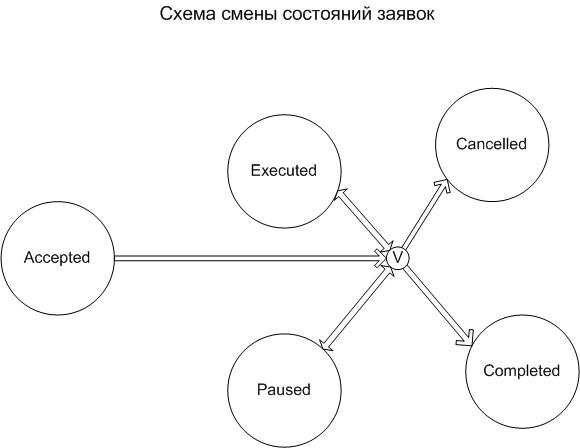
# Формулировка задачи:

Задача состоит продемонстрировать работу по формированию хранилища данных.

В качестве исходных данных берутся синтетически генерируемые данные, для упрощения, одна таблица. В качестве модели используется формирование заявок на обслуживание от пользователей в некой системе ЦДС (центральной диспетчерской службы). Заявки принимаются от пользователей и заносятся в таблицу oltp\_src\_system.request\_data (поля таблицы id – номер заявки, status\_nm – статус заявки, create\_dttm – дата создания заявки, update\_dttm – дата обновления заявки) .

Статус принятой заявки - accepted. Затем заявка в процессе работы может менять статус на 'executed', 'paused', 'complited', 'canselled’. Заявки статусы законченный и отмененный ('complited', 'canselled’ ) – более не меняют.



# Логирование:

Создадим механизм логирования. В схеме log, создадим таблицу log в которую процедура log.write\_log будет заносить данные. Обращу внимание, что в таблице log поле category принимает значения от 0 до 3.

Запись со значением в поле category =

* 0 – всё ок, просто информация.
* 1 – не очень хорошо, но ничего критичного, например, подряд по времени идут записи по одной и той же заявки с одним и тем же статусом.
* 2 – ошибки в данных, таких данных не должно формироваться, например, статус принятой заявки не «accepted»
* 3 – вероятно ошибки в БД, например, из источника данных приходит информация о действиях над записью, которая отсутствует в детальном слое хранилища

В поле type\_operation - вносятся, как и следует из названия, тип операции, это «start» ,«stop» в отношении функций, а так же «insert», «update», «delete» в отношении строк таблиц, «check» - проверка данных , в отношении функционала части процедур.

В процедуре log.write\_log можно сделать фильтрацию, какие данные записывать, какие – нет. Сейчас откидываются записи о нулевых вставка, обновлениях, удалениях. Можно настроить уровни логирования, но это тестовый пример, оставим так.

Создано представление log.v\_show\_log в котором не отображаются события по модифицированному вводу данных (не через механизм cdc, описано далее) и режим отображения, для удобства, в порядке убывания номера события.

# Генерация данных:

Для генерации приема заявок используется процедура oltp\_src\_system.create\_request(), которая формирует некоторое число записей (случайное число до 4-х) в таблицу request\_data, которая будет в дальнейшем источником данных, со статусом «accepted», текущим временем в полях дата создания и дата изменения, с новым номером, больше максимального в базе.

Для изменения статуса заявки используется процедура oltp\_src\_system.update\_existed\_request(), которая случайным образом меняет статус заявок на 'executed', 'paused', 'complited', 'canselled’, кроме имеющих статус , 'complited', 'canselled’. В поле update\_dttm заносится текущее время.

Старые закрытые заявки из таблицы request\_data удаляются процедурой oltp\_src\_system.deleted\_existed\_request(). Выбирается максимально старые записи, имеющие статус 'complited', 'canselled’.

Для загрузки данных в хранилище используется механизм change data capture — захват изменения данных. В схеме oltp\_cdc\_src\_system создана таблица request\_data\_changes (поля : id - номер заявки, status\_nm - статус заявки, create\_dttm - дата создания заявки, update\_dttm - дата обновления заявки ,operation - операция по зявке, stamp - время операции по заявке).

