

Расширяемость Асинхронная обработка



Авторские права

© Postgres Professional, 2017–2024

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

Использование материалов курса

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

edu@postgrespro.ru

Отказ от ответственности

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

Зачем нужна асинхронная обработка данных

Доступные решения

Реализация очереди средствами PostgreSQL

Разнесение во времени возникновения события и его обработки

Соображения производительности

клиенту не требуется ждать ответа

возможность управлять ресурсами для обработки

Реализация

очередь сообщений

более сложный вариант: модель публикация — подписка

Идея асинхронной обработки событий состоит в том, что возникновение события и его обработка разносятся во времени.

Например, пользователь хочет получить детализацию расходов на мобильную связь. Детализация формируется несколько минут. Можно показать пользователю «песочные часы» и заставить его ждать (синхронная обработка), а можно выслать детализацию по электронной почте, когда она будет готова (асинхронная обработка).

Другой пример: интеграция двух систем. Первая система обращается ко второй, передавая пакет сообщений. Обработка одного сообщения занимает несколько секунд, но в пакете может оказаться тысяча сообщений. Можно заставить первую систему ожидать получения результата (синхронная обработка), а можно ответить «работаем», обработать сообщения асинхронно и уже затем сообщить результат.

Асинхронная обработка сложнее синхронной, но часто оказывается очень удобной. Она позволяет работать эффективнее (клиенту не надо простаивать, дожидаясь ответа) и управлять ресурсами (обрабатывать события с удобной скоростью и в удобное время, а не немедленно).

(Асинхронная обработка широко применяется и в ядре PostgreSQL. Вспомните режим асинхронной фиксации; процесс контрольной точки; процесс автоочистки.)

Обычная реализация состоит в наличии очереди событий (сообщений): одни процессы создают события, другие — обрабатывают. Возможны более сложные модели, в которых есть возможность публиковать события и подписываться на события нужного типа.

RabbitMQ, ActiveMQ, ZeroMQ и т. п.

Плюсы

- эти системы работают
- следование стандартам (AMQP, JMS, STOMP, MQTT...)
- гибкость, масштабирование, производительность

Возможные минусы

- отдельная система (включающая отдельную СУБД)
- со своими особенностями настройки, администрирования, мониторинга
- все сложности построения распределенных систем (отсутствие глобальных транзакций)

Одним из вариантов реализации очередей событий являются внешние системы. Названия многих из них традиционно заканчиваются на MQ — Message Queuing.

Как правило, это большие серьезные системы, обеспечивающие гибкость, масштабируемость, высокую производительность и прочие полезные свойства. К тому же они реализуют один или несколько стандартных протоколов работы с сообщениями, что позволяет интегрировать их с другими системами, понимающими те же протоколы.

Но надо понимать, что любая большая система потребует серьезных затрат на ее изучение и внедрение. Потребуется разобраться с особенностями настройки, администрирования, мониторинга. Заметим, что в состав систем работы с очередями входит и отдельная СУБД для надежного хранения очередей.

Кроме того, использование внешней системы приводит ко всем сложностям построения распределенных систем. При отсутствии глобальных транзакций, объединяющих разные системы, возможны случаи потери сообщений в результате сбоев.

Очередь внутри базы: PgQ



Плюсы

система давно на рынке и широко используется

Возможные минусы

мало гибкости, например, исключительно пакетная обработка

плохо документирована

внешняя программа-демон

5

Более простым решением может служить реализация очереди в самой СУБД. Особенно это имеет смысл, если события возникают и обрабатываются на сервере баз данных.

Наиболее известна система PgQ, разработанная в свое время компанией Skype (<https://github.com/pgq>). Эта система достаточно широко используется и про нее известно, что она работает. Если требуется готовое решение, то ей можно и воспользоваться. Поддерживаются версии PostgreSQL 10+.

Из минусов этого решения отметим:

- Недостаточную гибкость. Например, возможна только пакетная обработка событий. Пока обработчик не пометит пакет, как полностью обработанный, все события пакета могут быть доставлены повторно в случае сбоя.
- Отсутствие качественной документации (есть описание API: <https://pgq.github.io/extension/pgq/>).
- Необходимость во внешней (относительно СУБД) программе, обеспечивающей работу очереди.

Очередь внутри базы: pgmq

Плюсы

- система активно развивается
- легковесная реализация
- все компоненты внутри базы

Возможные минусы

- недостаточная гибкость в управлении на низком уровне
- слабо документирована
- только исходные коды, необходимость сборки

6

Решение заявлено компанией Tembo как легковесная реализация очередей. Расширение pgmq написано на Rust и использует инфраструктуру сборки расширений pgrx.

Очереди и метаданные представлены таблицами в базе данных, а элементы очереди («сообщения») — записями в таблицах.

Сообщения можно помещать в очередь (в том числе по несколько сразу), читать из очереди, удалять и архивировать. Для выполнения этих основных действий (и еще ряда дополнительных) в расширении предусмотрен набор функций. Детали реализации очереди скрыты от пользователя, что упрощает работу.

