

Расширяемость Асинхронная обработка



Авторские права

© Postgres Professional, 2020 год.

Авторы: Егор Рогов, Павел Лузанов

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.

Плюсы

эта система работает

Возможные минусы

мало гибкости, например, исключительно пакетная обработка

плохо документирована

внешняя программа-демон

избыточно сложная реализация в расчете на старые версии PostgreSQL

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.

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.
- Избыточно сложная реализация. Система была написана во времена довольно старых версий PostgreSQL и содержит массу сложного кода, ненужного в современных версиях.

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 удобно представить типом JSON — так очередь будет достаточно универсальна.

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

```
CREATE DATABASE
```

```
=> \c ext_async
```

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

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.

Во-первых, убедимся, что запрос, выбирающий одно сообщение по порядку, выполняется эффективно.

```
=> EXPLAIN (costs off)  
SELECT * FROM msg_queue  
WHERE pid IS NULL -- никем не обрабатывается  
ORDER BY id LIMIT 1;
```

QUERY PLAN

```
-----  
Limit  
  -> Index Scan using msg_queue_pkey on msg_queue  
      Filter: (pid IS NULL)  
(3 rows)
```

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  
ORDER BY id LIMIT 1  
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) 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
```


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 AS $$  
DELETE FROM msg_queue  
WHERE id = msg.id;  
$$ LANGUAGE sql VOLATILE;  
  
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 %; backend_xmin=%',  
            pg_backend_pid(),  
            msg.payload,  
            (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: [97486] processed 2; backend_xmin=863  
NOTICE: [97486] processed 3; backend_xmin=863  
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: [97649] processed 2; backend_xmin=865  
| NOTICE: [97649] processed 4; backend_xmin=865  
| NOTICE: [97649] processed 6; backend_xmin=865  
| NOTICE: [97649] processed 8; backend_xmin=865  
| NOTICE: [97649] processed 10; backend_xmin=865  
| CALL
```

```
NOTICE: [97486] processed 1; backend_xmin=865
NOTICE: [97486] processed 3; backend_xmin=865
NOTICE: [97486] processed 5; backend_xmin=865
NOTICE: [97486] processed 7; backend_xmin=865
NOTICE: [97486] processed 9; backend_xmin=865
CALL
Time: 5013,634 ms (00:05,014)
```

```
=> \timing off
```

```
Timing is off.
```

Обратите внимание, что горизонт транзакций удерживается 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 модуля Многоверсионность «Обзор внутреннего устройства» и «Обзор очистки», можно с уверенностью сказать, что обработка очереди будет мешать нормальной работе очистки.

Что надо: каждое событие в отдельной транзакции

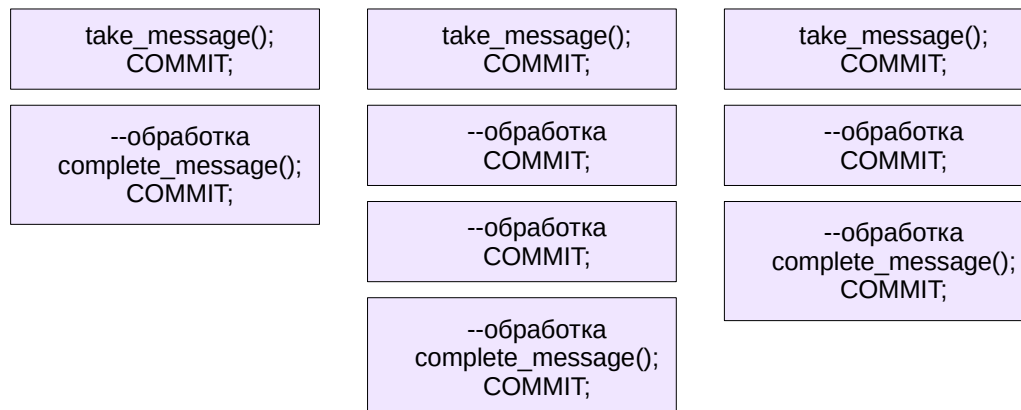
```
take_message();  
--обработка  
complete_message();  
  
COMMIT;
```

```
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



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, и 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 %; backend_xmin=%',
            pg_backend_pid(),
            msg.payload,
            (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:  [97486] processed 1; backend_xmin=871
NOTICE:  [97486] processed 2; backend_xmin=873
NOTICE:  [97486] processed 3; backend_xmin=875
NOTICE:  [97486] processed 4; backend_xmin=877
NOTICE:  [97486] processed 5; backend_xmin=879
CALL
```

Теперь горизонт транзакций продвигается вперед и не будет мешать очистке.