В схеме oltp\_cdc\_src\_system создана тригерная функция request\_data\_changes(). Функция в зависимости от типа операции над строкой в таблице oltp\_src\_system.request\_data добавляет в oltp\_cdc\_src\_system.request\_data\_changes данные с описанием операции над строкой.

# Формирования данных в хранилище:

В схему dwh\_stage данные из таблицы oltp\_cdc\_src\_system.request\_data\_changes

переносятся в таблицу request\_data\_changes\_dwh при помощи процедуры move\_dwh\_stage(). В таблице добавляем поле hash\_key, в которое генерируется хэш на основании полей id и create\_dttm . Считаем данную комбинацию в отношении каждой заявки уникальной, а номер – нет, номер может повторяться. Так же в данную таблицу добавляется поле is\_processed\_flg – по умолчанию False, которое означает, обработана данная запись, для формирования следующего слоя или нет.

В схеме dwh\_ods создана таблица request\_data\_hist, имеющая следующие поля: id – номер заявки, status\_nm - статус заявки, create\_dttm - дата создания заявки, hash\_key – идентификатор заявки, valid\_from\_dttm – дата начала действия версии, valid\_to\_dttm – дата конца действия версии, deleted\_flg – флаг что заявка удалена, deleted\_dttm дата удаления заявки .

Для формирования истории используем поле stamp и не используем поле update\_dttm, так как оно находится в таблице – источнике и может быть не достоверным (изменённым не нами). Заполнение request\_data\_hist производит процедура load\_from\_stage\_hist\_request().

Данная процедура состоит из следующих этапов

1 Создается временная таблица (temp\_stage), в которую копируются необработанные данные с предыдущего слоя( исходная таблица dwh\_stage.request\_data\_changes\_dwh) признак is\_processed\_flg = false)

2 Вставляем в конечную таблицу (dwh\_ods.request\_data\_hist) с временной(temp\_stage) новые заявки (признак: operation = 'I' и такой hash\_key отсутствует в конечной )

3 Вставляем в конечную таблицу записи о новых статусах существующих заявок. Здесь же отрабатываем статус удаленной заявки.

4 Корректируем время действия записей об операциях (valid\_to\_dttm)

5 Теоретически возможно, что нам придут подряд идущие, по времени, записи об одной и той же операцией над заявкой. Поэтому на этом шаге склеиваем подряд идущие интервалы по одним и тем же операциям над одними и теми же заявками.

6 Проставляем в исходной таблице с предыдущего слоя (dwh\_stage.request\_data\_changes\_dwh) флаги обработки у обработанных записей (мы можем это легко сделать так как обработанные записи лежат в временной таблице temp\_stage).

( Более подробно в «5\_Требования к Информационной Системе1\_1» )

Формируем календарь (dwh\_ods.dim\_date) при помощи функции dwh\_ods.load\_dim\_date(). Функция достаточно простая, хочу отметить, что при формирования календаря используются временные таблицы праздничных дней и переносов выходных. Без этих данных, на каждый год производственный календарь будет не точным, а данные утверждаются каждый год, поэтому нужны записи на все года, когда планируется использовать календарь. Функционал полей понятен из названия.

Заполним производственный календарь данными.

Создадим таблицу (dwh\_ods.dim\_work), в которой будем использовать все поля календаря, а так же добавим агрегированные данные по дням:

* количество выполненных заявок в этот день 'complited\_per\_day',
* количество отменных заявок в этот день 'canselled\_per\_day'
* количество принятых заявок в этот день 'accepted\_per\_day'
* количество поставленных на паузу в этот день 'paused\_per\_day '
* количество удаленных заявок в этот день 'deleted\_per\_day '
* по количеству выполняемых заявок на текущий день 'executed\_during\_day'
* находящихся в паузе на текущий день 'paused\_during\_day '

Создадим вспомогательную таблицу dwh\_ods.agg\_request\_data\_hist, в которой все поля с типом timestamp будут заменены на date. Загрузку вспомогательной таблицы выполним функцией dwh\_ods.load\_agg\_request\_data\_hist(). Загрузку таблицы dim\_work выполним функцией dwh\_ods.load\_dim\_work().