Из минусов pgmq отметим следующие:

- Недостаточная гибкость в управлении на низком уровне. Связано с тем, что расширение pgmq базируется на фреймворке pgrx, в который встроено, в том числе, управление соединениями с сервером СУБД. Повлиять на него средствами уже собранного pgmq нельзя.
- Документация недостаточно подробна.
- В репозитории отсутствуют готовые пакеты, а сборка расширения может оказаться нетривиальной и ресурсоемкой.

<https://tembo.io/pgmq/>

<https://github.com/tembo-io/pgmq>

Очередь средствами расширения pgmq

Создадим базу данных и подключимся к ней:

```
=> CREATE DATABASE ext_async;
```

```
CREATE DATABASE
```

```
=> \c ext_async
```

You are now connected to database "ext_async" as user "student".

Расширение pgmq уже собрано и доступно для установки. Выполним команду создания расширения в нашей базе. Все его объекты будут размещены в схеме pgmq:

```
=> CREATE EXTENSION pgmq;
```

```
CREATE EXTENSION
```

Создадим очередь под названием pgmq_queue:

```
=> SELECT pgmq.create('pgmq_queue');
```

```
create
```

```
-----
```

```
(1 row)
```

Информация об очередях хранится в таблице meta; посмотреть очереди можно с помощью табличной функции:

```
=> SELECT * FROM pgmq.list_queues();
```

queue_name	created_at	is_partitioned	is_unlogged
pgmq_queue	2025-06-24 05:12:31.264031+03	f	f

```
(1 row)
```

Также были созданы таблицы для сообщений очереди: основная q_pgmq_queue и архивная a_pgmq_queue:

```
=> \dt pgmq.*
```

List of relations			
Schema	Name	Type	Owner
pgmq	a_pgmq_queue	table	student
pgmq	meta	table	student
pgmq	q_pgmq_queue	table	student

```
(3 rows)
```

Поместим в очередь несколько сообщений (полезная информация представляется значением типа jsonb)...

```
=> SELECT pgmq.send('pgmq_queue', to_jsonb(i))
FROM (
  VALUES ('alpha'), ('beta'), ('gamma')
) AS v(i);
```

```
send
```

```
-----
```

```
1
```

```
2
```

```
3
```

```
(3 rows)
```

...и заглянем в основную таблицу очереди:

```
=> SELECT msg_id, enqueued_at, message
FROM pgmq.q_pgmq_queue
ORDER BY msg_id;
```

msg_id	enqueued_at	message
1	2025-06-24 05:12:31.549414+03	"alpha"
2	2025-06-24 05:12:31.549414+03	"beta"
3	2025-06-24 05:12:31.549414+03	"gamma"

```
(3 rows)
```

Простой способ забрать сообщение из очереди — вызвать функцию pop:

```
=> SELECT msg_id, enqueued_at, message
FROM pgmq.pop('pgmq_queue');

 msg_id |          enqueued_at          | message
-----+-----+-----
      1 | 2025-06-24 05:12:31.549414+03 | "alpha"
(1 row)
```

Другие обработчики тоже могут брать сообщения:

```
student$ psql -d ext_async

=> SELECT msg_id, enqueued_at, message
FROM pgmq.pop('pgmq_queue');

 msg_id |          enqueued_at          | message
-----+-----+-----
      2 | 2025-06-24 05:12:31.549414+03 | "beta"
(1 row)
```

```
student$ psql -d ext_async

=> SELECT msg_id, enqueued_at, message
FROM pgmq.pop('pgmq_queue');

 msg_id |          enqueued_at          | message
-----+-----+-----
      3 | 2025-06-24 05:12:31.549414+03 | "gamma"
(1 row)
```

А первый при очередном обращении обнаружит, что очередь пуста:

```
=> SELECT msg_id, enqueued_at, message
FROM pgmq.pop('pgmq_queue');

 msg_id | enqueued_at | message
-----+-----+-----
(0 rows)
```

И напоследок удалим саму очередь. При этом ее основная и архивная таблицы исчезнут, как и информация в таблице meta:

```
=> SELECT pgmq.drop_queue('pgmq_queue');

drop_queue
-----
t
(1 row)
```

```
=> \dt pgmq.*

      List of relations
Schema | Name | Type  | Owner
-----+-----+-----+-----
pgmq   | meta | table | student
(1 row)
```

Возможные плюсы

не требуются внешние зависимости
простые требования — простая реализация

Минусы

требуется отладка и тестирование
при усложнении требований готовая система может обойтись дешевле

Для решения простой задачи, требующей асинхронной обработки, использование сторонних систем может оказаться невыгодным. Возможно, проще написать собственную реализацию, чем приспособливаться к особенностям сторонней системы.

Конечно, нужно понимать, что:

- реализация должна быть сделана аккуратно, иначе она может привести к проблемам эксплуатации;
- если к системе очередей предъявляются серьезные требования (или есть шанс, что такие требования появятся в будущем), то развитие, тестирование и поддержка собственного решения, наоборот, может оказаться невыгодной.

Далее мы посмотрим, как реализовать очередь сообщений в PostgreSQL своими руками, и какие подводные камни есть на этом пути.

Реализация очереди сообщений

Наша задача: реализовать простую очередь сообщений с возможностью конкурентного получения сообщений из нескольких процессов. Полезную информацию снова представим типом JSON — так очередь будет достаточно универсальна.

