УТВЕРЖДЕН
RU_ПГ.1002-0111

СИСТЕМА УПРАВЛЕНИЯ БАЗАМИ ДАННЫХ PG360

Руководство программиста

Листов 325

Подп. и дата	
Инв.№ дубл.	
Взам. инв.№	
Подп. и дата	
Инв.№ подл.	

2025

АННОТАЦИЯ

Настоящий документ предоставляет сведения о системе управления базами данных " PG360" в защищенном исполнении (далее – СУБД PG360).

СОДЕРЖАНИЕ

1. НАЗНАЧЕНИЕ И УСЛОВИЯ ПРИМЕНЕНИЯ ПРОГРАММЫ

1.1. СУБД PG360 предназначена для сохранения, обработки и извлечения из базы данных предприятия той информации, к которой субъекту (пользователю, администратору или прикладной программе) предоставлен доступ, для последующего ее предоставления субъекту через интерфейс клиентского или серверного приложения (сервиса).

СУБД PG360 является посредником между пользователем и базой данных, представляя конечному субъекту единое интегрированное представление данных в базе данных.

Функционально СУБД PG360 обеспечивает управление созданием, обслуживанием баз данных и использованием информации баз данных. С ее помощью можно создавать, модифицировать или удалять записи, отправлять транзакцию – набор из нескольких последовательных запросов на языке запросов SQL.

1.2. Основные задачи, выполняемые PG360:

– гибкий доступ к базам данных, их организация и хранение;

– создавать и администрировать (удалять, изменять и объединять) базы данных;

– содержать данные в структурированном виде и необходимом формате;

– принимать подключения клиентских приложений пользователя;

– принимать подключения серверных приложений СУБД;

– выполнять различные запросы субъектов к базам данных;

– управлять файлами баз данных;

– защищать данные от нежелательных изменений и попыток взлома;

– загружать и сортировать данные с помощью фильтров;

– делать резервные копии, восстанавливать.

1.3. СУБД PG360 включает следующие взаимодействующие процессы (программы):

– postgres (программа сервера): главный серверный процесс, управляющий файлами баз данных, принимающий подключения клиентских приложений и выполняющий различные запро сы клиентов к базам данных;

– клиентские приложения пользователя: процессы, запрашивающие выполнение операций в базе данных;

– серверные приложения СУБД: процессы, обеспечивающие дополнительные возможности по обслуживанию СУБД.

1.4. Минимальный состав технических средств:

– объем жесткого диска – не менее 5 ГБ;

– объем оперативной памяти – не менее 2 ГБ;

– процессор архитектуры – x86_64;

– тактовая частота процессора – не менее 1 ГГц.

1.5. Минимальный состав программных средств:

Для функционирования системы управления базами данных (далее – СУБД) PG360 на сервере информационной системы, на котором устанавливается СУБД PG360, должна быть установлена операционная система (ОС) семейства Linux:

- Astra Linux 1.7

- AltLinux - Альт Сервер 10

2. ХАРАКТЕРИСТИКИ ПРОГРАММЫ

2.1. В состав СУБД PG360 входят следующие взаимодействующие процессы (программы):

– postgres (программа сервера): главный серверный процесс, управляющий файлами баз данных, принимающий подключения клиентских приложений и выполняющий различные запросы клиентов к базам данных;

– клиентские приложения пользователя: процессы, запрашивающие выполнение операций в базе данных;

– серверные приложения СУБД: процессы, обеспечивающие дополнительные возможности по обслуживанию СУБД.

2.2. СУБД PG360 обеспечивает выполнение следующих возможностей:

– работу с большими объемами;

– поддержку множества типов данных;

– поддержку сложных запросов;

– написание функций на нескольких языках;

– одновременную модификацию базы (возможность одновременного доступа к базе с нескольких устройств);

– возможность расширения. Разработчик может написать для СУБД PG360 собственные типы и их преобразования, операции и функции, ограничения и индексы, собственный процедурный язык для запросов;

– высокую мощность и широкую функциональность. СУБД PG360 мощная, производительная, способна эффективно работать с большими массивами данных.

2.3. Функциональные ограничения на применение СУБД PG360 приведены в таблице 1.

Таблица 1.

Объект	**Верхний предел**	**Комментарий**
размер базы данных	без ограничений	дополнительно ограничивается размером дискового пространства и требованиями ОС
количество баз данных	4 294 950 911	дополнительно ограничивается размером дискового пространства и требованиями ОС
отношений в базе данных	1 431 650 303	
размер отношения	32 ТБ	со значением BLCKSZ по умолчанию, равным 8192 байта
строк в таблице	ограничивается количеством кортежей, которое может уместиться в 4 294 967 295 страниц	
столбцов в таблице	1600	дополнительно ограничивается размером кортежа, который может уместиться в одной странице (см. примечание 1)
столбцов в наборе результатов	1664	
размер поля	1 ГБ	
длина идентификатора	63 байта	может быть увеличена при перекомпиляции PG360
индексов в таблице	без ограничений	ограничивается максимальным количеством отношений в базе данных
столбцов в индексе	32	может быть увеличена при перекомпиляции PG360

Объект	Верхний предел	Комментарий
ключей секционирования	32	может быть увеличена при перекомпиляции PG360

Примечание 1. Максимальное количество столбцов таблицы дополнительно уменьшается в связи с тем, что сохраняемый кортеж должен умещаться в одной странице размером 8192 байта. Например, если не учитывать размер заголовка, кортеж, состоящий из 1600 столбцов int, будет занимать 6400 байт и поместится в странице кучи, тогда как 1600 столбцов bigint займут 12800 байт и в одной странице не поместятся. Поля переменной длины, например типов text, varchar и char, могут храниться отдельно, в таблице TOAST, когда их значения достаточно велики для этого. При этом внутри кортежа кучи должен остаться только 18-байтовый указатель. Для более коротких значений полей переменной длины используется заголовок из 1 или 4 байт, и само значение сохраняется внутри кортежа в куче.

Максимальное количество столбцов может также зависеть от числа столбцов, удалённых из таблицы. И хотя значения удалённых столбцов для создаваемых впоследствии кортежей не хранятся, а только помечаются как NULL в специальной битовой карте, эта карта тоже занимает место.

Примечание 2. На практике, до исчерпания лимитов, указанных в таблице 1, могут быть исчерпаны другие лимиты ОС (например, предел производительности или доступного объёма дискового хранилища).

2.4. Организация работы СУБД PG360

2.4.1. Работа с СУБД PG360 включает следующие этапы:

– установка (инсталляция);

– работа с клиентскими интерфейсами;

– серверное программирование;

– штатная работа.

2.4.2. Установка СУБД PG360 состоит из:

– инсталляции СУБД PG360;

– автоматического запуска СУБД PG360.

2.4.3. Штатная работа СУБД PG360 предполагает следующие действия:

– запуск СУБД PG360;

– выключение СУБД PG360;

– настройку СУБД PG360;

– определение параметров в файлах конфигурации.

Штатная работа оператора с СУБД PG360 описана в документе "Система управления базами данных "PG360". Руководство оператора.

2.5. Инсталляция СУБД PG360

2.5.1. В процессе инсталляции система попросит создать (ввести) пароль суперпользователя СУБД (ожидание пользовательского ввода). Чтобы установить СУБД PG360 без ожидания ввода, для задания пароля суперпользователя можно использовать переменную окружения PG_PASSWORD. Например:

```
export PG_PASSWORD=your_password
<команда установки>
```

или:

```
PG_PASSWORD=your_password <команда установки>
```

или:

```
sudo PG_PASSWORD=your_password <команда установки>
```

2.5.2. Для инсталляции СУБД PG360 на ОС команда для ALT Linux: `apt-get install -y glibc-core libzstd1 liblz4 libxml2 libpam openssl3 krb5-libs libcom_err2 libgssapi_krb5 zlib openldap libicui18n74 libicuuc74 libicudata74 systemd liblzma audit libcap libcap-ng libkeyutils libsasl2-3`

```
libstdc++    libgcc_s    libgcrypt20    libgpg-error    libselinux    pcre2

rpm -i rpmbuild/RPMS/x86_64/pg360-1.11-1.x86_64.rpm
```

2.6. Автоматический запуск СУБД PG360

2.6.1. После инсталляции СУБД PG360 соответствии с пунктом 2.5, она будет запущена автоматически.

2.7. Клиентские интерфейсы

Клиентские программные интерфейсы, включённые в дистрибутив PG360:

– libpq – библиотека для языка C;

– программный интерфейс и функции языка запросов для работы с данными больших объектов PG360;

– ECPG — Встраиваемый SQL в C;

– Информационные схемы.

Для изучения клиентских интерфейсов нужно уметь работать с базой данных, используя команды SQL, и знать язык программирования, на который ориентирован определённый интерфейс.

2.7.1. libpq – библиотека для языка C

Библиотека libpq – это интерфейс PG360 для программирования приложений на языке C. Библиотека libpq содержит набор функций, используя которые клиентские программы могут передавать запросы серверу PG360 и принимать результаты этих запросов.

Клиентские программы, которые используют libpq, должны включать заголовочный файл libpq-fe.h и компоноваться с библиотекой libpq.

27.1.1. Функции управления подключением к базе данных

Прикладная программа может иметь несколько подключений к серверу, открытых одновременно. Каждое соединение представляется объектом PGconn, который можно получить от функции PQconnectdb, PQconnectdbParams_или PQsetdbLogin. Эти функции всегда возвращают ненулевой указатель на объект, кроме случая, когда не остаётся свободной памяти даже для объекта PGconn. Прежде чем передавать запросы через объект подключения, следует вызвать функцию PQstatus для проверки возвращаемого значения в случае успешного подключения.

Функции, связанные с созданием подключения к серверу PG360:

– PQconnectdbParams – Создаёт новое подключение к серверу баз данных.

```
PGconn *PQconnectdbParams(const char * const *keywords,
const char * const *values, int expand_dbname);
```

Эта функция открывает новое соединение с базой данных, используя параметры, содержащиеся в двух массивах, завершающихся символом NULL. Первый из них, keywords, определяется как массив строк, каждая из которых представляет собой ключевое слово. Второй, values, даёт значение для каждого ключевого слова.

Передаваемые массивы могут быть пустыми (в этом случае будут использоваться все параметры по умолчанию) или содержать одно или несколько имён/значений параметров. При этом они должны быть одинаковой длины. Просмотр массивов завершается, как только в массиве keywords встречается NULL. Если элемент массива values, соответствующий отличному от NULL элементу keywords, содержит пустую строку или NULL, такой параметр пропускается и просматривается следующая пара элементов массива.

Когда expand_dbname имеет ненулевое значение, первый параметр *dbname* может содержать *строку подключения*. В этом случае она «разворачивается» в отдельные параметры подключения, извлечённые из этой строки. Значение считается строкой подключения, а не просто именем базы данных, если оно содержит знак равно (=) или начинается с обозначения схемы URI. Таким способом обрабатывается только первое вхождение *dbname*, следующие параметры *dbname* будут восприниматься как просто имя базы данных.

После разбора всех элементов массива и развёрнутой строки подключения (если она задана), параметры подключения, которые остались незаданными, получают значения по умолчанию. Если незаданному параметру соответствует установленная переменная окружения, будет использоваться её значение. Если такая переменная не задана, для параметра будет использоваться встроенное значение по умолчанию.

– PQconnectdb – Создаёт новое подключение к серверу баз данных.

```
PGconn *PQconnectdb(const char *conninfo);
```

Эта функция открывает новое соединение с базой данных, используя параметры, полученные из строки conninfo.

Передаваемая строка может быть пустой. В этом случае используются все параметры по умолчанию. Она также может содержать одно или более значений параметров, разделённых пробелами, или URI.

– PQsetdbLogin – Создаёт новое подключение к серверу баз данных.

```
PGconn *PQsetdbLogin(const char *pghost,const char *pgport, const char *pgoptions, const char *pgtty, const char *dbName, const char *login, const char *pwd);
```

Если параметр *dbName* содержит знак = или имеет допустимый префикс URI для подключения, то он воспринимается в качестве строки *conninfo* точно таким же образом, как если бы он был передан функции PQconnectdb, а оставшиеся параметры применяются, как указано для PQconnectdbParams.

– PQsetdb – Создаёт новое подключение к серверу баз данных.

```
PGconn *PQsetdb(char *pghost,char *pgport, char *pgoptions, char *pgtty, char *dbName);
```

Это макрос, который вызывает PQsetdbLogin с нулевыми указателями в качестве значений параметров *login* и *pwd*.

– PQconnectStartParams PQconnectStart PqconnectPoll – Создают подключение к серверу баз данных неблокирующим способом.

```
PGconn *PQconnectStartParams(const char * const *keywords,const char * const *values, int expand_dbname);
PGconn *PQconnectStart(const char *conninfo);
PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

Три эти функции используются для того, чтобы открыть подключение к серверу баз данных таким образом, чтобы поток исполнения приложения не был заблокирован при выполнении удалённой операции ввода/вывода в процессе подключения. Суть этого подхода в том, чтобы ожидание завершения операций ввода/вывода могло происходить в главном цикле приложения, а не внутри функций PQconnectdbParams или PQconnectdb, с тем, чтобы приложение могло управлять этой операцией параллельно с другой работой.

С помощью функции PQconnectStartParams подключение к базе данных выполняется, используя параметры, взятые из массивов keywords и values, а управление осуществляется с помощью expand_dbname, как описано выше для PQconnectdbParams.

С помощью функции PQconnectStart подключение к базе данных выполняется, используя параметры, взятые из строки conninfo, как описано выше для PQconnectdb.

Ни PQconnectStartParams, ни PQconnectStart, ни PQconnectPoll не заблокируются до тех пор, пока выполняется ряд ограничений:

– Параметр hostaddr должен использоваться так, чтобы для разрешения заданного имени не требовалось выполнять запросы DNS.

– Если вызывается PQtrace, необходимо сделать так, чтобы поток, в который выводится трассировочная информация, не заблокировался.

– Перед вызовом PQconnectPoll необходимо перевести сокет в соответствующее состояние, как описано ниже.

Чтобы начать неблокирующий запрос на подключение, необходимо вызвать PQconnectStart или PQconnectStartParams. Если результатом будет null, значит libpq не смогла выделить память для новой структуры PGconn. В противном случае возвращается действительный указатель PGconn (хотя он ещё не представляет установленное подключение к базе данных). Затем необходимо вызвать PQstatus(conn). Если результатом будет CONNECTION_BAD, значит попытка подключения уже не будет успешной, возможно, из-за неверных параметров.

Если вызов PQconnectStart redили PQconnectStartParams оказался успешным, теперь нужно опросить libpq для продолжения процедуры подключения. Необходимо вызвать PQsocket(conn) для получения дескриптора нижележащего сокета, через который устанавливается соединение. (Внимание: этот сокет может меняться от вызова к вызову PQconnectPoll.) Необходимо организовать цикл таким образом: если PQconnectPoll(conn) при последнем вызове возвращает PGRES_POLLING_READING, ожидать, пока сокет не окажется готовым для чтения (это покажет функция select(), poll() или подобная системная функция). Затем снова вызвать PQconnectPoll(conn). Если же PQconnectPoll(conn) при последнем вызове возвратила PGRES_POLLING_WRITING, дождаться готовности сокета к записи, а затем снова вызвать PQconnectPoll(conn). На первой итерации, то есть когда ещё не вызвано PQconnectPoll, необходимо реализовать то же поведение, что и после получения PGRES_POLLING_WRITING. Продолжить этот цикл, пока PQconnectPoll(conn) не выдаст значение PGRES_POLLING_FAILED, сигнализирующее об ошибке при установлении соединения, или PGRES_POLLING_OK, показывающее, что соединение установлено успешно.

В любое время в процессе подключения его состояние можно проверить, вызвав PQstatus. Если этот вызов возвратит CONNECTION_BAD, значит, процедура подключения завершилась сбоем; если вызов возвратит CONNECTION_OK, значит, соединение готово. Оба эти состояния можно определить на основе возвращаемого значения функции PQconnectPoll. Другие состояния могут также иметь место в течение (и только в течение) асинхронной процедуры подключения. Они показывают текущую стадию процедуры подключения.

Возможные состояния:

CONNECTION_STARTED – Ожидание, пока соединение будет установлено.

CONNECTION_MADE – Соединение установлено; ожидание отправки.

CONNECTION_AWAITING_RESPONSE – Ожидание ответа от сервера.

CONNECTION_AUTH_OK – Аутентификация получена; ожидание завершения запуска серверной части.

CONNECTION_SSL_STARTUP – Согласование SSL-шифрования.

CONNECTION_SETENV – Согласование значений параметров, зависящих от программной среды.

CONNECTION_CHECK_WRITABLE – Проверка, можно ли через подключение выполнять пишущие транзакции.

CONNECTION_CONSUME – Прочтение всех оставшихся ответных сообщений через подключение.

Параметр подключения connect_timeout игнорируется, когда используется PQconnectPoll; именно приложение отвечает за принятие решения о том, является ли истекшее время чрезмерным. В противном случае вызов PQconnectStart с последующим вызовом PQconnectPoll в цикле будут эквивалентны вызову PQconnectdb.

– PQconndefaults – Возвращает значения по умолчанию для параметров подключения.