# Формирование представлений в витрине данных

В схеме report создадим представление v\_recover\_request\_data, по которому можно получить крайнее актуальное состояние данных в системе источнике, на основании данных хранилища.

В схеме report создадим представление v\_dif\_source\_target, по которому можно получить разницу восстановленными данными на основании источника и реальными данными источника.

В схеме report создадим представление report.v\_dif\_source\_target\_partial, по которому можно получить частичную разницу восстановленными данными на основании источника и реальными данными источника, не учитывая поле update\_dttm.

Создадим представление report. v\_agg\_per\_week в котором в разрезе дней недели сгруппируем агрегированные данные из таблицы dwh\_ods.dim\_work .

Так же создадим представление report.v\_corr в котором разместим матрицу корреляций между агрегированными данными из таблицы dwh\_ods.dim\_work.

# Модифицированая генерация данных (эмуляция генерации cdc)

При заполнении, через airflow мы получим не большое количество данных, всего за пару дней. Если запускать dag в прошлом, то тоже нужного качества данных не получим: генерация данных для слоя cdc содержит функцию now() в триггере. Наличие шаблонов Jinja в airflow ничем нам не поможет, если не менять механизм генерации данных. Менять механизм не будем, просто немного обойдём.

Сделаем модифицированный генератор данных, который обойдёт наш стандартный механизм генерации данных, за 2023 год до текущей даты, положит данные в таблицы oltp\_cdc\_src\_system.request\_data\_changes и oltp\_src\_system.request\_data.

Создадим функции oltp\_src\_system.mod\_create\_request(dt timestamp), oltp\_src\_system.mod\_update\_existed\_request(dt timestamp), oltp\_src\_system.mod\_deleted\_existed\_request(dt timestamp), которые будут генерировать данные для oltp\_src\_system.request\_data и oltp\_cdc\_src\_system.request\_data\_changes на конкретную дату. Останавливаем DAG в airflow. Отключаем на время триггер request\_data\_changes on oltp\_src\_system.request\_data через set session\_replication\_role или просто удалив.

Выполняем

with gen\_dt as

(

select

dt + random() \* (timestamp '2023-01-01 23:59:59' -

timestamp '2023-01-01 00:00:00') as dt

from

generate\_series('2023-01-01 00:00'::timestamp ,

now()::timestamp,

'1 day') dt

)

select

oltp\_src\_system.mod\_create\_request(dt),

oltp\_src\_system.mod\_update\_existed\_request(dt + interval '1 sec'),

oltp\_src\_system.mod\_deleted\_existed\_request(dt + interval '2 sec')

from

gen\_dt

order by dt;

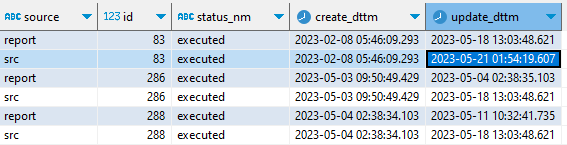
Возвращаем триггер обратно, что бы наша штатная генерация данных, через airflow с механизмом cdc работала корректно. Запускаем DAG в airflow.

Рассмотрим представления в витрине данных:  
Теперь, имея значительное количество данных, за большой промежуток времени рассмотрим отчеты в схеме report.

В представлении report.v\_recover\_request\_data мы видим восстановление данные источника из детального слоя хранилища. Мы можем их сравнить исходными данными с oltp\_src\_system.request\_data. Для сравнения удобно воспользоваться следующим представлением report.v\_dif\_source\_target, здесь уже формируется разница представления и исходной таблицы, объёдиненная с разницей таблицы и представления. В идеале данное представление должно быть пусто, но если данных сгенерировано достаточно много, то скорее всего, будет не так. В результате запроса

**select** \* **from** report.v\_dif\_source\_target;

Вы увидите, нечто вроде:



Данную таблицу следует понимать так: строки с восстановленными данными помечены в поле «source» значением «report», а с исходными данными – «src».