В каждый конкретный момент времени в таблице сообщений не будет много строк, но за все время работы их может оказаться существенное количество. Поэтому идентификатор надо сразу сделать 64-разрядным:

```
=> CREATE TABLE msg_queue(  
    id bigint PRIMARY KEY GENERATED ALWAYS AS IDENTITY,  
    payload jsonb NOT NULL,  
    pid integer DEFAULT NULL -- процесс-обработчик  
);
```

CREATE TABLE

Вставка сообщений в очередь проста:

```
=> INSERT INTO msg_queue(payload)  
VALUES  
    (to_jsonb(1)),  
    (to_jsonb(2)),  
    (to_jsonb(3));
```

INSERT 0 3

Теперь займемся функцией получения и блокирования очередного сообщения.

Нам требуется заблокировать полученную строку, чтобы одно сообщение не могло быть выбрано два раза (двумя одновременно работающими обработчиками). Это можно сделать с помощью фразы FOR UPDATE:

```
=> BEGIN;
```

BEGIN

```
=> SELECT * FROM msg_queue  
WHERE pid IS NULL -- никем не обрабатывается  
ORDER BY id LIMIT 1 -- одно в порядке поступления  
FOR UPDATE;
```

```
id | payload | pid  
----+-----+-----  
1  | 1       |  
(1 row)
```

Но в таком случае аналогичный запрос в другом процессе будет заблокирован до завершения первой транзакции.

```
| => \c ext_async
```

```
| You are now connected to database "ext_async" as user "student".
```

```
| => BEGIN;
```

```
| BEGIN
```

```
| => SELECT * FROM msg_queue  
| WHERE pid IS NULL  
| ORDER BY id LIMIT 1  
| FOR UPDATE;
```

Вторая транзакция заблокирована.

```
=> DELETE FROM msg_queue  
WHERE id = 1;
```

DELETE 1

```
=> COMMIT;
```

COMMIT

```
| id | payload | pid  
|----+-----+-----  
| 2  | 2       |  
| (1 row)
```

```
| => COMMIT;  
| COMMIT
```

Для того чтобы не останавливаться на заблокированных строках, служит фраза SKIP LOCKED команды SELECT.

```
=> BEGIN;
```

```
BEGIN
```

```
=> SELECT * FROM msg_queue  
WHERE pid IS NULL  
ORDER BY id LIMIT 1  
FOR UPDATE SKIP LOCKED;
```

```
id | payload | pid  
----+-----+----  
2  | 2       |  
(1 row)
```

```
| => BEGIN;
```

```
| BEGIN
```

```
| => SELECT * FROM msg_queue  
| WHERE pid IS NULL  
| ORDER BY id LIMIT 1  
| FOR UPDATE SKIP LOCKED;
```

```
| id | payload | pid  
| ---+-----+----  
| 3  | 3       |  
| (1 row)
```

```
=> COMMIT;
```

```
COMMIT
```

```
| => COMMIT;
```

```
| COMMIT
```

Итак, функция для получения и блокирования очередного сообщения может выглядеть следующим образом:

```
=> CREATE FUNCTION take_message(OUT msg msg_queue)  
LANGUAGE sql VOLATILE  
BEGIN ATOMIC  
    UPDATE msg_queue  
    SET pid = pg_backend_pid()  
    WHERE id = (SELECT id FROM msg_queue  
                WHERE pid IS NULL  
                ORDER BY id LIMIT 1  
                FOR UPDATE SKIP LOCKED) RETURNING *;  
END;
```

```
CREATE FUNCTION
```

В практических заданиях к темам «Очистка» и «Фоновые задания» мы рассматривали типичное решение для получения пакета строк таблицы, например, с целью обновления или удаления. Запрос выглядел так:

```
WITH batch AS (  
    SELECT * FROM t  
    WHERE /* необходимые условия */  
    LIMIT /* размер пакета */  
    FOR UPDATE SKIP LOCKED  
)  
...
```

Как видите, в обоих случаях используется тот же самый подход: выбирается и блокируется часть строк (одна или несколько), при этом уже заблокированные строки пропускаются.

Теперь напишем функцию завершения работы с сообщением. Мы будем просто удалять его из очереди.

```
=> CREATE FUNCTION complete_message(msg msg_queue) RETURNS void  
LANGUAGE sql VOLATILE  
BEGIN ATOMIC  
    DELETE FROM msg_queue WHERE id = msg.id;  
END;
```

```
CREATE FUNCTION
```

Теперь мы готовы написать цикл обработки сообщений. Оформим его в виде процедуры.

```
=> CREATE PROCEDURE process_queue() AS $$
DECLARE
    msg msg_queue;
BEGIN
    LOOP
        SELECT * INTO msg FROM take_message();
        EXIT WHEN msg.id IS NULL;

        -- обработка
        PERFORM pg_sleep(1);
        RAISE NOTICE '[%] processed %; n_tup_del=%, backend_xmin=%',
            pg_backend_pid(),
            msg.payload,

            (SELECT n_tup_del FROM pg_stat_xact_all_tables -- статистика, накопленная внутри транзакции
             WHERE relname = 'msg_queue'),

            (SELECT backend_xmin FROM pg_stat_activity
             WHERE pid = pg_backend_pid());

        PERFORM complete_message(msg);
    END LOOP;
END;
$$ LANGUAGE plpgsql;

CREATE PROCEDURE
```

В этом варианте цикл заканчивается, когда в очереди не остается необработанных сообщений. Вместо этого можно не прекращать цикл, но продолжать ожидать новые события, засыпая, например, на одну секунду.