```
PQconninfoOption *PQconndefaults(void);
```

Возвращает массив параметров подключения. Он может использоваться для определения всех возможных параметров PQconnectdb и их текущих значений по умолчанию. Возвращаемое значение указывает на массив структур PQconninfoOption, который завершается элементом,

имеющим нулевой указатель keyword. Если выделить память не удалось, то возвращается нулевой указатель. Текущие значения по умолчанию (поля val) будут зависеть от переменных среды и другого контекста. Отсутствующий или неверный файл служб будет просто проигнорирован. Вызывающие функции должны рассматривать данные параметров подключения как данные только для чтения.

После обработки массива параметров необходимо освободить память, передав его функции PQconninfoFree. Если этого не делать, то при каждом вызове функции PQconndefaults будет теряться небольшой объём памяти.

– PQconninfo – Возвращает параметры подключения, используемые действующим соединением.

```
PQconninfoOption *PQconninfo(PGconn *conn);
```

Возвращает массив параметров подключения. Он может использоваться для определения всех возможных параметров PQconnectdb и значений, которые были использованы для подключения к серверу. Возвращаемое значение указывает на массив структур PQconninfoOption, который завершается элементом, имеющим нулевой указатель keyword. Все замечания, приведённые выше для PQconndefaults, также справедливы и для результата PQconninfo.

– PqconninfoParse – Возвращает разобранные параметры подключения, переданные в строке подключения.

```
PQconninfoOption *PQconninfoParse(const char *conninfo, char **errmsg);
```

Разбирает строку подключения и возвращает результирующие параметры в виде массива; возвращает NULL, если возникают проблемы при разборе строки подключения. Возвращаемое значение указывает на массив структур PQconninfoOption, который завершается элементом, имеющим нулевой указатель keyword.

Все разрешённые параметры будут присутствовать в результирующем массиве, но PQconninfoOption для любого параметра, не присутствующего в строке подключения, будет иметь значение NULL в поле val; значения по умолчанию не подставляются.

Если errmsg не равно NULL, тогда в случае успеха *errmsg присваивается NULL, а в противном случае -– адрес строки сообщения об ошибке, объясняющего проблему. Память для этой строки выделяет функция malloc. (Также возможна ситуация, когда *errmsg будет установлено в NULL, и при этом функция возвращает NULL. Это указывает на нехватку памяти.)

После обработки массива параметров необходимо освободить память, передав его функции PQconninfoFree. Если этого не делать, некоторое количество памяти будет теряться при каждом вызове PQconninfoParse. Если же произошла ошибка и errmsg, указывающий на строку сообщения об ошибке, не равен NULL, необходимо освободить память с этой строкой, вызвав PQfreemem.

– PQfinish – Закрывает соединение с сервером. Также освобождает память, используемую объектом PGconn.

```
void PQfinish(PGconn *conn);
```

– Pqreset – Переустанавливает канал связи с сервером.

```
void PQreset(PGconn *conn);
```

Эта функция закроет подключение к серверу, а потом попытается установить новое подключение, используя все те же параметры, которые использовались прежде.

– PQresetStart PqresetPoll – Переустанавливает канал связи с сервером неблокирующим способом.

```
int PQresetStart(PGconn *conn);
PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

Эти функции закроют подключение к серверу, а потом попытаются установить новое подключение, используя все те же параметры, которые использовались прежде. Они отличаются от PQreset тем, что действуют неблокирующим способом. На эти функции налагаются те же ограничения, что и на PQconnectStartParams, PQconnectStart и PQconnectPoll.

Чтобы произвести переподключение, необходимо вызвать PQresetStart. Если она возвратит 0, значит при переподключении произошла ошибка. Если она возвратит 1, необходимо опросить результат операции, используя PQresetPoll, таким же образом, как это делается с помощью PQconnectPoll при обычном установления соединения.

– PQpingParams – сообщает состояние сервера.

```
PGPing PQpingParams(const char * const *keywords,
const char * const *values, int expand_dbname);
```

Функция возвращает одно из следующих значений:

PQPING_OK – Сервер работает и, по-видимому, принимает подключения.

PQPING_REJECT – Сервер работает, но находится в состоянии, которое запрещает подключения (запуск, завершение работы или восстановление после аварийного отказа).

PQPING_NO_RESPONSE – Контакт с сервером не удался.

PQPING_NO_ATTEMPT – Никакой попытки установить контакт с сервером сделано не было, поскольку предоставленные параметры были явно некорректными, или имела место какая- то проблема на стороне клиента (например, нехватка памяти).

– PQping – сообщает состояние сервера.

```
PGPing PQping(const char *conninfo);
```

Возвращаемые значения такие же, как и для PQpingParams.

– PQsetSSLKeyPassHook_OpenSSL – позволяет приложению переопределить реализованный в libpq стандартный вариант обработки файлов с зашифрованными ключами клиентских сертификатов, используя sslpassword или интерактивное приглашение.

```
void PQsetSSLKeyPassHook_OpenSSL(PQsslKeyPassHook_OpenSSL_type
hook);
```

Приложение передаёт указатель на функцию-обработчик со следующей сигнатурой:

```
int callback_fn(char *buf, int size, PGconn *conn);
```

Эту функцию libpq будет вызывать *вместо* своего стандартного обработчика PQdefaultSSLKeyPassHook_OpenSSL. Данная функция должна получить пароль для ключа и скопировать его в результирующий буфер buf размера size. Строка в buf должна завершаться нулём. В результате эта функция должна выдать длину пароля, сохранённого в buf, не считая завершающего нуля. В случае ошибки она должна установить buf[0] = '\0' и выдать 0.

Если пользователь задал размещение ключа явно, заданный путь будет передан при вызове этого обработчика в conn->sslkey. Это поле будет пустым, если используется путь к ключу по умолчанию. Что касается ключей, специфичных для модулей OpenSSL, для них модули могут по своему усмотрению получать пароль через стандартный обработчик OpenSSL или через свой собственный.

Пользовательский обработчик может полностью переопределить функцию PQdefaultSSLKeyPassHook_OpenSSL, либо делегировать ей необрабатываемые им случаи, либо сначала вызывать её и предпринимать какие-то другие действия, если она возвратит 0.

Этот обработчик *не должен* нарушать обычный ход выполнения, выбрасывая исключения, вызывая longjmp(...) и т. п. Он должен завершиться нормально.

PqgetSSLKeyPassHook_OpenSSL – возвращает текущий обработчик пароля для ключа клиентского сертификата либо NULL, если такой обработчик не установлен.

```
PQsslKeyPassHook_OpenSSL_type PQgetSSLKeyPassHook_OpenSSL(void);
```

27.1.1. Строки параметров подключения

Функций библиотеки libpq разбирают заданную пользователем строку для извлечения параметров подключения. Строка входных параметров может быть двух типов: простая строка ключ/значение или URI.

27.1.2. Строки параметров подключения вида «ключ/значение»

Согласно формату ключ/значение, установка каждого параметра выполняется в форме *ключ = значение*, с пробелами между параметрами. Пробелы вокруг знака равенства не являются обязательными. Для записи пустого значения или значения, содержащего пробелы, необходимо заключить его в одинарные кавычки, например, keyword = 'a value'. Одинарные кавычки и символы обратной косой черты внутри значения нужно обязательно экранировать с помощью символа обратной косой черты, т. е., \' и \\.

27.1.3. URI для подключения

Основная форма URI подключения:

```
postgresql://[пользователь@][сервер][/база_данных][?
указание_параметра]
где пользователь:
имя_пользователя[:пароль]
и сервер:
[узел][:порт][,...]
и указание_параметра:
имя=значение[&...]
```

В качестве обозначения схемы URI может использоваться postgresql:// или postgres://. Остальные части URI являются необязательными.

Сервер можно представить либо сетевым именем, либо IP-адресом. При использовании протокола IPv6 нужно заключить адрес в квадратные скобки:

```
postgresql://[2001:db8::1234]/database
```

Часть host интерпретируется в соответствии с описанием параметра host. Так, если эта часть строки пуста или выглядит как абсолютный путь, выбирается соединение через Unix-сокеты, в противном случае инициируется соединение по TCP/IP. Символ косой черты в иерархической части URI является зарезервированным. Поэтому, чтобы указать нестандартный каталог Unix- сокета, нужно поступить одним из двух способов: не задавать эту часть в URI и указать сервер в именованном параметре либо закодировать путь в части host с процентами:

`postgresql:///dbname?host=/var/lib/postgresql postgresql://%2Fvar%2Flib%2Fpostgresql/dbname`

В одном URI можно задать несколько компонентов узлов, каждый с необязательным указанием порта. URI вида:

`postgresql://host1:port1,host2:port2,host3:port3/`

равнозначно строке подключения вида:

`host=host1,host2,host3 port=port1,port2,port3.`

Эти узлы будут перебираться по очереди, пока не будет установлено подключение.

27.1.14. Указание нескольких узлов

В строке подключения можно задать несколько узлов, к которым клиент будет пытаться подключиться в заданном порядке. Параметры host, hostaddr и port в формате ключ/значение принимают списки значений, разделённых запятыми. В каждом определяемом параметре должно содержаться одинаковое число элементов, чтобы, например, первый элемент hostaddr соответствовал первому элементу host, второй – второму host и так далее. Исключение составляет port – если этот параметр содержит только один элемент, он применяется ко всем узлам.

В формате URI внутри компонента host можно указать несколько пар host:port, разделённых запятыми.

В любом формате одно имя узла может переводиться в несколько сетевых адресов. Например, часто бывает, что один узел имеет и адрес IPv4, и адрес IPv6.

Когда задаются несколько узлов или когда одно имя узла переводится в несколько адресов, все узлы и адреса перебираются по порядку, пока подключение не будет установлено. Если ни один из адресов не будет доступен, произойдёт сбой подключения. Если подключение устанавливается успешно, но происходит ошибка аутентификации, остальные узлы в списке не перебираются.

Если используется файл паролей, в нём можно задать разные пароли для разных узлов. Все остальные параметры подключения будут одинаковыми для всех узлов; например, нельзя задать для разных узлов различные имена пользователей.

27.1.15. Ключевые слова-параметры

Ключевые слова-параметры, распознаваемые в настоящее время, следующие:

– host – Имя компьютера для подключения. Если это имя выглядит как указание абсолютного пути, выбирается подключение через Unix-сокет, а не через TCP/IP, и данное значение определяет имя каталога, содержащего файл сокета. По умолчанию, если параметр host отсутствует или пуст, выполняется подключение к Unix-сокету в /tmp.

Также принимается разделённый запятыми список имён узлов; при этом данные имена будут перебираться по порядку.

– hostaddr – Числовой IP-адрес компьютера для подключения. Он должен быть представлен в стандартном формате адресов IPv4. Если машина поддерживает IPv6, можно использовать и адреса IPv6. Если в этом параметре передана непустая строка, для подключения всегда используется TCP/IP. В отсутствие этого параметра целевой IP-адрес будет получен из имени,

заданного в параметре host, либо если в host задан IP-адрес, будет использоваться непосредственно он.

Использование hostaddr позволяет приложению обойтись без разрешения имён, что может быть важно для приложений с жёсткими временными ограничениями. Применяются следующие правила:

– Если адрес host задаётся без hostaddr, осуществляется разрешение имени. (При использовании PQconnectPoll разрешение производится, когда PQconnectPoll рассматривает это имя в первый раз, и может заблокировать PQconnectPoll на неопределённое время.)

– Если указан hostaddr, а host не указан, тогда значение hostaddr даёт сетевой адрес сервера. Попытка подключения завершится неудачей, если метод аутентификации требует наличия имени компьютера.

– Если указаны как host, так и hostaddr, тогда значение hostaddr даёт сетевой адрес сервера, а значение host игнорируется, если только метод аутентификации его не потребует. В таком случае оно будет использоваться в качестве имени компьютера.

Аутентификация может завершиться неудачей, если host не является именем сервера с сетевым адресом hostaddr. Когда указывается и host, и hostaddr, соединение в файле паролей идентифицируется по значению host.

Также принимается разделённый запятыми список значений hostaddr, при этом данные узлы будут перебираться по порядку. Вместо пустого элемента в этом списке будет подставлено имя соответствующего узла или, если и оно не определено, имя узла по умолчанию.

Если не указаны ни имя компьютера, ни его адрес, libpq будет производить подключение, используя локальный Unix-сокет; в Windows и в системах, не поддерживающих Unix-сокеты, она будет пытаться подключиться к localhost.

– port – Номер порта, к которому нужно подключаться на сервере, либо расширение имени файла сокета для соединений через Unix-сокеты. Если в параметрах host или hostaddr задано несколько серверов, в данном параметре может задаваться через запятую список портов такой же длины, либо может указываться один номер порта для всех узлов. Пустая строка или пустой элемент в списке через запятую воспринимается как номер порта по умолчанию.

– dbname – Имя базы данных. По умолчанию оно совпадает с именем пользователя. В определённых контекстах это значение проверяется на соответствие расширенным форматам.

– user – Имя пользователя PG360, используемое для подключения. По умолчанию используется то же имя, которое имеет в ОС пользователь, от лица которого выполняется приложение.

– password – Пароль, используемый в случае, когда сервер требует аутентификации по паролю.

– passfile – Задаёт имя файла, в котором будут храниться пароли.

– channel_binding – Этот параметр определяет режим использования связывания каналов клиентом. Вариант require означает, что для соединения должно задействоваться связывание каналов, prefer – клиент будет выбирать связывание, если оно поддерживается, а disable предотвращает использование этого механизма. По умолчанию подразумевается вариант prefer, в противном случае – disable.

– connect_timeout – Максимальное время ожидания подключения (задаётся десятичным целым числом, например: 10). При нуле, отрицательном или неопределённом значении ожидание будет бесконечным. Минимальный допустимый тайм-аут равен 2 секундам; таким образом, значение 1 воспринимается как 2. Этот тайм-аут применяется для каждого отдельного IP-адреса или имени сервера.

– client_encoding – Этим устанавливается конфигурационный параметр client_encoding для данного подключения. В дополнение к значениям, которые принимает соответствующий параметр сервера, можно использовать значение auto. В этом случае правильная кодировка определяется на основе текущей локали на стороне клиента.

– options – Задаёт параметры командной строки, которые будут отправлены серверу при установлении соединения.

– application_name – Устанавливает значение для конфигурационного параметра application_name.

– fallback_application_name – Устанавливает альтернативное значение для конфигурационного параметра application_name. Это значение будет использоваться, если для параметра application_name не было передано никакого значения с помощью параметров подключения или переменной системного окружения PGAPPNAME.

– keepalives – Управляет использованием сообщений keepalive протокола TCP на стороне клиента. Значение по умолчанию равно 1, что означает использование сообщений. 0, если эти сообщения не нужны. Для соединений, установленных через Unix-сокеты, этот параметр игнорируется.

– keepalives_idle – Управляет длительностью периода отсутствия активности, выраженного числом секунд, по истечении которого TCP должен отправить сообщение keepalive серверу. При значении 0 действует системная величина.

– keepalives_interval – Задаёт количество секунд, по прошествии которых сообщение keepalive протокола TCP, получение которого не подтверждено сервером, должно быть отправлено повторно. При значении 0 действует системная величина.

– keepalives_count – Задаёт количество сообщений keepalive протокола TCP, которые могут быть потеряны, прежде чем соединение клиента с сервером будет признано неработающим. Нулевое значение этого параметра указывает, что будет использоваться системное значение по умолчанию.

