Инструкция по развертыванию стека Postgres-Debezium-Kafka-Clickhouse

Подготовка источника — сервера Postgres

Первое, что нам нужно будет сделать — настроить источник в виде транзакционной базы данных. Чаще всего на предприятиях сейчас используется Postgres, его и установим/настроим (в версии 16).

Будем считать, что у нас есть чистая Ubuntu 24.04. Все действия в операционной системе я буду производить от root (безопасники не одобряют, но для теста так проще).

1. Начинаем с того, что ставим на операционную систему обновления:

apt update

apt upgrade

и перезагружаем систему.

Еще для удобства просмотра каталогов и редактирования файлов конфигурации можно установить Midnight Commander (дело вкуса):

apt install mc

1. Дальше подключаем репозитарий apt.postgresql.org:

apt install -y postgresql-common

/usr/share/postgresql-common/pgdg/apt.postgresql.org.sh

1. Устанавливаем Postgres 16:

apt install postgresql-16

и проверяем, что все работает:

systemctl status postgresql

Служба Postgres должна быть active и enabled.

1. Дальше обычным образом разрешаем удаленные подключения к Postgres. В файле /etc/postgresql/16/main/postgresql.conf меняем строку

#listen\_address = 'localhost'

на

listen\_address = '\*'

(не забудьте снять комментарий!)

В файле /etc/postgresql/16/main/pg\_hba.conf меняем строку

host all all 127.0.0.1/32 scram-sha-256

на

host all all 0.0.0.0/0 scram-sha-256

Далее перезапускаем сервер PostgreSQL, чтобы он воспринял новые настройки:

systemctl restart postgresql

1. Следующая задача — настроить пароль для суперпользователя (администратора) Postgres, чтобы к Postgres можно было подключаться удаленно:

su postgres

psql

и окне приглашения утилиты psql выполняем команду

\password

Затем дважды вводим пароль, который будет назначен специальной роли postgres на сервере PostgreSQL.

1. Далее подключаемся к Postgres из какого-нибудь внешнего средства типа pgAdmin или DBeaver и создаем в нем базу test:

CREATE DATABASE test;

А в ней — простенькую таблицу:

CREATE TABLE test\_table (

id integer NOT NULL,

string\_val text,

PRIMARY KEY (id));

Добавление записей в эту таблицу мы и будем отслеживать и передавать в Clickhouse.

Обратим также внимание, что крайне рекомендуется, чтобы в таблице, которая отслеживается CDC, был первичный ключ. Postgres умеет отслеживать изменения и по таблице без первичного ключа, если выполнить команду вида

ALTER TABLE IF EXISTS public.test\_table REPLICA IDENTITY full;

но это и ресурсоемко, и намного менее удобно, чем использовать первичный ключ.

**Это были общие моменты подготовки Postgres, а дальше будут моменты, специфичные для CDC.**

1. Для того, чтобы превратить изменения в базе, которые записываются в журнал транзакций (в Postgres — Write-Ahead Log, WAL), в JSON для передачи в Debezium-Kafka-Clickhouse, нужно специальное расширение для Postgres, которое называется wal2json. Устанавливаем его командой операционной системы из репозитария:

apt install postgresql-16-wal2json

А дальше записываем его в подключаемые при запуске модули Postgres. Для этого в файле /etc/postgresql/16/main/postgresql.conf находим строку

#shared\_preload\_libraries = ''

раскомментируем ее и приводим в следующий вид:

shared\_preload\_libraries = 'wal2json'

Любые изменения в shared\_preload\_libraries требуют перезапуска Postgres, но мы его сделаем чуть позже.