Пробуем.

```
=> CALL process_queue();

NOTICE: [32641] processed 2; n_tup_del=0, backend_xmin=833
NOTICE: [32641] processed 3; n_tup_del=1, backend_xmin=833
CALL
```

Теперь в два потока.

```
=> INSERT INTO msg_queue(payload)
SELECT to_jsonb(id) FROM generate_series(1,10) id;

INSERT 0 10

=> \timing on

Timing is on.

=> CALL process_queue();

| => CALL process_queue();
|
| NOTICE: [33379] processed 2; n_tup_del=0, backend_xmin=835
| NOTICE: [33379] processed 4; n_tup_del=1, backend_xmin=835
| NOTICE: [33379] processed 6; n_tup_del=2, backend_xmin=835
| NOTICE: [33379] processed 8; n_tup_del=3, backend_xmin=835
| NOTICE: [33379] processed 10; n_tup_del=4, backend_xmin=835
| CALL
```

```
NOTICE: [32641] processed 1; n_tup_del=0, backend_xmin=835
NOTICE: [32641] processed 3; n_tup_del=1, backend_xmin=835
NOTICE: [32641] processed 5; n_tup_del=2, backend_xmin=835
NOTICE: [32641] processed 7; n_tup_del=3, backend_xmin=835
NOTICE: [32641] processed 9; n_tup_del=4, backend_xmin=835
CALL
Time: 5042,337 ms (00:05,042)
```

```
=> \timing off
```

Timing is off.

Обработка 10 сообщений двумя потоками заняла около 5 секунд, но горизонт транзакций держался на одном уровне все время обработки очереди! Это будет мешать выполнению очистки и создавать проблемы для всей базы данных.

Что получилось: одна большая транзакция

```
take_message();  
-- обработка  
complete_message();  
  
take_message();  
-- обработка  
complete_message();  
  
take_message();  
-- обработка  
complete_message();  
  
COMMIT;
```

Показанное решение имеет существенный недостаток: вся обработка выполняется в одной длинной транзакции. Вспоминая темы «Многоверсионность» и «Очистка» модуля «Архитектура», можно с уверенностью сказать, что обработка очереди будет мешать нормальной работе очистки.

Что надо: каждое событие в отдельной транзакции

```
take_message();  
-- обработка  
complete_message();  
  
COMMIT;
```

```
take_message();  
-- обработка  
complete_message();  
  
COMMIT;
```

```
take_message();  
-- обработка  
complete_message();  
  
COMMIT;
```

Чтобы таких проблем не возникало, надо раздробить длинную транзакцию на несколько более коротких. В нашем случае — обрабатывать каждое событие в собственной транзакции.

Еще лучше: позволить обработке события состоять из нескольких транзакций

```
take_message();  
COMMIT;
```

```
-- обработка  
complete_message();  
COMMIT;
```

```
take_message();  
COMMIT;
```

```
-- обработка  
COMMIT;
```

```
-- обработка  
COMMIT;
```

```
-- обработка  
complete_message();  
COMMIT;
```

```
take_message();  
COMMIT;
```

```
-- обработка  
COMMIT;
```

```
-- обработка  
complete_message();  
COMMIT;
```

Более того, обработка одного события тоже может (в принципе) разбиваться на несколько транзакций.

В таком случае мы сначала фиксируем изменение статуса события в очереди («в работе»), затем выполняем обработку, а в конце фиксируем факт завершения работы с событием (например, удаляем его из таблицы).

Учитываем горизонт транзакций

Это легко сделать, поскольку процедура позволяет управлять транзакциями.

```
=> CREATE OR REPLACE PROCEDURE process_queue() AS $$
DECLARE
    msg msg_queue;
BEGIN
    LOOP
        SELECT * INTO msg FROM take_message();
        COMMIT; --<<
        EXIT WHEN msg.id IS NULL;

        -- обработка
        PERFORM pg_sleep(1);
        RAISE NOTICE '[%] processed %; n_dead_tup=%, n_tup_del=%, backend_xmin=%',
            pg_backend_pid(),
            msg.payload,

            (SELECT n_dead_tup FROM pg_stat_all_tables -- статистика, учитываемая автоочисткой
             WHERE relname = 'msg_queue'),

            (SELECT n_tup_del FROM pg_stat_xact_all_tables
             WHERE relname = 'msg_queue'),

            (SELECT backend_xmin FROM pg_stat_activity
             WHERE pid = pg_backend_pid());

        PERFORM complete_message(msg);
        COMMIT; --<<
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

CREATE PROCEDURE

Проверим:

```
=> INSERT INTO msg_queue(payload)
SELECT to_jsonb(id) FROM generate_series(1,5) id;
```

INSERT 0 5

```
=> CALL process_queue();
```

```
NOTICE: [32641] processed 1; n_dead_tup=0, n_tup_del=0, backend_xmin=840
NOTICE: [32641] processed 2; n_dead_tup=0, n_tup_del=1, backend_xmin=842
NOTICE: [32641] processed 3; n_dead_tup=0, n_tup_del=2, backend_xmin=844
NOTICE: [32641] processed 4; n_dead_tup=0, n_tup_del=3, backend_xmin=846
NOTICE: [32641] processed 5; n_dead_tup=0, n_tup_del=4, backend_xmin=848
CALL
```

Теперь горизонт транзакций продвигается вперед и не мешает очистке. Однако момент срабатывания автоматической очистки определяется на основе данных статистики, которая обновляется лишь по окончании работы всего оператора CALL.