– tcp_user_timeout – Управляет длительностью интервала (в миллисекундах), в течение которого данные могут оставаться неподтверждёнными, прежде чем соединение будет принудительно закрыто. При значении 0 действует системная величина.

– tty – Игнорируется.

– replication – Этот параметр определяет, должно ли подключение использовать протокол репликации вместо обычного протокола.

Поддерживаются следующие значения этого параметра, без учёта регистра:

true, on, yes, 1 – Подключение осуществляется в режиме физической репликации.

database – Подключение осуществляется в режиме логической репликации, целевая база данных задаётся параметром dbname.

false, off, no, 0 – Подключение выполняется в обычном режиме; это поведение по умолчанию.

В режиме физической или логической репликации может использоваться только протокол простых запросов.

– sslmode – Этот параметр определяет, будет ли согласовываться с сервером защищённое SSL-соединение по протоколу TCP/IP, и если да, то в какой очередности. Всего предусмотрено шесть режимов:

disable – следует пытаться установить только соединение без использования SSL

allow – сначала следует попытаться установить соединение без использования SSL; если попытка будет неудачной, нужно попытаться установить SSL-соединение

prefer (по умолчанию) – сначала следует попытаться установить SSL-соединение; если попытка будет неудачной, нужно попытаться установить соединение без использования SSL

require – следует попытаться установить только SSL-соединение. Если присутствует файл корневого центра сертификации, то нужно верифицировать сертификат таким же способом, как будто был указан параметр verify-ca

verify-ca – следует попытаться установить только SSL-соединение, при этом проконтролировать, чтобы сертификат сервера был выпущен доверенным удостоверяющим центром (CA)

verify-full – следует попытаться установить только SSL-соединение, при этом проконтролировать, чтобы сертификат сервера был выпущен доверенным центром сертификации (CA) и чтобы имя запрошенного сервера соответствовало имени в сертификате

sslmode игнорируется при использовании Unix-сокетов.

– requiressl – Использовать этот параметр не рекомендуется, в качестве замены предлагается установить sslmode.

Если установлено значение 1, то требуется SSL-соединение с сервером (это эквивалентно sslmode require). libpq в таком случае откажется подключаться, если сервер не принимает SSL–соединений. Если установлено значение 0 (по умолчанию), тогда libpq будет согласовывать тип подключения с сервером (эквивалентно sslmode prefer).

– sslcompression – Если установлено значение 1, данные, передаваемые через SSL-соединения, будут сжиматься. Если установлено значение 0 (по умолчанию), сжатие будет отключено. Этот параметр игнорируется, если установлено подключение без SSL.

Сжатие SSL в настоящее время считается небезопасным, и использовать его уже не рекомендуется. В OpenSSL 1.1.0 сжатие отключено по умолчанию, к тому же во многих дистрибутивах оно отключается и с более ранними версиями. А если сервер не поддерживает сжатие, включение этого параметра не окажет никакого влияния.

Если вопросы безопасности не стоят на первом месте, сжатие может ускорить передачу данных, когда узким местом является сеть. Отключение сжатия может улучшить время отклика и пропускную способность, если ограничивающим фактором является производительность CPU.

– sslcert – Этот параметр предписывает имя файла для SSL-сертификата клиента.

– sslkey – Этот параметр предписывает местоположение секретного ключа, используемого для сертификата клиента. Он может либо указывать имя файла, которое будет использоваться вместо имени по умолчанию, либо он может указывать ключ, полученный от внешнего

«криптомодуля» (криптомодули – это загружаемые модули OpenSSL). Спецификация внешнего криптомодуля должна состоять из имени модуля и ключевого идентификатора, зависящего от конкретного модуля, разделённых двоеточием.

– sslpassword – Этот параметр задаёт пароль для секретного ключа, указанного в sslkey, что позволяет хранить на диске закрытые ключи клиентских сертификатов в зашифрованном виде, даже когда применять интерактивный ввод пароля непрактично.

– sslrootcert – Этот параметр указывает имя файла, содержащего SSL-сертификаты, выданные удостоверяющим центром (CA). Если файл существует, сертификат сервера будет проверен на предмет его подписания одним из этих центров. Имя по умолчанию root.crt.

– sslcrl – Этот параметр указывает имя файла, содержащего список отозванных серверных сертификатов (CRL) для SSL. Сертификаты, перечисленные в этом файле, если он существует, будут отвергаться при попытке установить подлинность сертификата сервера. Имя по умолчанию такое root.crl.

– requirepeer – Этот параметр указывает имя пользователя ОС, предназначенное для сервера.

– ssl_min_protocol_version – определяет, с какой минимальной версией протокола SSL/TLS может быть установлено подключение.

– ssl_max_protocol_version – определяет, с какой максимальной версией протокола SSL/TLS может быть установлено подключение.

– service – Имя сервиса, используемое для задания дополнительных параметров. Оно указывает имя сервиса в файле pg_service.conf, который содержит дополнительные параметры подключения.

– target_session_attrs – Если этот параметр равен read-write, по умолчанию будут приемлемы только подключения, допускающие транзакции на чтение/запись. При успешном подключении будет отправлен запрос SHOW transaction_read_only; если он вернёт on, соединение будет закрыто. Если в строке подключения указано несколько серверов, будут перебираться остальные серверы, как и при неудачной попытке подключения. Со значением по умолчанию (any) приемлемыми будут все подключения.

27.12. Функции, описывающие текущее состояние подключения

Эти функции могут использоваться для опроса состояния объекта, описывающего существующее подключение к базе данных.

Следующие функции возвращают значения параметров, которые были установлены в момент подключения. Эти значения не изменяются во время жизни соединения. Если используется строка соединения с несколькими узлами, значения PQhost, PQport и PQpass могут меняться, если с тем же объектом PGconn устанавливается новое соединение. Другие значения не меняются на протяжении жизни объекта PGconn.

– PQdb – Возвращает имя базы данных, с которой установлено соединение.

```
char *PQdb(const PGconn *conn);
```

– PQuser – Возвращает имя пользователя, который установил соединение.

```
char *PQuser(const PGconn *conn);
```

– PQpass – Возвращает пароль, использованный для подключения.

```
char *PQpass(const PGconn *conn);
```

PQpass возвратит либо пароль, указанный в параметрах подключения, либо пароль, полученный из файла паролей (в случае отсутствия первого). Во втором случае, если в параметрах подключения было указано несколько узлов, полагаться на результат PQpass нельзя, пока соединение не будет установлено. Состояние подключения позволяет проверить функция PQstatus.

– PQhost – Возвращает имя сервера для активного соединения. Это может быть имя компьютера, IP– адрес или путь к каталогу, если подключение установлено через сокет Unix. (Признаком подключения к сокету будет абсолютный путь, который начинается с /.)

```
char *PQhost(const PGconn *conn);
```

Если в параметрах подключения был задан и host, и hostaddr, функция PQhost выдаст содержимое host. Если был задан только hostaddr, возвращается это значение. Если в параметрах подключения было задано несколько узлов, PQhost возвращает адрес узла, с которым фактически установлено соединение.

PQhost возвращает NULL, если аргумент *conn* равен NULL. Иначе в случае ошибки при получении информации об узле (это возможно, если соединение не установлено до конца или произошла ошибка) она возвращает пустую строку.

Если в параметрах подключения указаны несколько узлов, на результат PQhost нельзя полагаться, пока соединение не будет установлено. Проверить состояние соединения позволяет функция PQstatus.

– PQhostaddr – Возвращает IP-адрес сервера для активного соединения. Это может быть адрес, в который разрешилось имя компьютера, или IP-адрес, переданный в параметре hostaddr.

```
char *PQhostaddr(const PGconn *conn);
```

PQhostaddr возвращает NULL, если аргумент *conn* равен NULL. Иначе в случае ошибки при получении информации об узле (это возможно, если соединение не установлено до конца или произошла ошибка) она возвращает пустую строку.

– PQport – Возвращает номер порта активного соединения.

```
char *PQport(const PGconn *conn);
```

Если в параметрах подключения было задано несколько портов, PQport возвращает порт, с которым фактически установлено соединение.

PQport возвращает NULL, если аргумент *conn* равен NULL. Иначе в случае ошибки при получении информации о порте (это возможно, если соединение не установлено до конца или произошла ошибка) она возвращает пустую строку.

Если в параметрах подключения указаны несколько портов, на результат PQport нельзя полагаться, пока соединение не будет установлено. Проверить состояние соединения позволяет функция PQstatus.

– PQtty – Возвращает имя отладочного терминала (TTY), связанного с данным соединением. (Это устаревшая функция, поскольку сервер более не обращает внимания на установку TTY, но она остаётся для обеспечения обратной совместимости.)

```
char *PQtty(const PGconn *conn);
```

– PQoptions – Возвращает параметры командной строки, переданные в запросе на подключение.

```
char *PQoptions(const PGconn *conn);
```

Следующие функции возвращают данные статуса, который может измениться в процессе выполнения операций на объекте PGconn.

– PQstatus – Возвращает состояние подключения.

```
ConnStatusType PQstatus(const PGconn *conn);
```

Статус может принимать одно из ряда значений. Однако только два из них видны извне процедуры асинхронного подключения: CONNECTION_OK и CONNECTION_BAD. Успешное подключение к базе данных имеет статус CONNECTION_OK. О неудачной попытке подключения сигнализирует статус CONNECTION_BAD. Обычно статус OK остаётся таковым до вызова PQfinish, но в случае ошибки соединения статус может смениться на CONNECTION_BAD преждевременно. В таком случае приложение может попытаться восстановить подключение, вызвав Pqreset.

– PQtransactionStatus – Возвращает текущий статус сервера, отражающий процесс выполнения транзакций.

```
PGTransactionStatusType PQtransactionStatus(const PGconn *conn);
```

Статус может быть одним из PQTRANS_IDLE (в настоящее время не занят обработкой транзакции), PQTRANS_ACTIVE (команда в процессе обработки), PQTRANS_INTRANS (не выполняет работу, но находится в рамках действительной транзакции) или PQTRANS_INERROR (не выполняет работу, но находится в рамках транзакции, завершившейся сбоем). Статус принимает значение PQTRANS_UNKNOWN, если соединение не работает. Статус принимает значение PQTRANS_ACTIVE только тогда, когда запрос был отправлен серверу, но ещё не завершён.

– PQparameterStatus – Отыскивает текущее значение параметра сервера.

```
const char *PQparameterStatus(const PGconn *conn, const char *paramName);
```

Значения определённых параметров сервер сообщает автоматически в начале процедуры подключения или тогда, когда их значения изменяются. PQparameterStatus можно использовать, чтобы запросить эти значения. Функция возвращает текущее значение параметра, если оно известно, или NULL, если параметр неизвестен.

– PQprotocolVersion – Запрашивает протокол, используемый между клиентом и сервером.

```
int PQprotocolVersion(const PGconn *conn);
```

– PQserverVersion – Возвращает целочисленное представление версии сервера.

```
int PQserverVersion(const PGconn *conn);
```

– PQerrorMessage – Возвращает сообщение об ошибке, наиболее недавно сгенерированное операцией, выполненной в рамках текущего подключения.

```
char *PQerrorMessage(const PGconn *conn);
```

– PQsocket – Получает номер файлового дескриптора для сокета соединения с сервером. Действительный дескриптор будет больше или равен 0; значение -1 показывает, что в данный момент не открыто ни одного соединения с сервером. (Значение не изменится во время обычной работы, но может измениться во время установки или переустановки подключения.)

```
int PQsocket(const PGconn *conn);
```

– PQbackendPID – Возвращает ID (PID) серверного процесса, обрабатывающего это подключение.

```
int PQbackendPID(const PGconn *conn);
```

– PQconnectionNeedsPassword – Возвращает true (1), если метод аутентификации соединения требовал пароля, однако он не был предоставлен. Возвращает false (0), если пароль не требовался.

```
int PQconnectionNeedsPassword(const PGconn *conn);
```

– PQconnectionUsedPassword – Возвращает true (1), если метод аутентификации соединения использовал пароль. Возвращает false (0) в противном случае.

```
int PQconnectionUsedPassword(const PGconn *conn);
```

Следующие функции возвращают информацию, относящуюся к SSL. Эта информация обычно не меняется после того, как подключение установлено.

– PQsslInUse – Возвращает true (1), если для текущего подключения используется SSL, и false (0) в противном случае.

```
int PQsslInUse(const PGconn *conn);
```

– PqsslAttribute – Возвращает связанную с SSL информацию о соединении.

```
const char *PQsslAttribute(const PGconn *conn, const char *attribute_name);
```

Список доступных атрибутов зависит от применяемой библиотеки SSL и типа подключения. Если атрибут недоступен, возвращается NULL.

Обычно доступны следующие атрибуты:

– library – Имя используемой реализации SSL.

– protocol – Применяемая версия SSL/TLS.

– key_bits – Число бит в ключе, используемом алгоритмом шифрования.

– cipher – Краткое имя применяемого комплекта шифров. Эти имена могут быть разными в разных реализациях SSL.

– compression – Возвращает on, если используется сжатие SSL, или off в противном случае.

– PQsslAttributeNames – Возвращает массив имён доступных атрибутов SSL. Завершается массив указателем NULL.

```
const char * const * PQsslAttributeNames(const PGconn *conn);
```

– PQsslStruct – Возвращает указатель на специфичный для реализации SSL объект, описывающий подключение.

```
void *PQsslStruct(const PGconn *conn, const char *struct_name);
```

– PQgetssl – Возвращает структуру SSL, использовавшуюся в соединении, или null, если SSL не используется.

```
void *PQgetssl(const PGconn *conn);
```

27.13. Функции для исполнения команд

После того как соединение с сервером было успешно установлено, функции, описанные в этом разделе, используются для выполнения SQL-запросов и команд.

27.13.1. Главные функции

– Pqexec – Передаёт команду серверу и ожидает результата.

```
PGresult *PQexec(PGconn *conn, const char *command);
```

Возвращает указатель на PGresult или, возможно, пустой указатель (null).

– PQexecParams – Отправляет команду серверу и ожидает результата. Имеет возможность передать параметры отдельно от текста SQL-команды.

```
PGresult *PQexecParams(PGconn *conn,
const char *command, int nParams,
const Oid *paramTypes,
const char * const *paramValues, const int *paramLengths,
const int *paramFormats, int resultFormat);
```

Параметры функции следующие:

conn – Объект, описывающий подключение, через которое пересылается команда.

command – Строка SQL-команды, которая должна быть выполнена.

nParams – Число предоставляемых параметров.

paramTypes[] – Предписывает, посредством OID, типы данных, которые должны быть назначены параметрам. Если значение *paramTypes* равно NULL или какой-либо отдельный элемент в массиве равен нулю, тогда сервер самостоятельно определит тип данных для параметра точно таким же образом, как он сделал бы для литеральной строки, тип которой не указан.

paramValues[] – Указывает фактические значения параметров. Нулевой указатель в этом массиве означает, что соответствующий параметр равен null; в противном случае указатель указывает на текстовую строку, завершающуюся нулевым символом (для текстового формата), или на двоичные данные в формате, которого ожидает сервер (для двоичного формата).

paramLengths[] – Указывает фактические длины данных для параметров, представленных в двоичном формате. Он игнорируется для параметров, имеющих значение null, и для параметров, представленных в текстовом формате. Указатель на массив может быть нулевым, когда нет двоичных параметров.

paramFormats[] – Указывает, являются ли параметры текстовыми (необходимо поместить нуль в элемент массива, соответствующий такому параметру) или двоичными (необходимо поместить единицу в элемент массива, соответствующий такому параметру). Если указатель на массив является нулевым, тогда все параметры считаются текстовыми строками.

– *resultFormat* – Требует указать ноль, чтобы получить результаты в текстовом формате, или единицу, чтобы получить результаты в двоичном формате.

– PQprepare – Отправляет запрос, чтобы создать подготовленный оператор с конкретными параметрами, и ожидает завершения.

```
PGresult *PQprepare(PGconn *conn, const char *stmtName, const char
*query, int nParams,const Oid *paramTypes);
```

– PQexecPrepared – Отправляет запрос на исполнение подготовленного оператора с данными параметрами и ожидает результата.

```
PGresult *PQexecPrepared(PGconn *conn,
const char *stmtName, int nParams,
const char * const *paramValues, const int *paramLengths,
const int *paramFormats, int resultFormat);
```

– PQdescribePrepared – Передаёт запрос на получение информации об указанном подготовленном операторе и ожидает завершения.

```
PGresult *PQdescribePrepared(PGconn *conn, const char *stmtName);
```

– PQdescribePortal – Передаёт запрос на получение информации об указанном портале и ожидает завершения.

```
PGresult *PQdescribePortal(PGconn *conn, const char *portalName);
```

– PQresultStatus – Возвращает статус результата выполнения команды.

```
ExecStatusType PQresultStatus(const PGresult *res);
```

PQresultStatus может возвращать одно из следующих значений:

PGRES_EMPTY_QUERY – Строка, отправленная серверу, была пустой.