1. Для Change Data Capture (точнее, подсистемы логической репликации, на которой основана CDC от Debezium), необходимо поднять уровень подробности ведения журнала транзакций на Postgres. Он определяется при помощи параметра конфигурации wal\_level в файле postgresql.conf. По умолчанию для него настроено значение replica (уровень, достаточный для работы в качестве реплики при логической репликации). Нужно снять комментарий и изменить уровень WAL на logical:

wal\_level = logical

1. Следующая, что нужно сделать — это создать в Postgres пользователя (роль), от имени которого будет производиться отслеживание изменений:

CREATE ROLE repluser WITH

LOGIN

NOSUPERUSER

INHERIT

NOCREATEDB

NOCREATEROLE

REPLICATION

NOBYPASSRLS

Обратите внимание на параметр REPLICATION — без него эта роль работать не будет. Назначим для порядка этому пользователю пароль:

ALTER ROLE repluser PASSWORD 'password';

И для наших тестовых целей разрешим аутентификацию без пароля. Для этого первыми незакомментированными строками в файл pg\_hba.conf нужно прописать

local replication repluser trust

host replication repluser 127.0.0.1/32 trust

host replication repluser ::1/128 trust

Мы на всякий случай обеспечили возможность подключения пользователя repluser и через Unix sockets (этот протокол поддерживает только локальные подключения), и по IPv4 (локально, если нужно, чтобы Debezium Connector мог подключаться удаленно, нужно указать его IP), и по IPv6 (то же самое). Возможно, вам потребуется не все.

Перезапускаем сервер Postgres:

systemctl restart postgresql

Дальше выдаем разрешения нашему пользователю repluser право SELECT на таблицу test\_table:

GRANT SELECT ON TABLE public.test\_table TO repluser;

И последнее, что нужно сделать — выбрать таблицы (или, в нашем случае, таблицу), изменения в которой мы будем отслеживать. Для этой цели нужно в базе данных test создать объект публикации:

CREATE PUBLICATION test\_publication

FOR TABLE public.test\_table

WITH (publish = 'insert', publish\_via\_partition\_root = false);

Мы будет отслеживать для простоты только вставку данных. С изменением и удалением данных все сложнее. Общий принцип решения задачи такой: делаем отдельные публикации (и отдельные топики Kafka) на UPDATE и на DELETE и анализируем приходящий в топики JSON на предмет тегов BEFORE (старая версия строки) и AFTER (новая версия строки) — про приходящий JSON будет рассказано чуть позже. А для таблицы назначения в Clickhouse используем движок CollapsingMergeTree.

В результате получается следующая схема: у нас есть три топика Kafka (для INSERT, UPDATE и DELETE). Для каждого топика есть своя таблица на движке Kafka, к которой прицеплен свое материализованное представление. Все три материализованных представления передают итоговый результат в общую целевую таблицу:

1. Работа с публикаций для команд INSERT — самая простая, она будет пошагово рассмотрена в следующих разделах;
2. При работе с публикацией для UPDATE средствами материализованного представления в таблице назначения создаются две строки — для старой версии строки со статусом -1, а для новой версии — со статусом 1. Как мы помним, движок CollapsingMergeTree автоматически удаляет строки с противоположным статусом, и таким образом по результатам UPDATE старая версия строки удаляется, а новая — появляется.
3. При работе с публикацией для DELETE в таблице назначения создается строка со статусом -1, и движок CollapsingMergeTree автоматически удалит ее вместе с ранее занесенной строкой, которую нужно удалить (со статусом 1).

Но для нашего простого примера, который ориентирован только на работу с INSERT, на этом все, на источнике данных в виде сервера Postgres мы подготовку завершили. Следующая наша задача — поднять Kafka.

Развертывание Kafka для передачи изменений в Clickhouse

На сервере Postgres у нас возникают изменения, которые нужно передать в Clickhouse. Debezium Connector эти изменения будет забирать. Но напрямую передавать их в Clickhouse неудобно: вдруг Clickhouse какое-то время будет недоступен, и изменения будут потеряны?

Поэтому обычно для промежуточного хранения данных используется средство работы с очередями сообщений. Debezium Connector передаст в очередь (топик) информацию об изменениях на Postgres, а Clickhouse эти изменения заберет из очереди. Если Clickhouse в течение какого-либо времени будет недоступен, ничего страшного: очередь сообщений будет эти сообщения хранить, пока Clickhouse их не заберет.