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)

12

Чего не хватает в нашей реализации?

Во-первых, отметим возможность того, что обработчик аварийно завершится в процессе работы. 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 периодическая очистка исторических данных
обращаем внимание на автоочистку

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.

Это удобно, но потребует реализовать периодическую очистку «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 настроить автоматическую очистку так, чтобы она справлялась с изменениями. Настройка автоочистки — тема модуля «Многоверсионность» курса DBA2.

Асинхронная обработка полезна во многих случаях

Внешние системы имеет смысл использовать, если

- они вписываются в общую архитектуру информационной системы
- 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

- чем больше требований, тем сложнее будет реализация

- обратить внимание на настройку автоочистки для таблицы очереди



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, т. е. в простейшем виде функция может быть написана на SQL и содержать один SQL-запрос. 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.

```
student$ psql bookstore2
```

1. Реализация обработки очереди заданий

Функция получения задания из очереди аналогично 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 «Физическая репликация».)

```
=> CREATE OR REPLACE 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
```

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

```
=> CREATE FUNCTION complete_task(task tasks, status text, result text)  
RETURNS void AS $$  
    UPDATE tasks  
    SET finished = current_timestamp,  
        status = complete_task.status,  
        result = complete_task.result  
    WHERE task_id = task.task_id;  
$$ LANGUAGE sql VOLATILE;  
  
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 «Физическая репликация» 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 серверного 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       | finished
params       |
pid          | 124860
started      | 2020-11-06 17:06:00.701851+03
finished     | 2020-11-06 17:06:01.713774+03
result       | num    greeting      +
              | ---  -----      +
              | 1    Hello, world!    +
              | 2    Hello, world!    +
              | 3    Hello, world!    +
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

```

```

-[ RECORD 1 ]---+-----
pid          | 124860
wait_event_type | Timeout
wait_event    | PgSleep
query         | CALL process_tasks()

```

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 AS $$  
DELETE FROM msg_queue  
WHERE id = msg.id;  
$$ LANGUAGE sql VOLATILE;
```

```
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(1,10000) id;
```

```
INSERT 0 10000
```

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: 14352,852 ms (00:14,353)
```

```
=> \timing off
```

Timing is off.

Проанализируем результаты. При корректной работе мы должны обнаружить в журнальной таблице ровно 10000 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
-----+-----
10000  | 10000
(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(1,10000) id;
```

```
INSERT 0 10000
```

Запускаем обработку:

```
=> \timing on
```

Timing is on.

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

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

```
| CALL
```



```
CALL
Time: 16373,856 ms (00:16,374)
```

```
=> \timing off
```

Timing is off.

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

```
count | count
-----+-----
15110 | 10000
(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
-----+-----
10896 | {125325,125129}
18803 | {125325,125129}
12502 | {125325,125129}
13093 | {125325,125129}
13520 | {125129,125325}
18181 | {125325,125129}
14852 | {125325,125129}
11092 | {125129,125325}
17695 | {125325,125129}
17573 | {125325,125129}
(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(1,10000) id;
```

```
INSERT 0 10000
```

Запускаем обработку...

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

...а в это время в другом сеансе:

```
=> BEGIN;
```

BEGIN

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

LOCK TABLE

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

```
pg_terminate_backend
-----
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
-----+-----
1659  | 1659
(1 row)
```

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

```
id | payload | pid
-----+-----+-----
21660 | 1660 | 125325
(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
-----+-----
10000  | 10000
(1 row)
```