PGRES_COMMAND_OK – Успешное завершение команды, не возвращающей никаких данных.

PGRES_TUPLES_OK – Успешное завершение команды, возвращающей данные (такой, как SELECT или SHOW).

PGRES_COPY_OUT – Начат перенос данных Copy Out (с сервера).

PGRES_COPY_IN – Начат перенос данных Copy In (на сервер).

PGRES_BAD_RESPONSE – Ответ сервера не был распознан.

PGRES_NONFATAL_ERROR – Произошла не фатальная ошибка (уведомление или предупреждение).

PGRES_FATAL_ERROR – Произошла фатальная ошибка.

PGRES_COPY_BOTH – Начат перенос данных Copy In/Out (на сервер и с сервера). Эта функция в настоящее время используется только для потоковой репликации, поэтому такой статус не должен иметь место в обычных приложениях.

PGRES_SINGLE_TUPLE – Структура PGresult содержит только одну результирующую строку, возвращённую текущей командой. Этот статус имеет место только тогда, когда для данного запроса был выбран режим построчного вывода.

Если статус результата PGRES_TUPLES_OK или PGRES_SINGLE_TUPLE, тогда для извлечения строк, возвращённых запросом, можно использовать функции, описанные ниже. Команда SELECT, даже когда она не извлекает ни одной строки, показывает PGRES_TUPLES_OK. PGRES_COMMAND_OK предназначен для команд, которые никогда не возвращают строки (INSERT или UPDATE без использования предложения RETURNING и др.). Ответ PGRES_EMPTY_QUERY может указывать на наличие ошибки в клиентском программном обеспечении.

– PQresStatus – Преобразует значение перечислимого типа, возвращённое функцией PQresultStatus, в строковую константу, описывающую код статуса. Вызывающая функция не должна освобождать память, на которую указывает возвращаемый указатель.

```
char *PQresStatus(ExecStatusType status);
```

– PQresultErrorMessage – Возвращает сообщение об ошибке, связанное с командой, или пустую строку, если ошибки не произошло.

```
char *PQresultErrorMessage(const PGresult *res);
```

Если произошла ошибка, то возвращённая строка будет включать завершающий символ новой строки. Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель. Она будет освобождена, когда соответствующий указатель PGresult будет передан функции PQclear.

– PQresultVerboseErrorMessage – Возвращает переформатированную версию сообщения об ошибке, связанного с объектом PGresult.

```
char *PQresultVerboseErrorMessage(const PGresult *res,
PGVerbosity verbosity, PGContextVisibility show_context);
```

– PQresultErrorField – Возвращает индивидуальное поле из отчёта об ошибке.

```
char *PQresultErrorField(const PGresult *res, int fieldcode);
```

fieldcode это идентификатор поля ошибки. Если PGresult не содержит ошибки или предупреждения или не включает указанное поле, то возвращается NULL. Значения полей обычно не включают завершающий символ новой строки. Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель. Она будет освобождена, когда соответствующий указатель PGresult будет передан функции PQclear.

Доступны следующие коды полей:

PG_DIAG_SEVERITY – Поле может содержать ERROR, FATAL или PANIC (в сообщении об ошибке) либо WARNING, NOTICE, DEBUG, INFO или LOG (в сообщении-уведомлении) либо локализованный перевод одного из этих значений. Присутствует всегда.

PG_DIAG_SEVERITY_NONLOCALIZED – Поле может содержать ERROR, FATAL или PANIC (в сообщении об ошибке) либо WARNING, NOTICE, DEBUG, INFO или LOG (в сообщении-уведомлении). Это поле подобно PG_DIAG_SEVERITY, но его содержимое никогда не переводится.

PG_DIAG_SQLSTATE – Код ошибки в соответствии с соглашением о кодах SQLSTATE. Код SQLSTATE идентифицирует тип случившейся ошибки; он может использоваться клиентскими приложениями, чтобы выполнять конкретные операции (такие, как обработка ошибок) в ответ на конкретную ошибку базы данных. Это поле не подлежит локализации. Оно всегда присутствует.

PG_DIAG_MESSAGE_PRIMARY – Главное сообщение об ошибке, предназначенное для прочтения пользователем. Как правило составляет всего одну строку. Это поле всегда присутствует.

PG_DIAG_MESSAGE_DETAIL – Необязательное дополнительное сообщение об ошибке, передающее более детальную информацию о проблеме. Может занимать несколько строк.

PG_DIAG_MESSAGE_HINT – Подсказка: необязательное предположение о том, что можно сделать в данной проблемной ситуации. Оно должно отличаться от детальной информации в том смысле, что оно предлагает совет (возможно, и неподходящий), а не просто факты. Может занимать несколько строк.

PG_DIAG_STATEMENT_POSITION – Строка, содержащая десятичное целое число, указывающее позицию расположения ошибки в качестве индекса в оригинальной строке оператора. Первый символ имеет позицию 1, при этом позиции измеряются в символах а не в байтах.

PG_DIAG_INTERNAL_POSITION – Это поле определяется точно так же, как и поле PG_DIAG_STATEMENT_POSITION, но оно используется, когда позиция местонахождения ошибки относится к команде, сгенерированной внутренними модулями, а не к команде, представленной клиентом. Когда появляется это поле, то всегда появляется и поле PG_DIAG_INTERNAL_QUERY.

PG_DIAG_INTERNAL_QUERY – Текст команды, сгенерированной внутренними модулями, завершившейся сбоем. Это мог бы быть, например, SQL-запрос, выданный функцией на языке PL/pgSQL.

PG_DIAG_CONTEXT – Характеристика контекста, в котором произошла ошибка. В настоящее время она включает вывод стека вызовов активных функций процедурного языка и запросов, сгенерированных внутренними модулями. Стек выводится по одному элементу в строке, при этом первым идет самый последний из элементов.

PG_DIAG_SCHEMA_NAME – Если ошибка была связана с конкретным объектом базы данных, то в это поле будет записано имя схемы, содержащей данный объект.

PG_DIAG_TABLE_NAME – Если ошибка была связана с конкретной таблицей, то в это поле будет записано имя таблицы.

PG_DIAG_COLUMN_NAME – Если ошибка была связана с конкретным столбцом таблицы, то в это поле будет записано имя столбца.

PG_DIAG_DATATYPE_NAME – Если ошибка была связана с конкретным типом данных, то в это поле будет записано имя типа данных.

PG_DIAG_CONSTRAINT_NAME – Если ошибка была связана с конкретным ограничением, то в это поле будет записано имя ограничения.

PG_DIAG_SOURCE_FILE – Имя файла, содержащего позицию в исходном коде, для которой было выдано сообщение об ошибка.

PG_DIAG_SOURCE_LINE – Номер строки той позиции в исходном коде, для которой было выдано сообщение об ошибке.

PG_DIAG_SOURCE_FUNCTION – Имя функции в исходном коде, сообщающей об ошибке.

– PQclear – Освобождает область памяти, связанную с PGresult. Результат выполнения каждой команды должен быть освобождён с помощью PQclear, когда он больше не нужен.

```
void PQclear(PGresult *res);
```

27.13.2. Извлечение информации, связанной с результатом запроса

Эти функции служат для извлечения информации из объекта PGresult, который представляет результат успешного запроса (то есть такого, который имеет статус PGRES_TUPLES_OK или PGRES_SINGLE_TUPLE). Их также можно использовать для извлечения информации об успешной операции DESCRIBE: результат этой операции содержит всю ту же самую информацию о столбцах, которая была бы получена при реальном исполнении запроса, но не содержит ни одной строки. Для объектов, имеющих другие значения статуса, эти функции будут действовать таким образом, как будто результат не содержит ни одной строки и ни одного столбца.

– PQntuples – Возвращает число строк (кортежей) в полученной выборке.

```
int PQntuples(const PGresult *res);
```

– PQnfields – Возвращает число столбцов (полей) в каждой строке полученной выборки.

```
int PQnfields(const PGresult *res);
```

– PQfname – Возвращает имя столбца, соответствующего данному номеру столбца. Номера столбцов начинаются с 0. Вызывающая функция не должна напрямую освобождать память, на

которую указывает возвращаемый указатель. Она будет освобождена, когда соответствующий указатель на PGresult будет передан функции PQclear.

```
char *PQfname(const PGresult *res,int column_number);
```

Если номер столбца выходит за пределы допустимого диапазона, то возвращается NULL.

– PQfnumber – Возвращает номер столбца, соответствующий данному имени столбца.

```
int PQfnumber(const PGresult *res,const char *column_name);
```

Если данное имя не совпадает с именем ни одного из столбцов, то возвращается -1.

– PQftable -Возвращает OID таблицы, из которой был получен данный столбец. Номера столбцов начинаются с 0.

```
Oid PQftable(const PGresult *res, int column_number);
```

В следующих случаях возвращается InvalidOid: если номер столбца выходит за пределы допустимого диапазона; если указанный столбец не является простой ссылкой на столбец таблицы.

Тип данных Oid и константа InvalidOid будут определены, когда подключен заголовочный файл для libpq. Они будут принадлежать к одному из целочисленных типов.

– PQftablecol – Возвращает номер столбца (в пределах его таблицы) для указанного столбца в полученной выборке. Номера столбцов в полученной выборке начинаются с 0, но столбцы в таблице имеют ненулевые номера.

```
int PQftablecol(const PGresult *res,int column_number);
```

В следующих случаях возвращается ноль: если номер столбца выходит за пределы допустимого диапазона; если указанный столбец не является простой ссылкой на столбец таблицы.

– PQfformat – Возвращает код формата, показывающий формат данного столбца. Номера столбцов начинаются с 0.

```
int PQfformat(const PGresult *res, int column_number);
```

Значение кода формата, равное нулю, указывает на текстовое представление данных, в то время, как значение, равное единице, означает двоичное представление.

– PQftype – Возвращает тип данных, соответствующий данному номеру столбца. Возвращаемое целое значение является внутренним номером OID для этого типа. Номера столбцов начинаются с 0.

```
Oid PQftype(const PGresult *res,int column_number);
```

– PQfmod – Возвращает модификатор типа для столбца, соответствующего данному номеру. Номера столбцов начинаются с 0.

```
int PQfmod(const PGresult *res, int column_number);
```

– PQfsize – Возвращает размер в байтах для столбца, соответствующего данному номеру. Номера столбцов начинаются с 0.

```
int PQfsize(const PGresult *res, int column_number);
```

PQfsize возвращает размер пространства, выделенного для этого столбца в строке базы данных, другими словами, это размер внутреннего представления этого типа данных на сервере. Отрицательное значение говорит о том, что тип данных имеет переменную длину.

– PQbinaryTuples – Возвращает 1, если PGresult содержит двоичные данные, или 0, если данные текстовые.

```
int PQbinaryTuples(const PGresult *res);
```

Эта функция не рекомендуется к использованию (за исключением применения в связи с командой COPY), поскольку один и тот же PGresult может содержать в некоторых столбцах текстовые данные, а в остальных – двоичные. Предпочтительнее использовать PQfformat. PQbinaryTuples возвращает 1, только если все столбцы в выборке являются двоичными (код формата 1).

– PQgetvalue – Возвращает значение одного поля из одной строки, содержащейся в PGresult. Номера строк и столбцов начинаются с 0. Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель. Она будет освобождена, когда соответствующий указатель на PGresult будет передан функции PQclear.

```
char *PQgetvalue(const PGresult *res, int row_number,int column_number);
```

Для данных в текстовом формате значение, возвращаемое функцией PQgetvalue, является значением поля, представленным в виде символьной строки с завершающим нулевым символом. Для данных в двоичном формате используется двоичное представление значения. Оно определяется функциями typsend и typreceive для конкретного типа данных.

Пустая строка возвращается в том случае, когда поле содержит null. Чтобы отличить значения null от пустых строковых значений, необхоимо воспользоваться функцией PQgetisnull.

Указатель, возвращаемый функцией PQgetvalue, указывает на область памяти внутри структуры PGresult. Модифицировать данные, на которые указывает этот указатель, не следует. Вместо этого нужно явно скопировать данные в другую область памяти, если предполагается использовать их за рамками жизненного цикла структуры Pgresult.

– PQgetisnull – Проверяет поле на предмет отсутствия значения (null). Номера строк и столбцов начинаются с 0.

```
int PQgetisnull(const PGresult *res, int row_number, int column_number);
```

Эта функция возвращает 1, если значение в поле отсутствует (null), и 0, если поле содержит отличное от null значение.

– PQgetlength – Возвращает фактическую длину значения поля в байтах. Номера строк и столбцов начинаются с 0.

```
int PQgetlength(const PGresult *res, int row_number,int column_number);
```

– PQnparams – Возвращает число параметров подготовленного оператора.

```
int PQnparams(const PGresult *res);
```

– PQparamtype – Возвращает тип данных для указанного параметра оператора. Номера параметров начинаются с 0.

```
Oid PQparamtype(const PGresult *res, int param_number);
```

– PQprint – Выводит все строки и, по выбору, имена столбцов в указанный поток вывода.

```
void PQprint(FILE *fout,/* поток вывода */ const PGresult *res,const PQprintOpt *po);
```

27.13.3. Получение другой информации о результате

Эти функции используются для получения остальной информации из объектов Pgresult:

– PQcmdStatus – Возвращает дескриптор статуса для SQL-команды, которая сгенерировала PGresult.

```
char *PQcmdStatus(PGresult *res);
```

– PQcmdTuples – Возвращает число строк, которые затронула SQL-команда.

```
char *PQcmdTuples(PGresult *res);
```

Эта функция возвращает строковое значение, содержащее число строк, которые затронул SQL– оператор, сгенерировавший данный PGresult. Эту функцию можно использовать только сразу после выполнения команд SELECT, CREATE TABLE AS, INSERT, UPDATE, DELETE, MOVE, FETCH или COPY, а также после оператора EXECUTE, выполнившего подготовленный запрос, содержащий команды INSERT, UPDATE или DELETE.

– PQoidValue – Возвращает OID вставленной строки, если SQL-команда была командой INSERT, которая вставила ровно одну строку в таблицу, имеющую идентификаторы OID, или командой EXECUTE, которая выполнила подготовленный запрос, содержащий соответствующий оператор INSERT. В противном случае эта функция возвращает InvalidOid. Эта функция также возвратит InvalidOid, если таблица, затронутая командой INSERT, не содержит идентификаторов OID.

```
Oid PQoidValue(const PGresult *res);
```

27.13.4. Экранирование строковых значений для включения в SQL– команды

– PQescapeLiteral – Экранирует строковое значение для использования внутри SQL-команды.

```
char *PQescapeLiteral(PGconn *conn, const char *str, size_t length);
```

В случае ошибки PQescapeLiteral возвращает NULL, и в объект *conn* помещается соответствующее сообщение.

– PQescapeIdentifier – Экранирует строку, предназначенную для использования в качестве идентификатора SQL.

```
char *PQescapeIdentifier(PGconn *conn, const char *str, size_t length);
```

В случае ошибки PQescapeIdentifier возвращает NULL, и в объект *conn* помещается соответствующее сообщение.

– PQescapeStringConn – Экранирует строковые литералы наподобие PqescapeLiteral. Но, в отличие от PQescapeLiteral, за предоставление буфера надлежащего размера отвечает вызывающая функция.

```
size_t PQescapeStringConn(PGconn *conn, char *to, const char *from, size_t length, int *error);
```

PQescapeStringConn возвращает число байт, записанных по адресу *to*, не включая завершающий нулевой байт.

