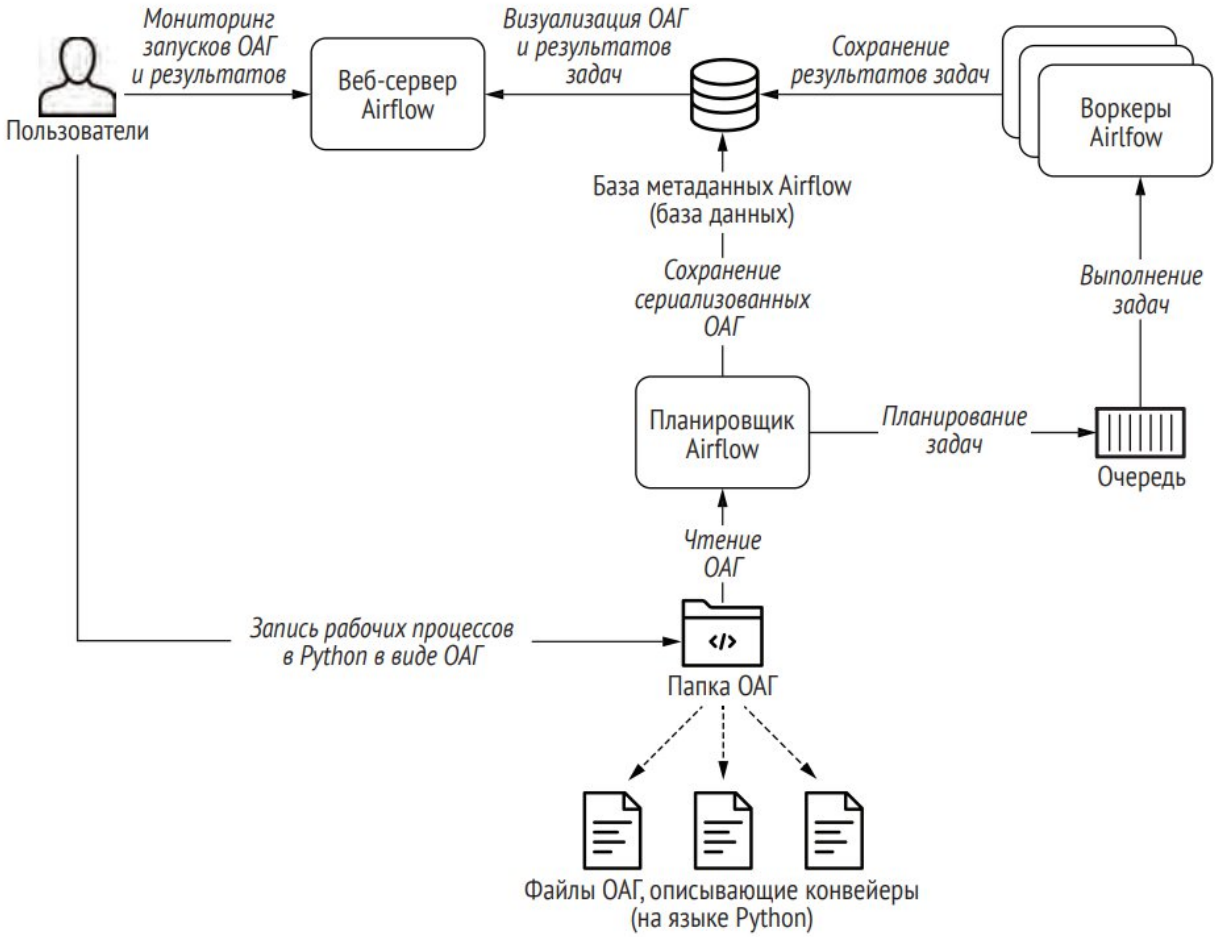

Рисунок 13 — обзор основных компонентов Airflow

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 14):

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.