Самое распространенное средство для работы с очередями сообщений — это Kafka, поэтому мы его и будем использовать. Хотя на практике можно использовать и другие средства, например, RabbitMQ, интеграция с которым тоже реализована в Clickhouse. Но мы установим и настроим Kafka.

Первым делом поставим Java. Я установлю ту версию, которую использовал в проекте, но вы можете поставить другую, которая у вас принята (главное, чтобы она была совместима с данной версией Kafka).

apt install openjdk-11-jdk -y

Проверяем, что с Java все нормально и у нас работает правильная версия:

java -version

Далее скачиваем Kafka. Я опять-таки использую привычную мне проверенную версию, но вы вполне можете установить и другую:

wget <https://downloads.apache.org/kafka/3.7.2/kafka_2.13-3.7.2.tgz>

После этого Kafka разворачиваем из архива:

tar xvf kafka\_2.13-3.7.2.tgz

И перемещаем в каталог /usr/local/kafka

mv kafka\_2.13-3.7.2 /usr/local/kafka

Для хранения метаданных Kafka использует Zookeeper (как мы помним, для этой же цели Zookeper может использовать и Clickhouse), поэтому нужно настроить параметры Zookeeper как службы.

Создаем и открываем на редактирование файл zookeeper.service:

mcedit /etc/systemd/system/zookeeper.service

И записываем в него следующие параметры:

[Unit]

Description=Apache Zookeeper server

Documentation=http://zookeeper.apache.org

Requires=network.target remote-fs.target

After=network.target remote-fs.target

[Service]

Type=simple

ExecStart=/usr/local/kafka/bin/zookeeper-server-start.sh /usr/local/kafka/config/zookeeper.properties

ExecStop=/usr/local/kafka/bin/zookeeper-server-stop.sh

Restart=on-abnormal

[Install]

WantedBy=multi-user.target

Точно так же создаем и открываем на редактирование файл настроек службы Kafka:

mcedit /etc/systemd/system/kafka.service

И записываем в него:

[Unit]

Description=Apache Kafka Server

Documentation=http://kafka.apache.org/documentation.html

Requires=zookeeper.service

[Service]

Type=simple

Environment="JAVA\_HOME=/usr/lib/jvm/java-11-openjdk-amd64"

ExecStart=/usr/local/kafka/bin/kafka-server-start.sh /usr/local/kafka/config/server.properties

ExecStop=/usr/local/kafka/bin/kafka-server-stop.sh

[Install]

WantedBy=multi-user.target

Дальше перечитываем конфигурацию подсистемы служб в Linux:

systemctl daemon-reload

Переставляем Zookeper и Kafka в режим автозапуска, запускаем их и проверяем, все ли хорошо:

systemctl enable zookeeper

systemctl start zookeeper

systemctl enable kafka

systemctl start kafka

systemctl status zookeeper

systemctl status kafka

Обе службы должны находиться в состоянии Active и Enabled.

На всякий случай можно проверить работоспособность Kafka, создав топик и поместив/прочитав из него сообщения.

Создаем топик:

cd /usr/local/kafka

bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic sampleTopic

Смотрим список топиков на сервере:

bin/kafka-topics.sh --list --bootstrap-server localhost:9092

Открываем консоль для отправки сообщений в этот топик на Kafka:

bin/kafka-console-producer.sh --broker-list localhost:9092 --topic sampleTopic

В открывшемся приглашении что-нибудь пишем и нажимаем Enter.

Дальше можно закрыть kafka-console-producer через Ctrl-C или открыть другое окно терминала. Наше сообщение теперь нужно получить, а для этого используется kafka-console-consumer:

bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic sampleTopic --from-beginning

Если вы получили тот же текст, что и писали, все работает нормально. Через Ctrl-C закрываем консьюмера и приступаем к следующей задаче: развертыванию коннектора Debezium.

Развертывание Debezium Connector для отслеживания изменений в Postgres

Debezium Connector — это программный модуль, который отслеживает изменения в Postgres через его журнал транзакций, и если это изменение нам нужно (произошло в правильной базе данных, в правильной таблице и т.п.), то Debezium заберет это изменение и поместит в топик Kafka.