– PQescapeByteaConn – Экранирует двоичные данные для их использования внутри SQL-команды с типом данных bytea. Как и в случае с PQescapeStringConn, эта функция применяется только тогда, когда данные вставляются непосредственно в строку SQL-команды.

```
unsigned char *PQescapeByteaConn(PGconn *conn,
const unsigned char *from, size_t from_length,
size_t *to_length);
```

Параметр *from* указывает на первый байт строки, которая должна экранироваться, а параметр *from_length* задаёт число байт в этой двоичной строке. (Завершающий нулевой байт не нужен и не учитывается.) Параметр *to_length* указывает на переменную, которая будет содержать длину результирующей экранированной строки. Эта длина включает завершающий нулевой байт результирующей строки.

PQescapeByteaConn возвращает экранированную версию двоичной строки, на которую указывает параметр *from*, и размещает её в памяти, распределённой с помощью malloc(). Эта память должна быть освобождена с помощью функции PQfreemem(), когда результирующая строка больше не нужна. В возвращаемой строке все специальные символы заменены так, чтобы синтаксический анализатор литеральных строк PG360 и функция ввода для типа bytea могли обработать их надлежащим образом. Завершающий нулевой байт также добавляется. Одинарные кавычки, которые должны окружать строковые литералы PG360, не являются частью результирующей строки.

В случае ошибки возвращается нулевой указатель, и соответствующее сообщение об ошибке записывается в объект *conn*. В настоящее время единственной возможной ошибкой может быть нехватка памяти для результирующей строки.

– PQunescapeBytea – Преобразует строковое представление двоичных данных в двоичные данные – является обратной функцией к функции PQescapeBytea. Она нужна, когда данные типа bytea извлекаются в текстовом формате, но не когда они извлекаются в двоичном формате.

```
unsigned char *PQunescapeBytea(const unsigned char *from, size_t
*to_length);
```

Параметр *from* указывает на строку, такую, какую могла бы возвратить функция PQgetvalue, применённая к столбцу типа bytea. PQunescapeBytea преобразует это строковое представление в его двоичное представление. Она возвращает указатель на буфер, выделенный функцией malloc(), или NULL в случае ошибки и помещает размер буфера по адресу *to_length*. Когда результат не будет нужен, необходимо освободить его память, вызвав PQfreemem.

27.14. Асинхронная обработка команд

Функция PQexec используется для отправки команд серверу в нормальных, синхронных приложениях. Однако она имеет ряд недостатков, которые могут иметь значение для некоторых пользователей:

– PQexec ожидает завершения выполнения команды. Однако приложение может быть занято чем-то другим (например, обрабатывать активность в пользовательском интерфейсе), в таком случае блокировка приложения в ожидании ответа будет нежелательной.

– Поскольку выполнение клиентского приложения приостанавливается, пока оно ожидает результата, то приложению трудно решить, что оно хотело бы попытаться отменить выполняющуюся команду. (Это можно сделать из обработчика сигнала, но никак иначе.)

– PQexec может возвратить только одну структуру PGresult. Если отправленная серверу командная строка содержит несколько SQL-команд, функция PQexec отбрасывает все результаты PGresult, кроме последнего.

– PQexec всегда собирает все результаты выполнения команды, буферизуя их в единственной структуре PGresult. В то время как для приложения это упрощает логику обработки ошибок, это может быть непрактично, когда результат содержит много строк.

Приложения, которым эти ограничения не подходят, могут вместо PQexec использовать функции, на которых она базируется, PQsendQuery и PQgetResult. Есть также функции PQsendQueryParams, PQsendPrepare, PQsendQueryPrepared, PQsendDescribePrepared и PQsendDescribePortal, которые в сочетании с PQgetResult действуют аналогично функциям PQexecParams, PQprepare, PQexecPrepared, PQdescribePrepared и PQdescribePortal, соответственно.

– PQsendQuery – Отправляет команду серверу, не ожидая получения результата. Если команда была отправлена успешно, то функция возвратит значение 1, в противном случае она возвратит 0 (тогда нужно воспользоваться функцией PQerrorMessage для получения дополнительной информации о сбое).

```
int PQsendQuery(PGconn *conn, const char *command);
```

После успешного вызова PQsendQuery необходимо вызвать PQgetResult один или несколько раз, чтобы получить результаты. Функцию PQsendQuery нельзя вызвать повторно (на том же самом соединении) до тех пор, пока PQgetResult не вернёт нулевой указатель, означающий, что выполнение команды завершено.

– PQsendQueryParams – Отправляет серверу команду и обособленные параметры, не ожидая получения результатов.

```
int PQsendQueryParams(PGconn *conn,
const char *command, int nParams,
const Oid *paramTypes,
const char * const *paramValues, const int *paramLengths,
const int *paramFormats, int resultFormat);
```

– PQsendPrepare – Посылает запрос на создание подготовленного оператора с данными параметрами и не дожидается завершения его выполнения.

```
int PQsendPrepare(PGconn *conn,
const char *stmtName, const char *query, int nParams,
const Oid *paramTypes);
```

Возвращает 1, если ей удалось отправить запрос, и 0 в противном случае. После успешного вызова следует вызвать функцию PQgetResult, чтобы определить, создал ли сервер подготовленный оператор. Эта функция обрабатывает свои параметры точно так же, как и функция PQprepare.

– PQsendQueryPrepared – Посылает запрос на выполнение подготовленного оператора с данными параметрами, не ожидая получения результата.

```
int PQsendQueryPrepared(PGconn *conn,
const char *stmtName, int nParams,
const char * const *paramValues, const int *paramLengths,
const int *paramFormats, int resultFormat);
```

– PQsendDescribePrepared – Отправляет запрос на получение информации об указанном подготовленном операторе и не дожидается завершения выполнения запроса.

```
int PQsendDescribePrepared(PGconn *conn, const char *stmtName);
```

Возвращает 1, если ей удалось отправить запрос, и 0 в противном случае. После её успешного вызова следует вызватЬ функцию PQgetResult для получения результата. – PQsendDescribePortal – Отправляет запрос на получение информации об указанном портале и не дожидается завершения выполнения запроса.

```
int PQsendDescribePortal(PGconn *conn, const char *portalName);
```

Возвращает 1, если ей удалось отправить запрос, и 0 в противном случае. После её успешного вызова следует вызвать функцию PQgetResult для получения результата.

– PQgetResult – Ожидает получения следующего результата после предшествующего вызова PQsendQuery, PqsendQueryParams, PqsendPrepare,PQsendQueryPrepared, PQsendDescribePrepared или PQsendDescribePortal и возвращает его. Когда команда завершена и результатов больше не будет, возвращается нулевой указатель.

```
PGresult *PQgetResult(PGconn *conn);
```

Функция PQgetResult должна вызываться повторно до тех пор, пока она не вернёт нулевой указатель, означающий, что команда завершена. (Если она вызвана, когда нет ни одной активной команды, тогда PQgetResult просто возвратит нулевой указатель сразу же.) Необходимо освобождать память, занимаемую каждым результирующим объектом, с помощью функции PQclear когда работа с этим объектом закончена. PQgetResult заблокируется, только если какая- либо команда активна, а необходимые ответные данные ещё не были прочитаны функцией PQconsumeInput .

Использование PQsendQuery и PQgetResult решает одну из проблем PQexec: если строка запроса содержит несколько SQL-команд, то результаты каждой из них можно получить индивидуально.

Сам по себе вызов PQgetResult всё же приведёт к блокировке клиента до тех пор, пока сервер не завершит выполнение следующей SQL-команды. Этого можно избежать с помощью надлежащего использования ещё двух функций:

– PQconsumeInput – Если сервер готов передать данные, принять их.

```
int PQconsumeInput(PGconn *conn);
```

PQconsumeInput обычно возвращает 1, показывая, что «ошибки нет», но возвращает 0, если имела место какая-либо проблема.

PQconsumeInput можно вызвать, даже если приложение ещё не готово иметь дело с результатом или уведомлением. Функция прочитает доступные данные и сохранит их в буфере, при этом обрабатывая условие готовности к чтению функции select(). Таким образом, приложение может использовать PQconsumeInput, чтобы немедленно обработать это состояние select(), а изучать результаты позже.

– PQisBusy – Возвращает 1, если команда занята работой, то есть функция PQgetResult в случае вызова будет заблокирована в ожидании ввода. Возвращаемое значение 0 показывает, что функция PQgetResult при её вызове гарантированно не будет заблокирована.

```
int PQisBusy(PGconn *conn);
```

Функция PQisBusy сама не будет пытаться прочитать данные с сервера, поэтому, чтобы выйти из занятого состояния, необходимо вызвать PQconsumeInput.

– PQsetnonblocking – Устанавливает неблокирующий статус подключения.

```
int PQsetnonblocking(PGconn *conn, int arg);
```

Устанавливает состояние подключения как неблокирующее, если *arg* равен 1, или блокирующее, если *arg* равен 0. Возвращает 0 в случае успешного завершения и -1 в случае ошибки.

В неблокирующем состоянии вызовы PQsendQuery, PQputline, PQputnbytes, PQputCopyData и PQendcopy не будут блокироваться, а вместо этого возвратят ошибку, если их нужно будет вызвать ещё раз.

Функция PQexec не соблюдает неблокирующий режим. Если она вызывается, она всё равно работает в блокирующем режиме.

– PQisnonblocking – Возвращает режим блокирования для подключения базы данных.

```
int PQisnonblocking(const PGconn *conn);
```

Возвращает 1, если подключение установлено в неблокирующем режиме, и 0, если режим блокирующий.

– PQflush – Пытается сбросить любые выходные данные, стоящие в очереди, на сервер. Возвращает 0 в случае успеха (или если очередь на отправку пуста), -1 в случае сбоя по какой-либо причине или 1, если она ещё не смогла отправить все данные, находящиеся в очереди (этот случай может иметь место, только если соединение неблокирующее).

```
int PQflush(PGconn *conn);
```

После отправки любой команды или данных через неблокирующее подключение следует вызвать функцию PQflush. Если она возвратит 1, необходимо подождать, пока сокет станет готовым к чтению или записи. Если он станет готовым к записи, снова вызвать PQflush. Если он станет готовым к чтению, вызвать PQconsumeInput , а затем вновь вызвать PQflush. Повторять до тех пор, пока PQflush не возвратит 0. (Необходимо выполнять проверку на состояние готовности к чтению и забирать входные данные с помощью PQconsumeInput , потому что сервер может заблокироваться, пытаясь отправить данные, например, сообщения NOTICE, и не будет читать данные до тех пор, пока они не будут прочитаны) Как только PQflush возвратит 0, необходимо подождать, пока сокет не станет готовым к чтению, а затем прочитать ответ.

27.15. Построчное извлечение результатов запроса

Библиотека libpq собирает весь результат выполнения SQL-команды и возвращает его приложению в виде единственной структуры PGresult. Это может оказаться неприемлемым для команд, которые возвращают большое число строк. В таких случаях приложение может воспользоваться функциями PQsendQuery и PQgetResult в *однострочном режиме*. В этом режиме результирующие строки передаются приложению по одной за один раз, по мере того, как они принимаются от сервера.

Для того чтобы войти в однострочный режим, необходимо вызвать PQsetSingleRowMode сразу же после успешного вызова функции PQsendQuery (или родственной функции). Выбор этого режима действителен только для запроса, исполняемого в данный момент. Затем повторно вызвать функцию PQgetResult до тех пор, пока она не возвратит null. Если запрос возвращает какое-то число строк, то они возвращаются в виде индивидуальных объектов PGresult, которые выглядят, как обычные выборки, за исключением того, что их код статуса будет PGRES_SINGLE_TUPLE вместо PGRES_TUPLES_OK. После последней строки (или сразу же, если запрос не возвращает ни одной строки) будет возвращён объект, не содержащий ни одной

строки и имеющий статус PGRES_TUPLES_OK; это сигнал о том, что строк больше не будет. Все эти объекты PGresult будут содержать те же самые описательные данные (имена столбцов, типы и т. д.), которые имел бы обычный объект PGresult. Память, занимаемую каждым объектом, нужно освобождать с помощью PQclear, как обычно.

– PQsetSingleRowMode – Выбирает однострочный режим для текущего выполняющегося запроса.

```
int PQsetSingleRowMode(PGconn *conn);
```

Эту функцию можно вызывать только непосредственно после функции PQsendQuery или одной из её родственных функций, до выполнения любой другой операции через это подключение, такой, как PQconsumeInput или PQgetResult. Если вызвать её в подходящий момент, функция активирует однострочный режим для текущего запроса и возвращает 1. В противном случае режим остаётся прежним, а функция возвращает 0. Режим в любом случае сбрасывается по завершении текущего запроса.

27.16. Отмена запросов в процессе выполнения

Клиентское приложение может запросить отмену команды, которая ещё обрабатывается сервером, используя функции:

– PQgetCancel – Создаёт структуру данных, содержащую информацию, необходимую для отмены команды, запущенной через конкретное подключение к базе данных.

```
PGcancel *PQgetCancel(PGconn *conn);
```

Функция PQgetCancel создаёт объект PGcancel, получив объект PGconn, описывающий подключение. Она возвратит NULL, если параметр *conn* равен NULL или представляет недействительное подключение. Объект PGcancel является непрозрачной структурой, которая не предназначена для того, чтобы приложение обращалось к ней напрямую; её можно только передавать функции PQcancel или PQfreeCancel.

– PQfreeCancel – Освобождает память, занимаемую структурой данных, созданной функцией PqgetCancel.

```
void PQfreeCancel(PGcancel *cancel);
```

– PQcancel – Требует, чтобы сервер прекратил обработку текущей команды.

```
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

Возвращаемое значение равно 1, если запрос на отмену был успешно отправлен, и 0 в противном случае. В случае неудачной отправки *errbuf* заполняется пояснительным сообщением об ошибке. *errbuf* должен быть массивом символов, имеющим размер *errbufsize* (рекомендуемый размер составляет 256 байт).

27.17. Интерфейс быстрого пути

PG360 предоставляет интерфейс для передачи серверу простых вызовов функций по быстрому пути.

Функция PQfn запрашивает выполнение серверной функции посредством интерфейса быстрого доступа:

```
PGresult *PQfn(PGconn *conn,int fnid,int *result_buf, int
*result_len, int result_is_int,const PQArgBlock *args, int nargs);
```

Аргумент *fnid* представляет собой OID функции, которая подлежит выполнению. *args* и *nargs* определяют параметры, которые должны быть переданы этой функции; они должны соответствовать списку аргументов объявленной функции. Когда поле *isint* структуры, передаваемой в качестве параметра, имеет значение true, тогда значение *u.integer* передаётся серверу в виде целого числа указанной длины (это должно быть 2 или 4 байта); при этом устанавливается нужный порядок байтов. Когда *isint* имеет значение false, тогда указанное число байт по адресу **u.ptr* отправляется без какой-либо обработки; данные должны быть представлены в формате, которого ожидает сервер для передачи в двоичном виде данных того типа, что и аргументы функции. (Объявление поля *u.ptr*, как имеющего тип int *, является историческим; было бы лучше рассматривать его как тип void *.) *result_buf* указывает на буфер, в который должно быть помещено возвращаемое значение функции. Вызывающий код должен выделить достаточное место для сохранения возвращаемого значения. (Это никак не проверяется!) Фактическая длина результирующего значения в байтах будет возвращена в переменной целого типа, на которую указывает *result_len*. Если ожидается получение двух– или четырёхбайтового целочисленного результата, то необходимо присвоить параметру *result_is_int* значение 1, в противном случае назначить ему 0. Когда параметр *result_is_int* равен 1, libpq переставляет байты в передаваемом значении, если это необходимо, так, чтобы оно было доставлено на клиентскую машину в виде правильного значения типа int; по адресу **result_buf* доставляется четырёхбайтовое целое для любого допустимого размера результата. Когда *result_is_int* равен 0, тогда строка байтов в двоичном формате, отправленная сервером, будет возвращена немодифицированной. (В этом случае лучше рассматривать *result_buf* как имеющий тип void *.)

PQfn всегда возвращает действительный указатель на объект PGresult со статусом PGRES_COMMAND_OK при успешном выполнении функции или PGRES_FATAL_ERROR, если произошла какая-то ошибка. Перед использованием результата нужно сначала проверить его статус. Вызывающая функция отвечает за освобождение памяти, занимаемой объектом PGresult, когда он больше не нужен, с помощью PQclear.

27.18. Асинхронное уведомление

PG360 предлагает асинхронное уведомление посредством команд LISTEN и NOTIFY. Клиентский сеанс работы регистрирует свою заинтересованность в конкретном канале уведомлений с помощью команды LISTEN (и может остановить прослушивание с помощью команды UNLISTEN). Все сеансы, прослушивающие конкретный канал, будут уведомляться в асинхронном режиме, когда в рамках любого сеанса команда NOTIFY выполняется с параметром, указывающим имя этого канала. Для передачи дополнительных данных прослушивающим сеансам может использоваться строка «payload».

Приложения, использующие libpq, отправляют серверу команды LISTEN, UNLISTEN и NOTIFY как обычные SQL-команды. Поступление сообщений от команды NOTIFY можно впоследствии отследить с помощью функции PQnotifies.

Функция PQnotifies возвращает следующее уведомление из списка необработанных уведомительных сообщений, полученных от сервера. Она возвращает нулевой указатель, если нет уведомлений, ожидающих обработки. Как только уведомление возвращено из функции PQnotifies, оно считается обработанным и будет удалено из списка уведомлений.