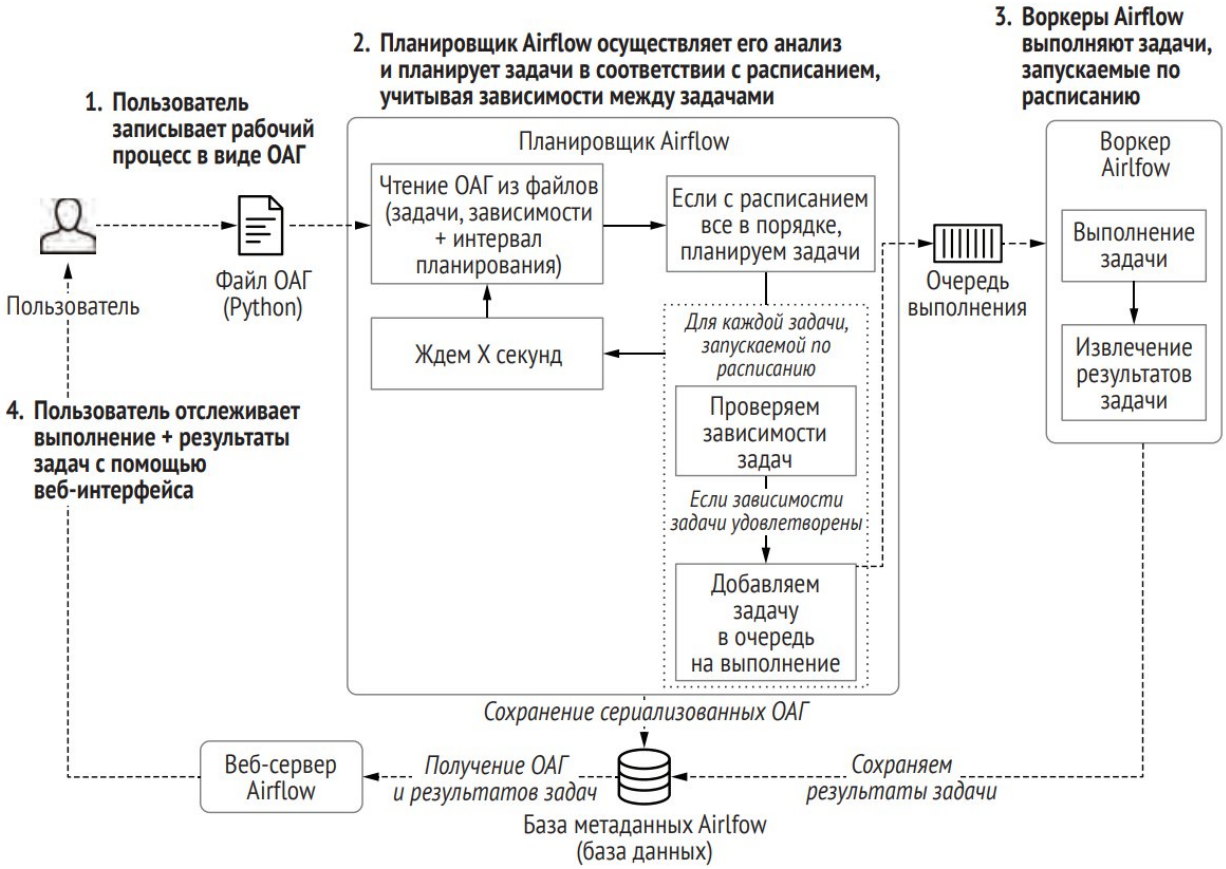

Рисунок 14 — Обзор основных компонентов Airflow (в динамике)

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).

2.3. 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 15-16), позволяющей выполнять новый DAG для интервалов, которые имели место в прошлом [6]. 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.