Debezium Connector у нас будет работать как подключаемый модуль Kafka. Создаем для него каталог и перехожу в него:

mkdir /usr/local/kafka/plugins

cd /usr/local/kafka/plugins

Дальше скачиваю в него Debezium Connector для Postgres:

wget <https://repo1.maven.org/maven2/io/debezium/debezium-connector-postgres/2.6.0.Final/debezium-connector-postgres-2.6.0.Final-plugin.tar.gz>

Извлекаю файлы из архива:

tar xvf debezium-connector-postgres-2.6.0.Final-plugin.tar.gz

И для порядка удаляю архив:

rm debezium-connector-postgres-2.6.0.Final-plugin.tar.gz

Далее указываю Kafka каталог plugins как источник подключаемых модулей. Для этого открываю файл connect-standalone.properties:

mcedit /usr/local/kafka/config/connect-standalone.properties

В самом конце будет строка "#plugin.path=". Снимаем комментарий и дописываем путь к каталогу подключаемых модулей:

plugin.path=/usr/local/kafka/plugins

Дальше нужно настроить параметры подключения к Postgres. Создаем файл

mcedit /usr/local/kafka/config/dbz-test-connector.properties

и прописываем в него параметры подключения:

name=dbz-test-connector

connector.class=io.debezium.connector.postgresql.PostgresConnector

tasks.max=1

plugin.name=pgoutput

database.hostname=localhost

database.port=5432

database.user=repluser

database.password=password

database.dbname =test

database.server.name=pkdc01

publication.name=test\_publication

key.converter=org.apache.kafka.connect.json.JsonConverter

value.converter=org.apache.kafka.connect.json.JsonConverter

key.converter.schemas.enable=false

value.converter.schemas.enable=false

topic.prefix=pkdc01

Мы записали только те параметры, которые требуются для нашей тестовой среды. На самом деле параметров подключения — множество, и они описаны в документации по Debezium Connector для Postgres:

<https://debezium.io/documentation/reference/stable/connectors/postgresql.html>

Далее нам нужно перезапустить Zookeeper и Kafka с новыми параметрами:

systemctl stop zookeeper

systemctl stop kafka

systemctl start zookeeper

systemctl start kafka

и проверить их статус:

systemctl status zookeeper

systemctl status kafka

Дальше переходим в каталог Kafka и запускаем Debezium Connector (лучше это сделать в отдельном окне терминала):

cd /usr/local/kafka/

bin/connect-standalone.sh config/connect-standalone.properties config/dbz-test-connector.properties

Проверяем, что все работает. На сервере Postgres нужно вставить запись в нашу таблицу:

insert into test\_table

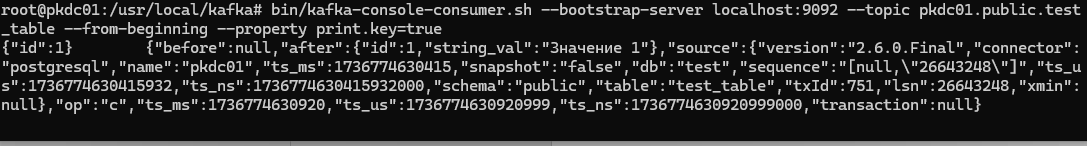
values

(1,'Значение 1')

И проверяем, что данные поступили в Kafka через встроенный консьюмер:

bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic pkdc01.public.test\_table --from-beginning --property print.key=true

Если все сделано правильно, мы должны поймать примерно такое сообщение, которое Debezium Connector отправил в Kafka:



Как мы видим, получился сложный JSON с массой информации об изменениях, но основная информация находится в элементе after.

Следующая задача — настроить Clickhouse для того, чтобы он автоматически ловил такие сообщения и складывал в свои таблицы.