```
PGnotify *PQnotifies(PGconn *conn);
```

После обработки объекта PGnotify, возвращённого функцией PQnotifies, обязательно необходимо освободить память, занимаемую им, с помощью функции PQfreemem. Достаточно освободить указатель на PGnotify; поля relname и extra не представляют отдельных областей памяти.

27.19. Функции, связанные с командой COPY

Команда COPY имеет возможность читать и записывать данные через сетевое подключение, установленное libpq. Описанные в этом разделе функции позволяют приложениям воспользоваться этой возможностью для передачи или приёма копируемых данных.

Общая процедура: сначала приложение выдаёт SQL-команду COPY, вызывая PQexec или одну из подобных функций. В ответ оно должно получить (если не возникла ошибка) объект PGresult с кодом состояния PGRES_COPY_OUT или PGRES_COPY_IN (в зависимости от направления копирования). Затем приложение должно использовать функции, описанные в этом разделе, и принимать или передавать строки данных. По завершении передачи возвращается ещё один объект PGresult, сообщающий о состоянии завершения передачи. В случае успеха он содержит код состояния PGRES_COMMAND_OK, а если возникает какая-то проблема – PGRES_FATAL_ERROR. После этого можно продолжать выполнять SQL-команды через PQexec. (Пока операция COPY не завершена, выполнять другие SQL-команды через то же подключение нельзя.)

Если команда COPY была выполнена через PQexec в строке, содержащей дополнительные команды, приложение должно продолжать получать результаты через PQgetResult после завершения последовательности COPY. Только когда PQgetResult возвращает NULL, можно с уверенностью считать, что переданные PQexec команды выполнены полностью, и безопасно передавать другие команды.

Функции должны выполняться только после получения кода состояния PGRES_COPY_OUT или PGRES_COPY_IN от функции PQexec или PqgetResult.

Объект PGresult с таким кодом состояния содержит дополнительные данные о начавшейся операции COPY. Эти данные можно получить функциями, также применяющимися при обработке результатов запроса:

– PQnfields – Возвращает число копируемых столбцов (полей).

– PQbinaryTuples – Значение 0 указывает, что для всей операции копирования применяется текстовый формат (строки разделяются символами новой строки, столбцы разделяются символами–разделителями и т. д.). Значение 1 указывает, что для всей операции копирования применяется двоичный формат.

– PQfformat – Возвращает код формата (0 – текстовый, 1 – двоичный), связанный с каждым копируемым столбцом. Коды форматов столбцов всегда будут нулевыми, если общий формат копирования – текстовый, но с двоичным форматом поддерживаются и текстовые, и двоичные столбцы.

27.19.1. Функции для передачи данных COPY

Эти функции применяются для передачи данных при операции COPY FROM STDIN. Они не будут работать, если подключение находится не в состоянии COPY_IN.

– PQputCopyData – Отправляет данные на сервер, когда активно состояние COPY_IN.

```
int PQputCopyData(PGconn *conn, const char *buffer, int nbytes);
```

Передаёт серверу данные COPY из указанного буфера (*buffer*), длиной *nbytes* байт. Она возвращает 1, если данные были переданы, 0, если они не попали в очередь, так как буферы были заполнены (это возможно только в неблокирующем режиме), или -1, если произошла ошибка. (Если возвращено -1, подробности ошибки можно узнать, вызвав PQerrorMessage. Если получен 0, необходимо дождаться состояния готовности к записи и повторить попытку.)

Приложение может разделять поток данных COPY на буферизуемые блоки любого удобного размера. Границы буфера не имеют семантического значения при передаче. Содержимое потока данных должно соответствовать формату данных, ожидаемому командой COPY.

– PQputCopyEnd – Отправляет признак конца данных на сервер, когда активно состояние COPY_IN.

```
int PQputCopyEnd(PGconn *conn, const char *errormsg);
```

Завершает операцию COPY_IN с успешным результатом, если в *errormsg* передаётся NULL. Если *errormsg* не NULL, команда COPY будет завершена с ошибкой, а сообщением об ошибке будет строка, переданная в *errormsg*.

Эта функция возвращает 1, если сообщение завершения было передано; в неблокирующем режиме это означает только, что сообщение завершения успешно поставлено в очередь. (Чтобы удостовериться, что данные были успешно отправлены в неблокирующем режиме, следует дождаться готовности к записи и вызывать PQflush в цикле, пока она не вернёт ноль.) Нулевой результат означает, что функция не смогла поставить сообщение завершения в очередь по причине заполнения буферов; это возможно только в неблокирующем режиме. (В этом случае нужно дождаться готовности к записи и попытаться вызвать PQputCopyEnd снова.) Если действительно происходит ошибка, возвращается -1; получить её подробности можно, вызвав PQerrorMessage.

После успешного вызова PQputCopyEnd необходимо вызвать PQgetResult, чтобы узнать окончательный результат команды COPY. Ожидать появления этого результата можно обычным образом. Затем вернуться к обычным операциям.

27.19.2. Функции для приёма данных COPY

Эти функции применяются для получения данных при операции COPY TO STDOUT. Они не будут работать, если подключение находится не в состоянии COPY_OUT.

– PQgetCopyData – Принимает данные от сервера, когда активно состояние COPY_OUT.

```
int PQgetCopyData(PGconn *conn,char **buffer, int async);
```

Запрашивает следующую строку данных с сервера в процессе операции COPY. Данные всегда возвращаются строка за строкой; если поступила только часть строки, она не возвращается. Успешное получение строки данных подразумевает выделение блока памяти для этих данных. В параметре *buffer* ей передаётся указатель, отличный от NULL. По адресу **buffer* записывается указатель на выделенную память, либо NULL, когда буфер не возвращается. Если буфер

результата отличен от NULL, его следует освободить, когда он станет не нужен, вызвав PQfreemem.

Когда строка получена успешно, возвращается число байт данных в этой строке (это число всегда больше нуля). Возвращаемое строковое значение всегда завершается нулём. Нулевой результат означает, что операция COPY продолжает выполняться, но строка ещё не готова (это возможно, только когда параметр *async* равен true). Возвращённое значение -1 означает, что команда COPY завершена, а -2 показывает, что произошла ошибка (её причину можно узнать с помощью PQerrorMessage).

Когда параметр *async* отличен от нуля (признак установлен), функция PQgetCopyData не будет блокироваться, ожидая данных; она возвратит ноль, если выполнение COPY продолжается, но полная строка ещё не получена. (В этом случае нужно дождаться готовности к чтению и затем вызвать PQconsumeInput , прежде чем вызывать PQgetCopyData ещё раз.) Когда *async* равен нулю (признак не установлен), PQgetCopyData будет заблокирована до поступления данных или окончания операции.

Когда PQgetCopyData возвращает -1, необходимо вызвать PQgetResult, чтобы узнать окончательный результат команды COPY. Ожидать появления этого результата можно обычным образом. Затем вернуться к обычным операциям.

27.1.10. Функции управления

Эти функции управляют различными аспектами поведения libpq:

– PQclientEncoding – Возвращает кодировку клиента.

```
int PQclientEncoding(const PGconn *conn);
```

Возвращает идентификатор кодировки, а не символьную строку вида EUC_JP. В случае ошибки она возвращает -1. Преобразовать идентификатор кодировки в имя можно, воспользовавшись следующей функцией:

```
char *pg_encoding_to_char(int encoding_id);
```

– PQsetClientEncoding – Устанавливает кодировку клиента.

```
int PQsetClientEncoding(PGconn *conn, const char *encoding);
```

В *conn* передаётся соединение с сервером, а в *encoding* – имя требуемой кодировки. Если функция устанавливает кодировку успешно, она возвращает 0, или -1 в противном случае. Определить текущую кодировку для соединения можно, воспользовавшись функцией PQclientEncoding.

– PQsetErrorVerbosity – Определяет уровень детализации сообщений, возвращаемых функциями PQerrorMessage и PQresultErrorMessage.

```
PGVerbosity PQsetErrorVerbosity(PGconn *conn, PGVerbosity verbosity);
```

PQsetErrorVerbosity устанавливает режим детализации и возвращает предыдущее значение для соединения. В режиме *TERSE* возвращаемые сообщения содержат только уровень важности, основной текст и позицию; всё это обычно умещается в одной строке. В режиме *DEFAULT* выдаваемые сообщения дополнительно содержат поля подробного описания, подсказки или контекста (они могут занимать несколько строк). В режиме *VERBOSE* передаются все доступные поля сообщения. В режиме *SQLSTATE* выдаётся только уровень важности и код ошибки

SQLSTATE, если он имеется (если же его нет, выводится та же информация, что и в режиме *TERSE*).

Изменение уровня детализации не влияет на сообщения, уже сформированные в существующих объектах PGresult, а затрагивает только последующие сообщения.

– PQsetErrorContextVisibility – Определяет вариант обработки полей CONTEXT в сообщениях, возвращаемых функциями PQerrorMessage и PQresultErrorMessage.

```
PGContextVisibility PQsetErrorContextVisibility(PGconn *conn, PGContextVisibility show_context);
```

PQsetErrorContextVisibility устанавливает режим вывода контекста и возвращает предыдущее значение для соединения. Этот режим определяет, будет ли поле CONTEXT включаться в сообщения. В режиме *NEVER* поле CONTEXT не включается никогда, а в режиме *ALWAYS* включается всегда, при наличии. В режиме *ERRORS* (по умолчанию) поле CONTEXT включается только в сообщения об ошибках, но не в замечания и предупреждения. (Однако при уровне детализации *TERSE* или *SQLSTATE* поле CONTEXT опускается вне зависимости от режима вывода контекста.)

Смена этого режима не влияет на сообщения, уже сформированные в существующих объектах PGresult, а затрагивает только последующие сообщения.

– PQtrace – Включает трассировку клиент-серверного взаимодействия с выводом в поток отладочных сообщений.

```
void PQtrace(PGconn *conn, FILE *stream);
```

– PQuntrace – Выключает трассировку, запущенную функцией Pqtrace.

```
void PQuntrace(PGconn *conn);
```

27.11.11. Функции разного назначения

– PQfreemem – Освобождает память, которую выделила libpq.

```
void PQfreemem(void *ptr);
```

Освобождает память, выделенную библиотекой libpq, а именно функциями PQescapeByteaConn, PQescapeBytea, PQunescapeBytea и PQnotifies.

– PQconninfoFree – Освобождает структуры данных, выделенные функциями PQconndefaults и PQconninfoParse.

```
void PQconninfoFree(PQconninfoOption *connOptions);
```

– PQencryptPasswordConn – Подготавливает зашифрованную форму пароля.

```
char *PQencryptPasswordConn(PGconn *conn, const char *passwd, const char *user, const char *algorithm);
```

В аргументах *passwd* и *user* задаётся пароль в открытом виде и SQL-имя пользователя, для которого он задаётся. В аргументе *algorithm* задаётся алгоритм для шифрования пароля.

Эта функция возвращает строку, выделенную функцией malloc.

– PQencryptPassword – Подготавливает зашифрованную md5 форму пароля.

```
char *PQencryptPassword(const char *passwd, const char *user);
```

– PQmakeEmptyPGresult – Конструирует пустой объект PGresult с указанным состоянием.

```
PGresult *PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

Это внутренняя функция libpq, выделяющая память и инициализирующая пустой объект PGresult. Эта функция возвращает NULL, если не может выделить память. Она сделана экспортируемой, так как некоторые приложения создают объекты результатов (в частности, объекты с состоянием ошибки) самостоятельно. Если в *conn* передаётся не null и *status* указывает на ошибку, в PGresult копируется текущее сообщение об ошибке для заданного соединения. Также, если в *conn* передаётся не null, в PGresult копируются все процедуры событий, зарегистрированные для этого соединения. (При этом вызовы PGEVT_RESULTCREATE не выполняются; см. описание PQfireResultCreateEvents.) В конце для этого объекта следует вызвать PQclear, как и для объекта PGresult, возвращённого самой библиотекой libpq.

– PQfireResultCreateEvents – Вызывает событие PGEVT_RESULTCREATE для каждой процедуры событий, зарегистрированной в объекте PGresult. Возвращает ненулевое значение в случае успеха или ноль в случае ошибки в одной из процедур.

```
int PQfireResultCreateEvents(PGconn *conn, PGresult *res);
```

Аргумент conn передаётся процедурам событий, но непосредственно не используется. Он может быть равен NULL, если он не нужен процедурам событий.

Процедуры событий, уже получившие событие PGEVT_RESULTCREATE или PGEVT_RESULTCOPY для этого объекта, больше не вызываются.

– PQcopyResult – Создаёт копию объекта PGresult. Эта копия никак не связана с исходным результатом и поэтому, когда она становится не нужна, необходимо вызвать PQclear. Если функция завершается ошибкой, она возвращает NULL.

```
PGresult *PQcopyResult(const PGresult *src, int flags);
```

– PQsetResultAttrs – Устанавливает атрибуты объекта PGresult.

```
int PQsetResultAttrs(PGresult *res, int numAttributes, PGresAttDesc
*attDescs);
```

Предоставленная структура *attDescs* копируется в результат. Если указатель *attDescs* равен NULL или *numAttributes* меньше одного, запрос игнорируется и функция выполняется без ошибки. Если *res* уже содержит атрибуты, функция завершается ошибкой. В случае ошибки функция возвращает ноль, а в обратном случае – ненулевое значение.

– PQsetvalue – Устанавливает значение поля кортежа в объекте PGresult.

```
int PQsetvalue(PGresult *res, int tup_num, int field_num, char
*value, int len);
```

– PQresultAlloc – Выделяет подчинённую область памяти для объекта PGresult.

```
void *PQresultAlloc(PGresult *res, size_t nBytes);
```

Любая память, выделенная этой функцией, будет освобождена при очистке объекта *res*. В случае ошибки эта функция возвращает NULL. Результат гарантированно выравнивается должным образом для любого типа данных, как и при malloc.

– PQresultMemorySize – Выдаёт объём памяти (в байтах), выделенной для объекта Pgresult.

```
size_t PQresultMemorySize(const PGresult *res);
```

Этот объём равен сумме размеров всех запросов malloc, связанных с данным объектом PGresult, то есть это общий объём памяти, который будет освобождён функцией PQclear.

– PQlibVersion – Возвращает версию используемой библиотеки libpq.

```
int PQlibVersion(void);
```

По результату этой функции можно во время выполнения определить, предоставляется ли определённая функциональность загруженной в данный момент версией libpq.

27.1.11.1. Типы событий

Перечисление PGEventId описывает типы событий, обрабатываемых системой событий. Имена всех их значений начинаются с PGEVT. Для каждого типа событий имеется соответствующая структура информации о событии, содержащая параметры, передаваемые обработчикам событий. Определены следующие типы событий:

– PGEVT_REGISTER – Событие регистрации происходит, когда вызывается PQregisterEventProc. Это подходящий момент для инициализации данных экземпляра (instanceData), которые могут понадобиться процедуре событий. Для каждого обработчика событий в рамках соединения будет выдаваться только одно событие регистрации. Если обработка события завершается ошибкой, регистрация прерывается.

При поступлении события PGEVT_REGISTER указатель *evtInfo* следует привести к PGEventRegister *. Эта структура содержит объект PGconn, который должен быть в состоянии CONNECTION_OK; это гарантируется, если PQregisterEventProc вызывается сразу после получения рабочего объекта PGconn. В случае выдачи кода ошибки всю очистку необходимо провести самостоятельно, так как событие PGEVT_CONNDESTROY не поступит.