Мы видим, что у некоторых записей не совпадает поле «update\_dttm». Разберемся, почему.

Следующий запрос в лог:

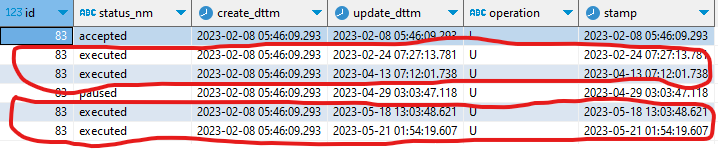
**select** \* **from** **log**.v\_show\_log **where** category >0;

Возвращает:



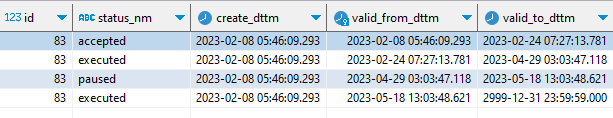
Т.е процедура dwh\_ods.load\_from\_stage\_hist\_request() проводила объединение касающихся интервалов, и всего данная склейка затронула 106 строк в таблице dwh\_ods.request\_data\_hist. Таким образом, проблема возникает, скорее всего, из-за объединения интервалов подряд идущих операций, давайте проверим это на примере операции с номером 83.

**select** \* **from** oltp\_cdc\_src\_system.request\_data\_changes **where** id = 83 **order** **by** stamp;



Мы видим, действительно, подряд идущие операции, которые в слое ods должны быть объединены. Проверим

**select** \* **from** dwh\_ods.request\_data\_hist **where** id = 83 **order** **by** valid\_from\_dttm;

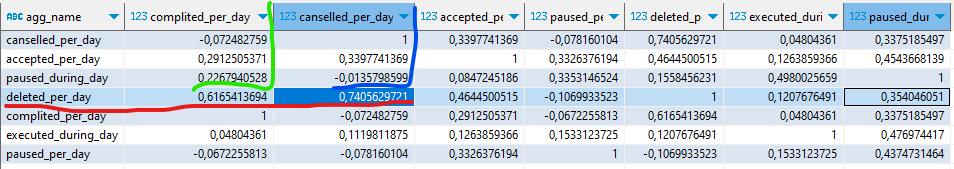


Как видим, всё работает, как и должно работать. Можно сказать ,что в востанновленных данных поле «update\_dttm» правильнее, чем в исходных. Можно ли сделать, что бы update\_dttm совпадало – да, это легко сделать, просто отменить объёдинение интервалов.

Как же сравнивать восстановленные данные и исходные? Для этого создал представление report.v\_dif\_source\_target\_partial , которое производит частичное сравнение данных источника и восстановленного, без поля «update\_dttm». Поскольку у нас данные генерируются и протягиваются одномоментно, то представлении v\_dif\_source\_target\_partial должно быть пустое. Но в реальной жизни, поступление данных и протяжка до хранилища будет всегда разнесена по времени, и восстановленные данные и реальные будут всегда отличаться.

Представление report.v\_agg\_per\_week было бы интересно смотреть на реальной базе, в разрезе дней недели, посмотреть количество принимаемы, закрываемы заявок, заявок поставленных на паузу и т.п..

А вот представление report.v\_corr интересно посмотреть даже на текущих, сгенерированных данных. В представлении построена матрица корреляции между агрегированными показателями истории заявок.



Видна значительная корреляция между отменёнными и удаленными в текущий день, так же между выполненными и удаленными в текущий день. На самом деле здесь нет ничего странного, мы видим случайный процесс, а не набор случайных величин, поэтому корреляция и значительна, ведь удаляются только отмененные и удаленные задачи.