Настройка интеграции Clickhouse и Kafka

Настроить интеграцию Clickhouse и Kafka можно разными способами, но чаще всего используется специальная таблица на движке Kafka в связке с материализованным представлением и обычной таблицей, в которую попадают итоговые данные. Мы тоже используем данный способ.

Но пока Clickhouse на нашем тестовом компьютере еще нет, устанавливаем его (или можно использовать Clickhouse, который уже развернут где-то в сети). Для целостности нашего пилотного проекта все-таки установим Clickhouse локально:

apt-get install -y apt-transport-https ca-certificates dirmngr

apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 8919F6BD2B48D754

echo "deb https://packages.clickhouse.com/deb stable main" | sudo tee /etc/apt/sources.list.d/clickhouse.list

apt update

apt-get install -y clickhouse-server clickhouse-client

А дальше разрешаем удаленные подключения к нашему серверу:

mcedit /etc/clickhouse-server/config.xml

и раскомментируем строку

<listen\_host>0.0.0.0</listen\_host>

Последнее, что нужно сделать — запустить сервер Clickhouse:

systemctl start clickhouse-server

И подключиться к нему каким-нибудь удобным для вас средством выполнения запросов типа DBeaver.

Дальше нам нужно создать последовательно таблицу на движке Kafka, итоговую таблицу и материализованное представление, которое по мере поступления данных в таблицу Kafka будет отправлять данные в итоговую таблицу. Но есть одна проблема: парсер JSON, который читает данные на движке Kafka, по выражению разработчиков, "быстрый и грязный". Он умеет считывать значения из элементов JSON только верхнего уровня. А, как мы видели, Debezium Connector отправляет в Kafka данные об изменениях в виде достаточно сложного JSON, с вложенными элементами. Поэтому нам придется сделать выбор:

1. либо преобразовать JSON в Kafka (например, перекладывая его в другой топик Kafka) при помощи своего программного модуля;
2. либо получать данные об изменениях в Clickhouse как строку и дальше ее парсить, получая нужные значения элементов (например, при помощи функций типа JSONExtract() в определении материализованного представления).

Мы будем использовать второй способ. Но вначале лучше убедиться, что все работает, просто получив JSON об изменениях в Postgres в виде строки.

Создаем таблицу на движке Kafka:

CREATE TABLE kafka\_table (

json\_string String

)

ENGINE = Kafka

SETTINGS kafka\_broker\_list = 'localhost:9092',

kafka\_topic\_list = 'pkdc01.public.test\_table',

kafka\_format = 'JSONAsString',

kafka\_group\_name = 'pkdc.public.test\_table\_consumer\_group1',

kafka\_max\_block\_size = 1048576;

Создаем итоговую таблицу:

CREATE TABLE kafka\_result (

all String,

) Engine = MergeTree

ORDER BY (all);

Настраиваем передачу данных из таблицы на движке Kafka в обычную таблицу MergeTree средствами материализованного представления:

CREATE MATERIALIZED VIEW kafka\_table\_mv TO kafka\_result AS

SELECT json\_string

FROM kafka\_table;

И проверяем, что у нас получилось в итоговой таблице kafka\_result.

Мы должны увидеть что-то вроде

JSON уместился на скриншоте не полностью, но мы его получили. Можно на Postgres добавить еще записей в нашу таблицу test\_table и убедиться, что все приходит в kafka\_result (опрос таблицы на движке Kafka происходит в соответствии с параметром kafka\_poll\_timeout\_ms, по умолчанию 500 миллисекунд).

Осталось совершить последний подвиг: сделать на Clickhouse таблицу с обычными столбцами типа Int и String и получить в них данные из нашего JSON (при желании, конечно, можно извлечь и служебные данные типа временных меток).