```
=> SELECT n_dead_tup, n_live_tup, n_mod_since_analyze, n_ins_since_vacuum
FROM pg_stat_all_tables
WHERE relname = 'msg_queue';
```

```
 n_dead_tup | n_live_tup | n_mod_since_analyze | n_ins_since_vacuum
-----+-----+-----+-----
          10 |          0 |          15 |          5
(1 row)
```

Таким образом, чтобы таблица с очередью вовремя очищалась, можно:

- модифицировать процедуру process_queue таким образом, чтобы обеспечить ее гарантированное периодическое завершение и положиться на автоочистку;
- периодически запускать обычную (неавтоматическую) очистку. Один из способов реализации — использование фоновых процессов, которые рассмотрены в соответствующей теме этого курса.

Зависшие сообщения

остаются в статусе «в работе» при аварийном завершении обработчика

```
take_message();  
COMMIT;
```

```
-- обработка
```



Решение

проверять существование процесса-обработчика, указанного в таблице при отсутствии возвращать сообщение в статус «новый» (возможно)

Чего не хватает в нашей реализации?

Во-первых, обработчик может завершиться аварийно. Если мы фиксируем изменение статуса обработки, то событие «повиснет» в статусе «в работе» и не будет больше обрабатываться.

В нашей реализации мы уже сделали шаг в нужную сторону: в таблице сохраняется номер обслуживающего процесса (pid), который взял событие в работу. Можно написать простую проверку: если pid имеется в таблице, но процесса с таким номером нет в системе — значит, произошел сбой.

Что делать в таком случае? Если обработка события выполнялась в одной транзакции, то она была прервана и, следовательно, можно безопасно вернуть событие в статус «новое» — оно будет обработано повторно.

Если же обработка делится на несколько транзакций, надо быть уверенным в том, что обработку можно запускать повторно.

Корректная обработка исключительных ситуаций

Сохранение результатов обработки

Решение

- не удалять обработанные сообщения, а помечать отдельным статусом
- потребуется правильный индекс
- потребуется периодическое удаление исторических данных
- обращаем внимание на очистку

15

Во-вторых, наша реализация никак не обрабатывает исключительные ситуации. Это, конечно, несложно добавить. При возникновении исключения хотелось бы иметь информацию о том, что случилось.

Да и если событие обработано без ошибок, может быть полезным сохранять какую-то информацию об обработке. Это, конечно, зависит от конкретной задачи.

Наша реализация удаляет обработанные события из очереди, но вместо этого можно оставлять их, помечая специальным статусом («завершено», «ошибка» и т. п.). Тогда всю информацию об обработке можно иметь непосредственно в таблице с событиями. В таком случае потребуется эффективный доступ к еще не обработанным сообщениям: частичный индекс с условием `pid IS NULL`. (Другим решением может быть перенос обработанных событий в отдельную таблицу.)

За удобство потребуется платить реализацией периодического удаления «хвоста» очереди — исторических данных. Если период достаточно большой, то, возможно, удаление надо выполнять пакетами — чтобы не допускать лишнего разрастания таблицы и не мешать очистке.

И, поскольку таблица очередей изменяется довольно активно, надо обеспечить ее своевременную очистку, о чем говорилось в демонстрации.

Асинхронная обработка полезна во многих случаях

Внешние системы имеет смысл использовать, если

- вписываются в общую архитектуру информационной системы
- к реализации предъявляются серьезные требования

Очередь сообщений в базе данных —
простое решение для простых задач

- важна правильная реализация:
- эффективное получение очередного события (SKIP LOCKED),
- избегание долгих транзакций,
- своевременная очистка таблицы очереди
- чем больше требований, тем сложнее будет реализация



1. В приложении предусмотрен механизм фоновых заданий, но серверная часть обработки очереди отсутствует.
Напишите недостающие функции:
 - `take_task` — получает очередное задание из очереди;
 - `complete_task` — завершает обработку задания;
 - `process_tasks` — основной цикл обработки заданий.
2. Запустите процедуру обработки очереди заданий в фоновом режиме. Проверьте, что фоновые задания, поставленные в очередь в приложении, выполняются, а результаты их работы доступны для просмотра.

1. Фоновые задания позволяют запустить специально зарегистрированную функцию из пользовательского интерфейса и затем просматривать состояние и результат выполнения.

В качестве результата функция может возвращать множество строк, то есть в простейшем виде тело функции может состоять из одного SQL-запроса. На вход функция должна принимать один параметр типа `jsonb`. Пример задания: **public.greeting_program**.

Напишите подпрограммы **take_task**, **complete_task** и **process_tasks** по аналогии с показанными в демонстрации примерами. Учтите:

- **take_task** должна возвращать задачу в статусе «scheduled» и заполнить подходящие поля таблицы `tasks`:
`started` = текущее время, `status` = «running», `pid` = номер процесса.
- **complete_task** должна не удалять задание, а заполнить поля `tasks`:
`finished` = текущее время,
при нормальном завершении: `status` = «finished», `result` = результат,
в случае ошибки: `status` = «error», `result` = сообщение об ошибке.
- **process_tasks** не должна завершаться; организуйте бесконечный цикл с задержкой в 1 сек между задачами. Убедитесь, что в режиме ожидания не возникает долгой транзакции. Для удобства установите параметр `application_name` в значение «process_tasks».

Для фактического выполнения задания процедура должна вызвать функцию **empapi.run(task tasks)**. В случае успешного выполнения функция вернет результат, оформленный в виде текстовой строки. В случае ошибки будет сгенерировано исключение.

1. Реализация обработки очереди заданий

Функция получения задания из очереди аналогична показанной в демонстрации, но должна учитывать поля таблицы:

```
=> SELECT * FROM tasks \gx
```

```
-[ RECORD 1 ]-----  
task_id      | 1  
program_id   | 1  
status       | scheduled  
params       |  
pid          |  
started      |  
finished     |  
result       |  
host         |  
port         |
```

(Игнорируйте столбцы host и port — они пригодятся в теме «Обзор физической репликации».)

```
=> CREATE FUNCTION take_task(OUT task tasks) AS $$  
BEGIN  
    SELECT *  
    INTO task  
    FROM tasks  
    WHERE status = 'scheduled'  
    ORDER BY task_id LIMIT 1  
    FOR UPDATE SKIP LOCKED;  
  
    UPDATE tasks  
    SET status = 'running',  
        started = current_timestamp,  
        pid = pg_backend_pid()  
    WHERE task_id = task.task_id;  
END;  
$$ LANGUAGE plpgsql VOLATILE;  
  
CREATE FUNCTION
```

Поскольку мы не будем удалять задания из очереди, создадим частичный индекс для эффективного доступа к следующему необработанному заданию:

```
=> CREATE INDEX ON tasks(task_id) WHERE status = 'scheduled';  
  
CREATE INDEX
```

Функция завершения работы с заданием дополнительно принимает статус завершения и текстовый результат:

```
=> CREATE FUNCTION complete_task(task tasks, status text, result text) RETURNS void  
LANGUAGE sql VOLATILE  
BEGIN ATOMIC  
    UPDATE tasks  
    SET finished = current_timestamp,  
        status = complete_task.status,  
        result = complete_task.result  
    WHERE task_id = task.task_id;  
END;  
  
CREATE FUNCTION
```

Процедура обработки очереди:

```

=> CREATE PROCEDURE process_tasks() AS $$
DECLARE
    task tasks;
    result text;
    ctx text;
BEGIN
    SET application_name = 'process_tasks';
    <<forever>>
    LOOP
        PERFORM pg_sleep(1);
        SELECT * INTO task FROM take_task();
        COMMIT;
        CONTINUE forever WHEN task.task_id IS NULL;

        BEGIN
            result := empapi.run(task);
            PERFORM complete_task(task, 'finished', result);
        EXCEPTION
            WHEN others THEN
                GET STACKED DIAGNOSTICS
                    result := MESSAGE_TEXT, ctx := PG_EXCEPTION_CONTEXT;
                PERFORM complete_task(
                    task, 'error', result || E'\n' || ctx
                );
        END;

        COMMIT;
    END LOOP;
END;
$$ LANGUAGE plpgsql;

```

CREATE PROCEDURE

Обратите внимание, что первая команда COMMIT предшествует команде CONTINUE. В противном случае при отсутствии заданий возникала бы долгая транзакция.

Несколько слов о том, зачем нужна функция run. В принципе, выполнить задание и получить результат можно было бы таким образом:

```

func := (
    SELECT p.func FROM programs p WHERE p.program_id = task.program_id
);

EXECUTE format(
    $$SELECT string_agg(f::text, E'\n') FROM %I($1) AS f$$,
    func
)
INTO result
USING task.params;

```

К сожалению, PL/pgSQL не позволяет гибко работать со значениями составного типа: у значения неизвестного наперед типа (record) нельзя перебрать все имеющиеся в нем поля. Поэтому для вывода приходится полагаться на стандартное преобразование строки в текст. Это будет некрасиво выглядеть в случае нескольких полей:

```

=> SELECT string_agg(f::text, E'\n') FROM greeting_task() AS f;

 string_agg
-----
(1,"Hello, world!")
(2,"Hello, world!")
(3,"Hello, world!")
(1 row)

```

Для аккуратного оформления результата можно воспользоваться другим процедурным языком. Мы используем функцию, написанную на PL/Python. Функция run не вызывается напрямую приложением, но в теме «Обзор физической репликации» мы будем вызывать ее на другом сервере, поэтому она находится в схеме empapi, а не public.

Подробнее о том, в каких случаях могут пригодиться другие языки, будет рассказано в теме «Языки программирования».

2. Запуск обработки очереди в фоновом режиме

В очереди стоит тестовое задание:

```

=> SELECT * FROM tasks \gx

```

```

-[ RECORD 1 ]-----
task_id      | 1
program_id   | 1
status       | scheduled
params       |
pid          |
started      |
finished     |
result       |
host         |
port         |

```

Запускаем обработку (в один поток) и, если все сделано правильно, оно будет выполнено.