– PGEVT_CONNRESET – Событие сброса соединения происходит при завершении PQreset или PQresetPoll. В обоих случаях это событие вызывается, только если сброс был успешным. Если обработка события завершается ошибкой, происходит сбой всей операции сброса соединения; объект PGconn переходит в состояние CONNECTION_BAD и PQresetPoll возвращает PGRES_POLLING_FAILED.

При поступлении события PGEVT_CONNRESET указатель *evtInfo* следует привести к PGEventConnReset *. Хотя переданный объект PGconn был только что сброшен, все данные события остаются неизменными. При поступлении этого события должны быть сброшены/ перезагружены/вновь запрошены все сопутствующие данные instanceData. Даже если обработчик события выдаст ошибку при обработке PGEVT_CONNRESET, событие PGEVT_CONNDESTROY всё равно поступит при закрытии соединения.

– PGEVT_CONNDESTROY – Событие уничтожения соединения вызывается в ответ на вызов PQfinish. Обработчик этого события отвечает за корректную очистку своих данных событий, так как libpq не может управлять его памятью. Невыполнение очистки должным образом приведёт к утечкам памяти.

При поступлении события PGEVT_CONNDESTROY указатель *evtInfo* следует привести к PGEventConnDestroy *. Это событие происходит перед тем, как PQfinish производит всю остальную очистку. Значение, возвращаемое обработчиком событий, игнорируется, так как из PQfinish никак нельзя сообщить об ошибке.

– PGEVT_RESULTCREATE – Событие создания объекта результата происходит при завершении любой функции, выполняющей запрос и получающей результат, включая PQgetResult. Это событие происходит только после того, как результат был успешно получен.

При поступлении события PGEVT_RESULTCREATE указатель *evtInfo* следует привести к PGEventResultCreate *. В *conn* передаётся соединение, для которого сформирован результат. Это

подходящее место для инициализации любых данных instanceData, которые нужно связать с результатом. В случае сбоя обработчика объект результата очищается и ошибка распространяется дальше. Обработчик события не должен пытаться выполнять PQclear для объекта результата самостоятельно. Возвращая ошибку, необходимо выполнить очистку данных, так как событие PGEVT_RESULTDESTROY для этого объекта не поступит.

– PGEVT_RESULTCOPY – Событие копирования объекта результата происходит при выполнении функции PQcopyResult. Это событие происходит только после завершения копирования. Только те обработчики событий, которые успешно обработали событие PGEVT_RESULTCREATE или PGEVT_RESULTCOPY для исходного объекта, получат событие PGEVT_RESULTCOPY.

При поступлении события PGEVT_RESULTCOPY указатель *evtInfo* следует привести к PGEventResultCopy *. Поле *src* указывает на объект результата, который копируется, а *dest* – на целевой объект. Это событие может применяться для реализации внутреннего копирования instanceData, так как сама функция PQcopyResult не может это сделать. В случае сбоя обработчика вся операция копирования прерывается и объект результата в *dest* очищается. Возвращая ошибку, необходимо выполнить очистку данных целевого объекта, так как событие PGEVT_RESULTDESTROY для него не поступит.

– PGEVT_RESULTDESTROY – Событие уничтожения объекта результата происходит при выполнении PQclear. Обработчик этого события отвечает за корректную очистку своих данных событий, так как libpq не может управлять его памятью. Невыполнение очистки должным образом приведёт к утечкам памяти.

При поступлении события PGEVT_RESULTDESTROY указатель *evtInfo* следует привести к PGEventResultDestroy *. Это событие происходит перед тем, как PQclear производит всю остальную очистку. Значение, возвращаемое обработчиком событий, игнорируется, так как из PQclear никак нельзя сообщить об ошибке. Кроме того, ошибка в обработчике событий не должна прерывать процесс очистки ставшей ненужной памяти.

27.11.12. Процедура обработки событий

– PGEventProc – это определение типа для указателя на обработчик событий, то есть функцию обратного вызова, получающую события от libpq. Обработчик событий должен иметь такую сигнатуру:

```
int eventproc(PGEventId evtId, void *evtInfo, void *passThrough)
```

Параметр *evtId* говорит, какое событие PGEVT произошло. Указатель *evtInfo* должен приводиться к типу определённой структуры для получения дополнительной информации о событии. В параметре *passThrough* передаётся сквозной указатель, поступивший в PQregisterEventProc при регистрации обработчика события. Эта функция должна вернуть ненулевое значение в случае успеха или ноль в противном случае.

Обработчик определённого события может быть зарегистрирован в любом PGconn только раз. Это связано с тем, что адрес обработчика используется как ключ для выбора связанных данных экземпляра.

27.11.13. Функции поддержки событий

– PQregisterEventProc – Регистрирует обработчик событий в libpq.

```
int PQregisterEventProc(PGconn *conn, PGEventProc proc,
const char *name, void *passThrough);
```

Обработчик событий должен быть зарегистрирован один раз для каждого соединения PGconn, события которого представляют интерес. Число обработчиков событий, которые можно зарегистрировать для соединения ограничивается только объёмом памяти. Эта функция возвращает ненулевое значение в случае успеха или ноль в противном случае.

Процедура, переданная в аргументе *proc*, будет вызываться, когда произойдёт событие libpq. Её адрес в памяти также применяется для поиска данных instanceData. Аргумент *name* используется при упоминании обработчика событий в сообщениях об ошибках. Это значение не может быть равно NULL или указывать на строку нулевой длины. Эта строка имени копируется в PGconn, так что переданная строка может быть временной. Сквозной указатель (*passThrough*) будет передаваться обработчику *proc* при каждом вызове события. Этот аргумент может равняться NULL.

– PQsetInstanceData – Устанавливает для подключения *conn* указатель instanceData для обработчика *proc* равным *data*. Эта функция возвращает ненулевое значение в случае успеха или ноль в противном случае. (Ошибка возможна, только если обработчик *proc* не был корректно зарегистрирован для соединения *conn*.)

```
int PQsetInstanceData(PGconn *conn, PGEventProc proc,
void *data);
```

– PQinstanceData – Возвращает для соединения *conn* указатель на instanceData, связанный с обработчиком *proc*, либо NULL, если такого обработчика нет.

```
void *PQinstanceData(const PGconn *conn, PGEventProc proc);
```

– PQresultSetInstanceData – Устанавливает для объекта результата (*res*) указатель instanceData для обработчика *proc* равным *data*. Эта функция возвращает ненулевое значение в случае успеха или ноль в противном случае. (Ошибка возможна, только если обработчик *proc* не был корректно зарегистрирован для объекта результата.)

```
int PQresultSetInstanceData(PGresult *res, PGEventProc proc,
void *data);
```

Память, представленная параметром *data*, не будет учитываться в PQresultMemorySize, если только она не была выделена функцией PQresultAlloc. (Этой функцией рекомендуется пользоваться, так как это избавляет от необходимости явно освобождать память после уничтожения результата.)

– PQresultInstanceData – Возвращает для объекта результата (*res*) указатель на instanceData, связанный с обработчиком *proc*, либо NULL, если такого обработчика нет.

```
void *PQresultInstanceData(const PGresult *res,
PGEventProc proc);
```

27.11.12. Переменные окружения

Воспользовавшись следующими переменными окружения, можно задать значения параметров соединения по умолчанию, которые будут использоваться функциями PQconnectdb, PQsetdbLogin и PQsetdb, если никакое значение не будет задано вызывающим кодом:

– PGHOST действует так же, как параметр соединения host.

– PGHOSTADDR действует так же, как параметр соединения hostaddr. Эту переменную можно задать вместо или вместе с PGHOST для предотвращения поиска адреса в DNS.

– PGPORT действует так же, как параметр соединения port.

– PGDATABASE действует так же, как параметр соединения dbname.

– PGUSER действует так же, как параметр соединения user.

– PGPASSWORD действует так же, как параметр соединения password. Использовать эту переменную окружения не рекомендуется по соображениям безопасности, так как в некоторых ОС непривилегированные пользователи могут видеть переменные окружения процессов в выводе ps; вместо этого лучше использовать файл паролей.

– PGPASSFILE действует так же, как параметр соединения passfile.

– PGCHANNELBINDING действует так же, как параметр соединения channel_binding.

– PGSERVICE действует так же, как параметр соединения service.

– PGSERVICEFILE задаёт имя личного файла пользователя с параметрами подключения к службам. По умолчанию применяется имя файла ~/.pg_service.conf.

– PGOPTIONS действует так же, как параметр соединения options.

– PGAPPNAME действует так же, как параметр соединения application_name.

– PGSSLMODE действует так же, как параметр соединения sslmode.

– PGREQUIRESSL действует так же, как параметр соединения requiressl. Эта переменная окружения утратила актуальность с появлением переменной PGSSLMODE; если установить обе переменные, значение данной не возымеет эффекта.

– PGSSLCOMPRESSION действует так же, как параметр соединения sslcompression.

– PGSSLCERT действует так же, как параметр соединения sslcert.

– PGSSLKEY действует так же, как параметр соединения sslkey.

– PGSSLROOTCERT действует так же, как параметр соединения sslrootcert.

– PGSSLCRL действует так же, как параметр соединения sslcrl.

– PGREQUIREPEER действует так же, как параметр соединения requirepeer.

– PGSSLMINPROTOCOLVERSION действует так же, как параметр соединения ssl_min_protocol_version.

– PGSSLMAXPROTOCOLVERSION действует так же, как параметр соединения ssl_max_protocol_version.

– PGKRBSRVNAME действует так же, как параметр соединения krbsrvname.

– PGCONNECT_TIMEOUT действует так же, как параметр соединения connect_timeout.

– PGCLIENTENCODING действует так же, как параметр соединения client_encoding.

– PGTARGETSESSIONATTRS действует так же, как параметр соединения target_session_attrs.

Следующие переменные окружения позволяют задать поведение по умолчанию для каждого отдельного сеанса:

– PGDATESTYLE устанавливает стиль представления даты/времени по умолчанию.

– PGTZ устанавливает часовой пояс по умолчанию.

– PGGEQO устанавливает режим по умолчанию для генетического оптимизатора запросов.

Следующие переменные среды определяют внутреннее поведение libpq; они переопределяют встроенные значения:

– PGSYSCONFDIR задаёт каталог, в котором содержится файл pg_service.conf, а в будущем он может содержать и другие общесистемные файлы конфигурации.

– PGLOCALEDIR задаёт каталог, содержащий файлы locale, предназначенные для перевода сообщений.

27.1.13. Файл паролей

Файл .pgpass в домашнем каталоге пользователя может содержать пароли, которые будут использоваться, если для подключения требуется пароль (и пароль не задаётся другим способом). Имя используемого файла паролей также можно задать в параметре подключения passfile или в переменной окружения PGPASSFILE.

Этот файл должен содержать строки следующего формата:

сервер:*порт*:*база_данных*:*имя_пользователя*:*пароль*

Первые четыре поля могут содержать строковые значения, либо знак *, соответствующий всему. Применяться будет пароль, указанный в первой из строк, значения полей в которой соответствуют текущему соединению. Поле с именем узла сопоставляется с параметром подключения host (если он указан) или с параметром hostaddr (если указан он); в случае отсутствия обоих параметров подразумевается имя localhost. Имя узла localhost также подразумевается, когда соединение устанавливается через Unix-сокет и параметр host соответствует установленному в libpq каталогу сокетов по умолчанию. На ведомом сервере имя базы данных replication соответствует подключениям к ведущему серверу, которые применяются для потоковой репликации. Поле *база_данных* имеет ограниченную ценность, так как пользователи используют один пароль для всех баз данных в кластере.

В системах Unix разрешения для файла паролей должны запрещать любой доступ к нему всем и группе; этого можно добиться командой chmod 0600 ~/.pgpass. Если разрешения будут менее строгими, этот файл будет игнорироваться.

27.1.14. Инициализация библиотеки SSL

Если приложение инициализирует библиотеку libssl и/или libcrypto, и libpq собрана с поддержкой SSL, необходимо вызвать PQinitOpenSSL, чтобы сообщить libpq, что библиотека libssl и/или libcrypto уже инициализированы приложением, чтобы libpq не пыталась ещё раз инициализировать их.

– PQinitOpenSSL – Позволяет приложениям выбрать, какие библиотеки безопасности нужно инициализировать.

```
void PQinitOpenSSL(int do_ssl, int do_crypto);
```

Когда параметр *do_ssl* отличен от нуля, libpq будет инициализировать библиотеку OpenSSL перед первым подключением к базе данных. Когда параметр *do_crypto* не равен нулю, будет инициализироваться библиотека libcrypto. По умолчанию (если функция PQinitOpenSSL не вызывается) инициализируются обе библиотеки.

Если приложение использует и инициализирует библиотеку OpenSSL или её нижележащую библиотеку libcrypto, необходимо вызвать эту функцию, передав нули в соответствующих параметрах, перед первым подключением к базе данных.

– PQinitSSL – Позволяет приложениям выбрать, какие библиотеки безопасности нужно инициализировать.

```
void PQinitSSL(int do_ssl);
```

Эта функция равнозначна вызову PQinitOpenSSL(do_ssl, do_ssl). Приложениям достаточно инициализировать или не инициализировать обе библиотеки OpenSSL и libcrypto одновременно.

27.1.15. Поведение в многопоточных программах

Библиотека libpq по умолчанию поддерживает повторные вызовы и многопоточность. Для соответствующего варианта сборки приложения может понадобиться передать компилятору специальные параметры командной строки. Чтобы собрать многопоточное приложение необходимо поискать в файле src/Makefile.global значения PTHREAD_CFLAGS и PTHREAD_LIBS.

Функция, позволяющая узнать, поддерживает ли libpq многопоточность:

– PQisthreadsafe – Возвращает состояние потокобезопасности в библиотеке libpq.

```
int PQisthreadsafe();
```

Возвращает 1, если библиотека libpq потокобезопасная, или 0 в противном случае.

27.1.16. Сборка программ с libpq

Чтобы собрать (то есть, скомпилировать и скомпоновать) программу, использующую libpq, необходимо выполнить следующие действия:

– Включить заголовочный файл libpq-fe.h: #include <libpq-fe.h>

– Сообщить компилятору каталог, в котором установлены заголовочные файлы PG360, передав ему параметр -I*каталог*. (В некоторых случаях компилятор сам может обращаться к нужному каталогу, так что этот параметр можно опустить.)

– При компоновке окончательной программы добавить параметр -lpq, чтобы была подключена библиотека libpq, а также параметр -L*каталог*, указывающий на каталог, в котором находится libpq.

2.7.2. Большие объекты

В PG360 имеется механизм для работы с *большими объектами*, предоставляющий доступ в потоковом режиме к пользовательским данным, сохранённым в специальной структуре больших объектов. Потоковый доступ удобен, когда нужно обрабатывать данные, объём которых слишком велик, чтобы оперировать ими как единым целым.

Далее описывается реализация, а также программный интерфейс и функции языка запросов для работы с данными больших объектов PG360.

27.2.1. Введение

Все большие объекты хранятся в одной системной таблице с именем pg_largeobject. Для каждого большого объекта также имеется запись в системной таблице pg_largeobject_metadata.

Большие объекты можно создавать, изменять и удалять, используя API чтения/записи, подобный стандартному API для работы с файлами.

PG360 также поддерживает систему хранения, названную «TOAST», которая автоматически переносит значения, не умещающиеся в одну страницу таблицы, в дополнительную область хранилища. Вследствие этого подсистема больших объектов отчасти оказывается устаревшей. Однако её преимуществом остаётся то, что она позволяет сохранять значения размером до 4 Тбайт, тогда как поля в TOAST ограничиваются 1 Гбайтом. Кроме того, чтение и изменение больших объектов можно выполнять эффективнее по сравнению с полями TOAST, которые при большинстве операций считываются и записываются как единое целое.

Механизм больших объектов разбивает большие объекты на «фрагменты» и сохраняет эти фрагменты в строках таблицы. При произвольном доступе на запись и чтение быстрый поиск нужного фрагмента обеспечивается индексом-B-деревом в этой таблице.

Фрагменты больших объектов не должны быть последовательными. Например, если приложение откроет новый большой объект, переместится к смещению 1000000 байт и запишет несколько байт, это не приведёт к выделению лишнего 1000000 байт в хранилище; записаны будут только фрагменты, покрывающие диапазон собственно записанных байт. Операция чтения прочитает нули для всех неразмещённых в хранилище байт, предшествующих последнему записанному фрагменту. Это соответствует принятому поведению «разреженных» файлов в файловых системах Unix.