Создаем таблицу с обычными столбцами:

CREATE TABLE kafka\_result\_columns (

id UInt64,

string\_val String

) Engine = MergeTree

ORDER BY (id);

--создаем материализованное выражение, которое парсит JSON

CREATE MATERIALIZED VIEW kafka\_table\_mv\_columns TO kafka\_result\_columns AS

SELECT simpleJSONExtractUInt(json\_string, 'id') as id,

simpleJSONExtractString(json\_string, 'string\_val') as string\_val

FROM kafka\_table;

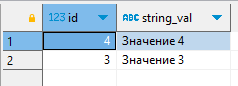
--старое материализованное представление удаляем

drop view kafka\_table\_mv

А дальше на Postgres вставляем в нашу таблицу test\_table еще записи и видим, что они автоматически появились в Clickhouse в таблице kafka\_result\_columns:

select \* from kafka\_result\_columns

Итог должен выглядеть примерно так:



Дополнительные моменты интеграции Clickhouse и Kafka

Мы рассмотрели с вами пошаговую инструкцию по реализации наиболее часто используемой схемы интеграции Clickhouse и Kafka для непрерывной загрузки данных из транзакционного источника, такого, как Postgres. Но нужно посмотреть также дополнительные моменты, которые тоже могут пригодиться в практической работе.

Вначале посмотрим на команду CREATE TABLE… ENGINE = Kafka и прокомментируем некоторые параметры. Напомним, что наш код на создание таблицы на Kafka выглядел так:

CREATE TABLE kafka\_table (

json\_string String

)

ENGINE = Kafka

SETTINGS kafka\_broker\_list = 'localhost:9092',

kafka\_topic\_list = 'pkdc01.public.test\_table',

kafka\_format = 'JSONAsString',

kafka\_group\_name = 'pkdc.public.test\_table\_consumer\_group1',

kafka\_max\_block\_size = 1048576;

Расшифруем эти параметры:

* kafka\_broker\_list — это список серверов Kafka (указывается через запятую). Kafka может работать в отказоустойчивом режиме, поэтому при помощи этого параметра обычно указывается несколько реплик Kafka;
* kafka\_topic\_list — список топиков Kafka. Если вы получаете данные в одну таблицу из нескольких источников, возможность прослушивать несколько топиков может оказаться очень полезной;
* kafka\_format — формат, в котором данные будут приходить из Kafka. Поддерживаемых форматов — на момент написания этого текста (версия Clickhouse 24.12) было 81 штука, и несколько десятков из них имеют отношение к JSON. Полный список можно просмотреть здесь:

<https://clickhouse.com/docs/en/interfaces/formats>

Нам, конечно, придется подстраиваться под формат данных, который отправляет Debezium Connector. Поскольку JSON генерируется сложный, то мной был выбран формат JSONAsString с последующим парсингом. Для плоского JSON (в котором нет вложенных друг в друга элементов) обычно удобнее всего использовать формат JSONEachRow. Все зависит от источника данных, но, как показывает практика, подходящее решение удается найти всегда.

* kafka\_group\_name — этот параметр определяет группу потребителей Kafka. Передача данных из топика Kafka производится на каждую группу потребителей по отдельности. Если у вас, например, две таблицы, которые смотрят на один топик Kafka, то можно поместить каждую из этих таблиц в разные группы. Каждая группа будет извлекать информацию самостоятельно (и все сообщения попадут во все группы);
* kafka\_max\_block\_size — максимальное количество получаемых из Kafka сообщений в рамках одного блока (bacth). Если его не указать, то он будет принят равным значению параметра max\_insert\_block\_size (по умолчанию 1048449).

Есть еще один важный параметр, который в нашей демонстрации указан не был: kafka\_handle\_error\_mode, режим обработки ошибок Kafka. Он будет рассмотрен в отдельном разделе.

Остальные параметры используются редко, посмотреть их можно в документации в разделе

<https://clickhouse.com/docs/en/engines/table-engines/integrations/kafka>

Настройка количества потоков при обработке сообщений из Kafka

При работе с таблицами на движке Kafka (или аналогичными, например, RabbitMQ) сообщений может приходить много, и Clickhouse может просто не хватать потоков, чтобы их обрабатывать. По умолчанию максимальное число потоков, которое можно использовать — 16.