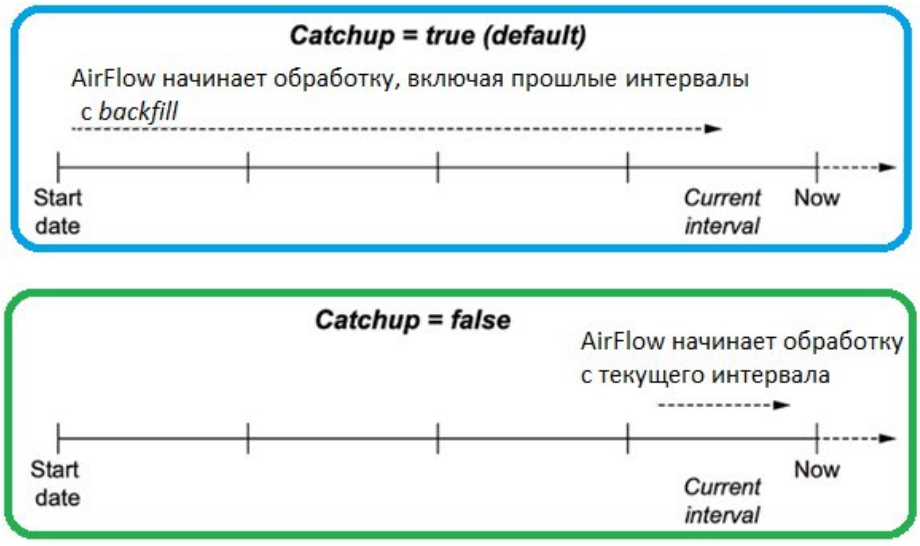

Рисунок 15 — обратное заполнение в Airflow

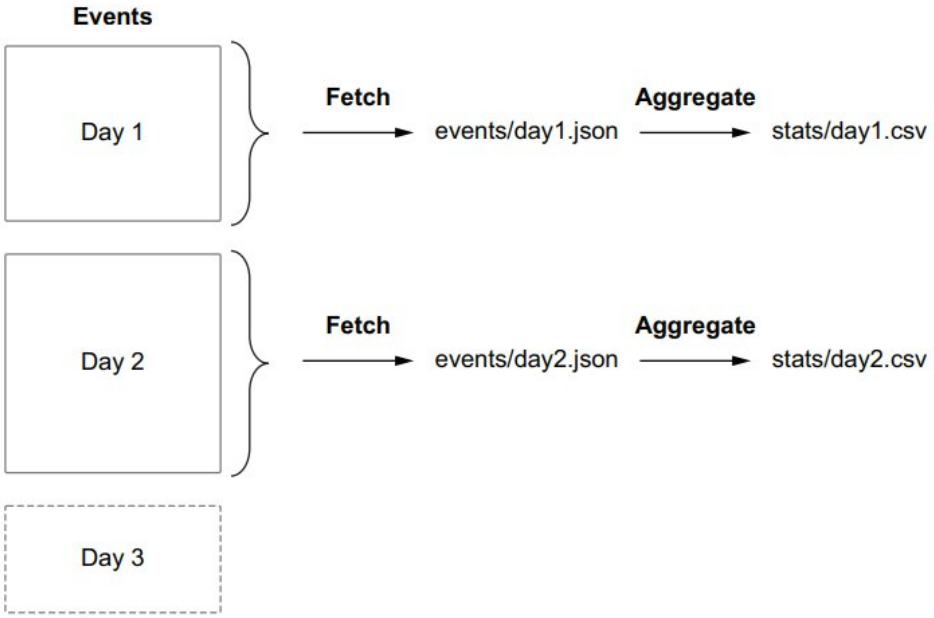

Рисунок 16 — инкрементальная обработка данных

2.4. 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 [7].

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 2), 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 2 — 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 17, листинг 3).