```

=> SELECT * FROM pg_background_detach(
      pg_background_launch('CALL process_tasks()')
);

```

```

pg_background_detach
-----

```

(1 row)

Подождем немного...

```

=> SELECT * FROM tasks \gx

```

```

-[ RECORD 1 ]-----
task_id      | 1
program_id   | 1
status       | scheduled
params       |
pid          |
started      |
finished     |
result       |
host         |
port         |

```

Задание успешно выполнено. Обратите внимание, что результат выполнения содержит и названия столбцов из оригинального запроса.

Фоновые процессы, обрабатывающие очередь, легко найти благодаря тому, что процедура устанавливает параметр application_name:

```

=> SELECT pid, wait_event_type, wait_event, query
FROM pg_stat_activity
WHERE application_name = 'process_tasks' \gx

```

(0 rows)

1. Напишите тест, проверяющий, что обработка очереди, показанная в демонстрации, работает корректно при выполнении в несколько потоков.
Убедитесь, что тест не проходит, если убрать предложение `FOR UPDATE SKIP LOCKED`.
2. Добавьте в реализацию проверку «зависших» сообщений.
Если такая ситуация будет обнаружена, зависшее сообщение должно быть снова принято в работу.

1. Вставьте в таблицу сообщений большое количество строк и проверьте, что:

а) было обработано каждое сообщение;

б) каждое сообщение было обработано ровно один раз.

Уберите из реализации секундную задержку (имитацию работы), чтобы тест выполнялся быстрее и с достаточным уровнем конкурентности между процессами.

1. Тестирование реализации очереди

```
=> CREATE DATABASE ext_async;
```

CREATE DATABASE

```
=> \c ext_async
```

You are now connected to database "ext_async" as user "student".

Повторим реализацию очереди, показанную в демонстрации.

Таблица:

```
=> CREATE TABLE msg_queue(  
    id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,  
    payload jsonb NOT NULL,  
    pid integer DEFAULT NULL  
);
```

CREATE TABLE

Функция получения и блокирования очередного сообщения:

```
=> CREATE FUNCTION take_message(OUT msg msg_queue) AS $$  
BEGIN  
    SELECT *  
    INTO msg  
    FROM msg_queue  
    WHERE pid IS NULL  
    ORDER BY id LIMIT 1  
    FOR UPDATE SKIP LOCKED;  
  
    UPDATE msg_queue  
    SET pid = pg_backend_pid()  
    WHERE id = msg.id;  
END;  
$$ LANGUAGE plpgsql VOLATILE;
```

CREATE FUNCTION

Функция завершения работы с сообщением:

```
=> CREATE FUNCTION complete_message(msg msg_queue) RETURNS void  
LANGUAGE sql VOLATILE  
BEGIN ATOMIC  
    DELETE FROM msg_queue WHERE id = msg.id;  
END;
```

CREATE FUNCTION

В процедуру обработки очереди внесем изменение: вместо секундной задержки будем записывать информацию об обрабатываемом сообщении в отдельную таблицу:

```
=> CREATE TABLE msg_log(  
    id bigint,  
    pid integer  
);
```

CREATE TABLE

```
=> CREATE PROCEDURE process_queue() AS $$  
DECLARE  
    msg msg_queue;  
BEGIN  
    LOOP  
        SELECT * INTO msg FROM take_message();  
        EXIT WHEN msg.id IS NULL;  
        COMMIT;  
  
        -- обработка  
        INSERT INTO msg_log(id, pid) VALUES (msg.id, pg_backend_pid());  
  
        PERFORM complete_message(msg);  
        COMMIT;  
    END LOOP;  
END;  
$$ LANGUAGE plpgsql;
```

CREATE PROCEDURE

Создаем большое количество сообщений:

```
=> INSERT INTO msg_queue(payload)
SELECT to_jsonb(id) FROM generate_series(1,1000) id;
```

INSERT 0 1000

Запускаем обработку в два потока, засекая время:

```
student$ psql ext_async
```

```
=> \timing on
```

Timing is on.

```
=> CALL process_queue();
```

```
| => CALL process_queue();
```

```
| CALL
```

CALL

Time: 10281,169 ms (00:10,281)

```
=> \timing off
```

Timing is off.

Проанализируем результаты. При корректной работе мы должны обнаружить в журнальной таблице ровно 1000 уникальных идентификаторов, что будет означать, что обработаны все события, и ни одно не обработано дважды.

```
=> SELECT count(*), count(DISTINCT id) FROM msg_log;
```

count		count
-----+		-----
1000		1000
(1 row)		

Все корректно.

Проверим теперь реализацию без предложения FOR UPDATE SKIP LOCKED.