Проверить текущее значение свободных потоков можно при помощи запроса вида

SELECT

(

SELECT value

FROM system.metrics

WHERE metric = 'BackgroundMessageBrokerSchedulePoolTask'

) AS tasks,

(

SELECT value

FROM system.metrics

WHERE metric = 'BackgroundMessageBrokerSchedulePoolSize'

) AS pool\_size,

pool\_size - tasks AS free\_threads

Должно вернуться что-то похожее на



где tasks — количество активных потоков, pool\_size — максимальный размер пула, free\_threads — количество потоков, которое остается в резерве.

Если включена подсистема сбора данных metric\_log (по умолчанию она включена), то можно узнать минимальное количество свободных потоков за текущее число:

SELECT min(CurrentMetric\_BackgroundMessageBrokerSchedulePoolSize - CurrentMetric\_BackgroundMessageBrokerSchedulePoolTask) AS min\_free\_threads

FROM system.metric\_log

WHERE event\_date = today()

Если вы увидели min\_free\_threads равным 0 (или близким к 0), то у вас, вполне возможно, возникали узкие места с точки зрения потоков для обработки данных из таблиц Kafka. Если позволяют ресурсы центрального процессора, то нужно увеличить их количество. Для этого нужно раскомментировать и настроить параметр background\_message\_broker\_schedule\_pool\_size в файле /etc/clickhouse-server/config.xml.

Сбор информации об ошибках при обработке сообщений из таблиц Kafka

В реальной работе в таблицы на движке Kafka вполне может прийти неверно сформированный JSON или не JSON вовсе. В этом случае наше материализованное представление не сможет провести парсинг JSON и возникнет исключение. Но лучше, конечно, такое исключение обработать, ошибку запротоколировать и разобраться, отчего она возникла.

Для того, чтобы обработать ошибку, в Clickhouse предусмотрено специальное средство в виде параметра kafka\_handle\_error\_mode при создании таблицы Kafka. Для него можно использовать два значения:

1. default (оно используется по умолчанию, если этот параметр не указывать) — если обработать данные в формате JSON не удалось, возникнет исключение;
2. stream — исключение будет обрабатываться, а в таблицу на ядре Kafka будут добавлены два новых виртуальных столбца: \_error (с информацией об ошибке) и \_raw\_message (само сообщение из топика Kafka, которое привело к ошибке).

Например, в нашем случае создание таблицы на движке Kafka с обработкой ошибок могло бы выглядеть так:

CREATE TABLE kafka\_table (

json\_string String

)

ENGINE = Kafka

SETTINGS kafka\_broker\_list = 'localhost:9092',

kafka\_topic\_list = 'pkdc01.public.test\_table',

kafka\_format = 'JSONAsString',

kafka\_group\_name = 'pkdc.public.test\_table\_consumer\_group1',

kafka\_max\_block\_size = 1048576,

**kafka\_handle\_error\_mode='stream'**

И к ней можно приделать второе материализованное представление, которое будет использоваться только для сбора ошибок:

CREATE MATERIALIZED VIEW kafka\_errors

(

`topic` String,

`partition` Int64,

`offset` Int64,

`raw` String,

`error` String

)

ENGINE = MergeTree

ORDER BY (topic, partition, offset)

SETTINGS index\_granularity = 8192 AS

SELECT

\_topic AS topic,

\_partition AS partition,

\_offset AS offset,

\_raw\_message AS raw,

\_error AS error

FROM kafka\_table

WHERE length(\_error) > 0

Ничего особенного от целевой таблицы нам не требуется, поэтому положимся на ту таблицу, которую создаст материализованное представление в автоматическом режиме. А просмотр запротоколированных ошибок можно производить через само материализованное представление:

select \* from kafka\_errors

Еще одна возможность диагностики проблем — включить отладочный лог для библиотеки подключения к Kafka (librdkafka). Для этой цели нужно прописать в раздел <kafka> в файле конфигурации /etc/clickhouse-server/config.xml включение максимального уровня отладки, примерно таким образом:

<kafka>

…

<debug>all</debug>

</kafka>

Вся диагностическая информация будет записываться в файл stderr.log.