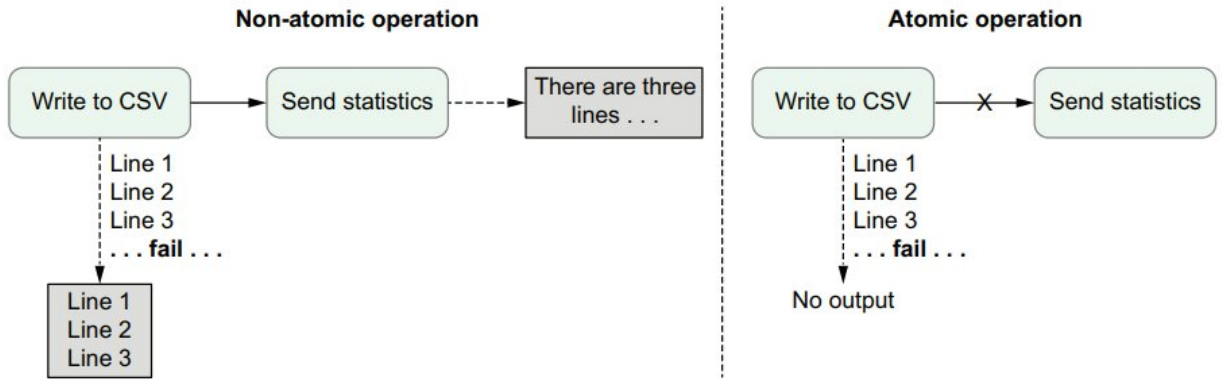

Рисунок 17 — атомарность задач (шагов)