```
=> CREATE OR REPLACE FUNCTION take_message(OUT msg msg_queue) AS $$
BEGIN
```

```
    SELECT *
    INTO msg
    FROM msg_queue
    WHERE pid IS NULL
    ORDER BY id LIMIT 1
    /*FOR UPDATE SKIP LOCKED*/;
```

```
    UPDATE msg_queue
    SET pid = pg_backend_pid()
    WHERE id = msg.id;
```

```
END;
```

```
$$ LANGUAGE plpgsql VOLATILE;
```

CREATE FUNCTION

```
=> TRUNCATE msg_queue;
```

TRUNCATE TABLE

```
=> TRUNCATE msg_log;
```

TRUNCATE TABLE

```
=> INSERT INTO msg_queue(payload)
SELECT to_jsonb(id) FROM generate_series(1,1000) id;
```

INSERT 0 1000

Запускаем обработку:

```
=> \timing on
```

Timing is on.

```
=> CALL process_queue();
```

```
| => CALL process_queue();
```

CALL
Time: 15988,459 ms (00:15,988)

| CALL

=> \timing off

Timing is off.

=> SELECT count(*), count(DISTINCT id) FROM msg_log;

count	count
1499	1000

(1 row)

Как видим, часть сообщений была обработана дважды. Например:

=> SELECT id, array_agg(pid) FROM msg_log
GROUP BY id HAVING count(*) > 1
LIMIT 10;

id	array_agg
1489	{73541,73334}
1989	{73334,73541}
1108	{73334,73541}
1128	{73541,73334}
1284	{73541,73334}
1899	{73334,73541}
1494	{73541,73334}
1831	{73541,73334}
1003	{73541,73334}
1331	{73541,73334}

(10 rows)

Это произошло из-за того, что сообщение, обрабатываемое одним процессом, никак не блокируется и доступно для другого процесса.

Восстановим корректную функцию:

```
=> CREATE OR REPLACE FUNCTION take_message(OUT msg msg_queue) AS $$  
BEGIN  
    SELECT *  
    INTO msg  
    FROM msg_queue  
    WHERE pid IS NULL  
    ORDER BY id LIMIT 1  
    FOR UPDATE SKIP LOCKED;  
  
    UPDATE msg_queue  
    SET pid = pg_backend_pid()  
    WHERE id = msg.id;  
END;  
$$ LANGUAGE plpgsql VOLATILE;
```

CREATE FUNCTION

2. Обработка зависших сообщений

Мы можем перехватить ошибку, возникающую при обработке события, но тем не менее всегда есть шанс того, что сама процедура-обработчик завершится аварийно. Сымитируем такую ситуацию:

=> TRUNCATE msg_queue;

TRUNCATE TABLE

=> TRUNCATE msg_log;

TRUNCATE TABLE

=> INSERT INTO msg_queue(payload)
SELECT to_jsonb(id) FROM generate_series(1,1000) id;

INSERT 0 1000

Запускаем обработку...

| => CALL process_queue();

...а в это время в другом сеансе:

```
=> BEGIN;
```

```
BEGIN
```

```
=> LOCK TABLE msg_log;
```

```
LOCK TABLE
```

```
=> SELECT pid, pg_terminate_backend(pid) FROM msg_log LIMIT 1;
```

```
pid | pg_terminate_backend
-----+-----
73541 | t
(1 row)
```

```
=> COMMIT;
```

```
COMMIT
```

```
FATAL: terminating connection due to administrator command
CONTEXT: SQL statement "INSERT INTO msg_log(id, pid) VALUES (msg.id, pg_backend_pid())"
PL/pgSQL function process_queue() line 11 at SQL statement
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
connection to server was lost
```

Обработчик «упал». Причем, благодаря команде LOCK TABLE, — сразу после того, как зафиксировал номер процесса в таблице очереди. В очереди остались необработанные сообщения и среди них — одно зависшее:

```
=> SELECT count(*), count(DISTINCT id) FROM msg_log;
```

```
count | count
-----+-----
5 | 5
(1 row)
```

```
=> SELECT * FROM msg_queue WHERE pid IS NOT NULL;
```

```
id | payload | pid
-----+-----+-----
2006 | 6 | 73541
(1 row)
```

Самый простой способ исправить ситуацию — изменить функцию выбора сообщения:

```
=> CREATE OR REPLACE FUNCTION take_message(OUT msg msg_queue) AS $$
BEGIN
    SELECT *
    INTO msg
    FROM msg_queue
    WHERE pid IS NULL OR pid NOT IN (SELECT pid FROM pg_stat_activity)
    ORDER BY id LIMIT 1
    FOR UPDATE SKIP LOCKED;

    UPDATE msg_queue
    SET pid = pg_backend_pid()
    WHERE id = msg.id;
END;
$$ LANGUAGE plpgsql VOLATILE;
```

```
CREATE FUNCTION
```

Если события обрабатываются быстро и важна высокая пропускная способность, то проверку лучше выполнять отдельно и только время от времени, чтобы избежать постоянного обращения к pg_stat_activity.

Снова запустим обработчик, и все сообщения, включая зависшее, будут обработаны.

```
=> CALL process_queue();
```

```
CALL
```

```
=> SELECT count(*), count(DISTINCT id) FROM msg_log;
```

count	count
1000	1000

(1 row)