Листинг 3 — 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
```

2.5. 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 18).

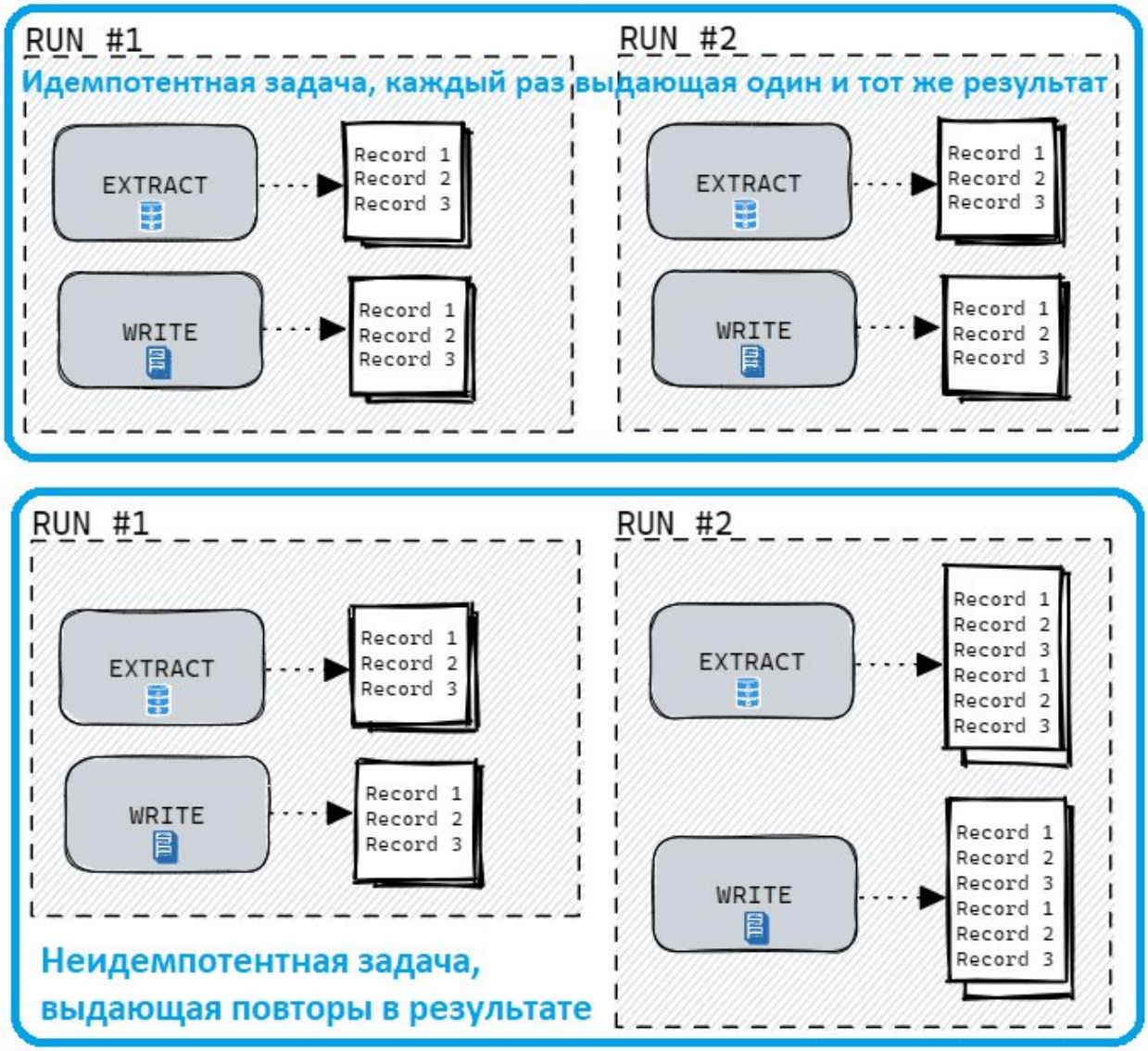

Рисунок 18 — идемпотенстность задач (шагов)

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.

2.6. Ветвление в Apache Airflow.

Зачастую при проектировании ETL-процессов в Airflow необходимо реализовывать ветвления в таких ситуациях, как, например [8]:

- 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 19).

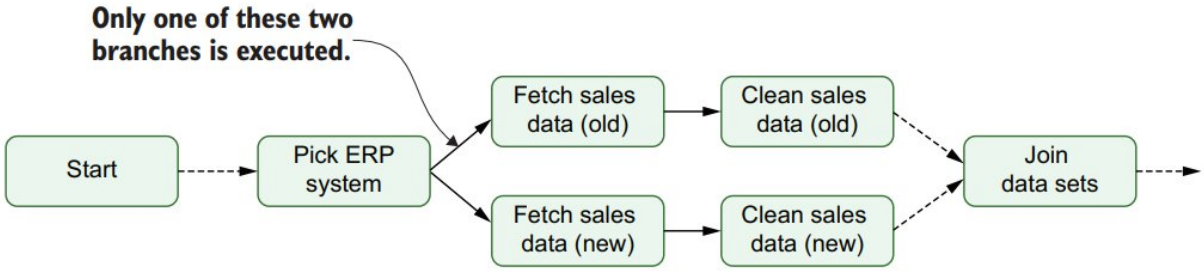

Рисунок 19 — DAG с ветвлением

Реализуется такое ветвление программным кодом, приведенным в листинге 4.

Листинг 4 — 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) [9].

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

Несмотря на растущую популярность таких направлений обработки больших данных, как машинное обучение и нейронные сети, в настоящее время остается актуальной проблема проектирования высоконагруженных хранилищ данных, ETL-процессов и витрин данных.

В данной работе были рассмотрены такие вопросы, как:

- определение витрины данных;
- актуальные технические проблемы построения высоконагруженных витрин данных;
- организация версионного хранения данных;
- использование Apache Airflow в построении ETL-процессов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Простор для данных // habr : сайт. – URL: https://habr.com/ru/articles/650237/ (дата обращения: 15.03.2024)
2. Drupal : сайт. – URL: https://www.drupal.org/node/2360815 (дата обращения: 18.03.2024)
3. Версионность и история данных // habr : сайт. – URL: https://habr.com/ru/articles/101544/ (дата обращения: 21.03.2024)
4. Harenslak, B. Data Pipelines with Apache Airflow / B. Harenslak, J. de Ruiter. – Shelter Island : Manning Publications Co., 2021. – 482 с. – ISBN 9781617296901.
5. Как Apache Airflow помогает дирижировать данными компаний // Яндекс.Практикум : сайт. – URL: https://practicum.yandex.ru/blog/apache-airflow/ (дата обращения: 26.03.2024)
6. 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)
7. Атомарность и идемпотентность в Apache AirFlow // BigDataSchool : сайт. – URL: https://bigdataschool.ru/blog/atomicity-and-idempotency-in-airflow.html (дата обращения: 08.04.2024)
8. 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)
9. 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)