Для больших объектов назначается владелец и набор прав доступа, которыми можно управлять командами GRANT и REVOKE. Для чтения большого объекта требуются права SELECT, а для записи или усечения его — права UPDATE. Удалять большой объект, задавать комментарий для него, либо сменять его владельца разрешается только его владельцу (или суперпользователю базы данных).

2722. Клиентские интерфейсы

В этом разделе описываются средства, которые предоставляет клиентская библиотека PG360 libpq для обращения к большим объектам. Интерфейс работы с большими объектами PG360 создан по подобию интерфейса файловых систем Unix, так что он включает аналоги функций open, read, write, lseek и т. д.

Все операции с большими объектами с применением этих функций *должны* иметь место в блоке транзакции SQL, так как дескрипторы больших объектов актуальны только во время транзакции.

Если при выполнении одной из этих функций происходит ошибка, эта функция возвращает значение, иначе невозможное, обычно 0 или -1. Сообщение, описывающее ошибку, сохраняется в объекте соединения; получить его можно с помощью PQerrorMessage.

Клиентские приложения, которые используют эти функции, должны включать заголовочный файл libpq/libpq-fs.h и компоноваться с библиотекой libpq.

27221. Создание большого объекта

Функция

```
Oid lo_creat(PGconn *conn, int mode);
```

создаёт новый большой объект. Возвращаемым значением будет OID, назначенный новому объекту, либо InvalidOid (ноль) в случае ошибки. Параметр *mode* не используется и игнорируется.

Функция

```
Oid lo_create(PGconn *conn, Oid lobjId);
```

также создаёт новый большой объект. В *lobjId* можно задать назначаемый ему OID; при этом произойдёт ошибка, если этот OID уже присвоен какому-либо большому объекту. Если в *lobjId* передаётся InvalidOid (ноль), lo_create присваивает большому объекту свободный OID (так же, как и lo_creat). Возвращаемым значением будет OID, назначенный новому большому объекту, либо InvalidOid (ноль) в случае ошибки.

27222. Импорт большого объекта

Чтобы импортировать в качестве большого объекта файл ОС, необходимо вызвать функцию:

```
Oid lo_import(PGconn *conn, const char *filename);
```

В *filename* задаётся имя файла в ОС, который будет импортирован как большой объект. Возвращаемым значением будет OID, назначенный новому большому объекту, либо InvalidOid (ноль) в случае ошибки.

Файл читает библиотека клиентского интерфейса, а не сервер; таким образом, он должен существовать в файловой системе на стороне клиента и быть доступным для чтения клиентскому приложению.

Функция

```
Oid lo_import_with_oid(PGconn *conn, const char *filename, Oid lobjId);
```

также импортирует новый большой объект. В *lobjId* можно задать назначаемый ему OID; при этом произойдёт ошибка, если этот OID уже присвоен какому-либо большому объекту. Если в *lobjId* передаётся InvalidOid (ноль), lo_import_with_oid присваивает большому объекту свободный OID (так же, как и lo_import).

Возвращаемым значением будет OID, назначенный новому большому объекту, либо InvalidOid (ноль) в случае ошибки.

27223. Экспорт большого объекта

Чтобы экспортировать большой объект в файл ОС, необходимо вызвать функцию:

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
```

В аргументе *lobjId* задаётся OID экспортируемого большого объекта, а в аргументе *filename* задаётся имя файла в ОС. Файл записывается библиотекой клиентского интерфейса, а не сервером.

Возвращает 1 при успешном выполнении, -1 при ошибке.

27224. Открытие существующего большого объекта

Чтобы открыть существующий большой объект для чтения или записи, необходимо вызвать функцию:

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
```

В аргументе *lobjId* задаётся OID открываемого большого объекта. Биты в аргументе *mode* определяют, открывается ли файл для чтения (INV_READ), для записи (INV_WRITE), либо для чтения/записи. (Эти константы определяются в заголовочном файле libpq/libpq-fs.h.)

Функция lo_open возвращает дескриптор большого объекта (неотрицательный) для последующего использования в функциях lo_read, lo_write, lo_lseek, lo_lseek64, lo_tell, lo_tell64, lo_truncate, lo_truncate64 и lo_close. Этот дескриптор актуален только до завершения текущей транзакции. В случае ошибки возвращается -1.

Функция lo_open завершится ошибкой, если пользователь не имеет права SELECT для данного большого объекта или если указан флаг INV_WRITE и отсутствует право UPDATE.

27225. Запись данных в большой объект

Функция

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

записывает *len* байт из буфера *buf* (который должен иметь размер *len*) в дескриптор большого объекта *fd*. В *fd* должно передаваться значение, возвращённое предыдущим вызовом lo_open.

Возвращает число фактически записанных байт (в текущей реализации это всегда *len*, если только не произошла ошибка). В случае ошибки возвращается значение -1.

Хотя параметр *len* объявлен как size_t, эта функция не принимает значение длины, превышающее INT_MAX. На практике лучше передавать данные фрагментами не больше нескольких мегабайт.

27226. Чтение данных из большого объекта

Функция

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

читает до *len* байт из дескриптора большого объекта *fd* в буфер *buf* (который должен иметь размер *len*). В *fd* должно передаваться значение, возвращённое предыдущим вызовом lo_open.

Возвращает эта функция число фактически прочитанных байт; это число должно быть меньше *len*, если при чтении был достигнут конец объекта. В случае ошибки возвращается -1.

Хотя параметр *len* объявлен как size_t, эта функция не принимает значение длины, превышающее INT_MAX. На практике лучше передавать данные фрагментами не больше нескольких мегабайт.

27227. Перемещение в большом объекте

Чтобы изменить текущее положение чтения или записи, связанное с дескриптором большого объекта, необходимо вызвать функцию:

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
```

Эта функция перемещает указатель текущего положения для дескриптора большого объекта *fd* в новое положение, заданное аргументом *offset*. Для аргумента *whence* задаются значения SEEK_SET (перемещение от начала объекта), SEEK_CUR (перемещение от текущего положения) и SEEK_END (перемещение от конца объекта).

Возвращает новое положение указателя, либо -1 в случае ошибки.

Для больших объектов, размер которых превышает 2 ГБ, необходимо использовать функцию:

```
pg_int64 lo_lseek64(PGconn *conn, int fd, pg_int64 offset, int whence);
```

Эта функция действует так же, как и lo_lseek, но может принять значение *offset*, превышающее 2 ГБ, и/или вернуть результат, превышающий 2 ГБ. Если новое положение указателя оказывается за границей в 2ГБ, функция lo_lseek выдаёт ошибку.

27228. Получение текущего положения в большом объекте

Чтобы получить текущее положение чтения или записи для дескриптора большого объекта, необходимо вызвать функцию:

```
int lo_tell(PGconn *conn, int fd);
```

Если возникает ошибка, возвращается -1.

Для больших объектов, размер которых может превышать 2 ГБ, необходимо использовать функцию:

```
pg_int64 lo_tell64(PGconn *conn, int fd);
```

Эта функция действует так же, как lo_tell, но может выдавать результат, превышающий 2 ГБ. lo_tell выдаёт ошибку, если текущее положение чтения/записи оказывается за границей в 2 ГБ.

27229. Усечение большого объекта

Чтобы усечь большой объект до требуемой длины, необходимо вызвать функцию:

```
int lo_truncate(PGconn *conn, int fd, size_t len);
```

которая усекает большой объект с дескриптором *fd* до длины *len*. В *fd* должно передаваться значение, возвращённое предыдущим вызовом lo_open. Если *len* превышает текущую длину большого объекта, большой объект расширяется до заданной длины нулевыми байтами ('\0').

В случае успеха lo_truncate возвращает ноль, а при ошибке возвращается -1.

Положение чтения/записи, связанное с дескриптором *fd*, при этом не меняется.

Для больших объектов, размер которых может превышать 2 ГБ, необходимо использовать функцию:

```
int lo_truncate64(PGconn *conn, int fd, pg_int64 len);
```

Эта функция действует так же, как lo_truncate, но может принимать значения *len*, превышающие 2 ГБ.

272210. Закрытие дескриптора большого объекта

Для закрытия дескриптора большого объекта необходимо вызвать функцию:

```
int lo_close(PGconn *conn, int fd);
```

где *fd* — дескриптор большого объекта, возвращённый функцией lo_open.

В случае успеха lo_close возвращает ноль. При ошибке возвращается -1.

Все дескрипторы больших объектов, остающиеся открытыми в конце транзакции, закрываются автоматически.

272211. Удаление большого объекта

Для удаления большого объекта из базы данных необходимо вызвать функцию:
`int lo_unlink(PGconn *conn, Oid lobjId);`
В аргументе *lobjId* задаётся OID большого объекта, который нужно удалить.
В случае успеха возвращается 1, а в случае ошибки -1.

2723. Серверные функции

Функции, предназначенные для работы с большими объектами на стороне сервера из SQL: – `lo_from_bytea ( loid oid, data bytea ) → oid` – Создаёт большой объёкт и сохраняет в нём переданные данные (*data*). Если *loid* равен нулю, система выбирает свободный OID, а иначе использует заданный идентификатор OID (и выдаёт ошибку, если этот OID уже назначен какому-то большому объекту). В случае успеха возвращается OID созданного большого объекта.

– `lo_put ( loid oid, offset bigint, data bytea ) → void` – Записывает данные (*data*) в большой объект по заданному смещению; в случае необходимости большой объект расширяется.

– `lo_get ( loid oid [, offset bigint, length integer] ) → bytea` – Извлекает содержимое большого объекта или его часть.

2.7.3. ECPG – Встраиваемый SQL в C

2.7.3.1 Обзор

Программа со встраиваемым SQL состоит из кода, написанного на языке программирования C, дополненного командами SQL в специально обозначенных секциях. Чтобы собрать программу, её исходный код (*.pgc) сначала нужно пропустить через препроцессор встраиваемого SQL, который превратит её в обычную программу на C (*.c), воспринимаемую компилятором C. Преобразованные приложения ECPG вызывают функции в библиотеке libpq через библиотеку встраиваемого SQL (ecpglib) и взаимодействуют с сервером PG360 по обычному клиент-серверному протоколу.

Встраиваемый SQL имеет ряд преимуществ по сравнению с другими методами вызова команд SQL из кода C: передача информации через переменные в программу на C; проверка на синтаксическую правильность во время сборки; встраиваемый в C язык SQL описан стандартом SQL и поддерживается многими другими СУБД SQL. Реализация в PG360 разработана так, чтобы максимально соответствовать этому стандарту, поэтому обычно достаточно портировать в PG360 программы с встраиваемым SQL, написанные для других СУБД.

Специальный код для операторов SQL имеет следующую форму: |"EXEC SQL ...;". Такие операторы синтаксически занимают место операторов C. В зависимости от конкретного оператора, они могут размещаться на глобальном уровне или внутри функции. Встраиваемые операторы SQL следуют правилам учёта регистра, принятым в обычном коде SQL, а не в C. Они также допускают вложенные комментарии в стиле C, разрешённые стандартом SQL. Однако остальная часть программы, написанная на C, в соответствии со стандартом C содержать вложенные комментарии не может.

2732. Управление подключениями к базе данных

В этом разделе описывается, как открывать, закрывать и переключать подключения к базам данных.

27321. Подключение к серверу баз данных

Подключение к базе данных выполняется следующим оператором:

EXEC SQL CONNECT TO *цель-подключения* [AS *имя-подключения*] [USER *имя-пользователя*];

Цель может задаваться следующими способами:

– *имя_бд*[@*имя_сервера*][:*порт*]

– tcp:postgresql://*имя_сервера*[:*порт*][/*имя_бд*][?*параметры*]

– unix:postgresql://*имя_сервера*[:*порт*][/*имя_бд*][?*параметры*]

– строковая константа SQL, содержащая одну из вышеприведённых записей

– ссылка на символьную переменную, содержащую одну из вышеприведённых записей

– DEFAULT

Если цель подключения задаётся буквально (то есть не через переменную) и значение не заключается в кавычки, регистр в этой строке не учитывается, как в обычном SQL. В этом случае при необходимости также можно заключить в двойные кавычки отдельные параметры. На практике, чтобы не провоцировать ошибки, лучше заключать строку в апострофы, либо передавать её в переменной. С целью подключения DEFAULT устанавливается подключение к базе данных по умолчанию с именем пользователя по умолчанию. Другое имя пользователя или имя подключения в этом случае указать нельзя.

Также разными способами можно указать имя пользователя:

– *имя_пользователя*

– *имя_пользователя*/*пароль*

– *имя_пользователя* IDENTIFIED BY *пароль*

– *имя_пользователя* USING *пароль*

Параметры *имя_пользователя* и *пароль* могут задаваться идентификатором или строковой константой SQL, либо ссылкой на символьную переменную.

Если указание цели подключения включает какие-либо *параметры*, они должны записываться в виде *имя=значение* и разделяться амперсандами (&). В качестве имён параметров принимаются те же, что поддерживает libpq. Перед элементами *имя* или *значение* пробелы игнорируются, но сохраняются внутри или после этих элементов.

Указание *имя-подключения* применяется, когда в одной программе нужно использовать несколько подключений. Его можно опустить, если программа работает только с одним подключением. Соединение, открытое последним, становится текущим и будет использоваться по умолчанию при выполнении SQL-операторов.

Если к базе данных, которая не приведена в соответствие шаблону безопасного использования схем, имеют доступ недоверенные пользователи, необходимо начинать сеанс с удаления схем, доступных всем для записи, из пути поиска (search_path).

27322. Выбор подключения

SQL-операторы в программах со встраиваемым SQL по умолчанию выполняются с текущим подключением, то есть с подключением, которое было открыто последним. Если приложению нужно управлять несколькими подключениями, это можно сделать двумя способами.

Первый вариант – явно выбирать подключение для каждого SQL-оператора, например:

```
EXEC SQL AT имя-подключения SELECT ...;
```

Этот вариант хорошо подходит для случаев, когда приложению нужно использовать несколько подключений в смешанном порядке.

Если приложение выполняется в нескольких потоках, они не могут использовать подключение одновременно. Поэтому необходимо либо явно управлять доступом (используя мьютексы), либо использовать отдельные подключения для каждого потока.

Второй вариант – выполнять оператор, переключающий текущее подключение. Этот оператор записывается следующим образом:

```
EXEC SQL SET CONNECTION имя-подключения;
```

Этот вариант особенно удобен, когда с одним подключением нужно выполнить несколько операторов.

27323. Закрытие подключения

Чтобы закрыть подключение, необходимо применить следующий оператор:

```
EXEC SQL DISCONNECT [подключение];
```

Подключение можно задать следующими способами:

– *имя-подключения;*

– DEFAULT;

– CURRENT;

– ALL.

Если имя подключения не задано, закрывается текущее подключение.

Приложение должно явно закрывать каждое подключение, которое оно открыло.

2733. Запуск команд SQL

В приложении со встраиваемым SQL можно запустить любую команду SQL. Ниже приведены несколько примеров, показывающих как это делать.

27331. Выполнение SQL-операторов

Создание таблицы:

```
EXEC SQL CREATE TABLE foo (number integer, ascii char(16)); EXEC SQL CREATE UNIQUE INDEX num1 ON foo(number);
EXEC SQL COMMIT;
```

Добавление строк:

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad'); EXEC SQL COMMIT;
```

Удаление строк:

```
EXEC SQL DELETE FROM foo WHERE number = 9999; EXEC SQL COMMIT;
```

Изменение:

```
EXEC SQL UPDATE foo
SET ascii = 'foobar' WHERE number = 9999;
EXEC SQL COMMIT;
```

Операторы SELECT, возвращающие одну строку результата, также могут выполняться непосредственно командой EXEC SQL.

Выборка одной строки:

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

Так же можно получить параметр конфигурации командой SHOW:

```
EXEC SQL SHOW search_path INTO :var;
```

Идентификаторы вида :*имя* воспринимаются как *переменные среды*, то есть они ссылаются на переменные программы C.

27332. Использование курсоров

Чтобы получить набор результатов, содержащий несколько строк, приложение должно объявить курсор и выбирать каждую строку через него. Использование курсора подразумевает следующие шаги: объявление курсора, открытие его, выборку строки через курсор, повторение предыдущего шага, и наконец, закрытие курсора.

Выборка с использованием курсоров:

```
EXEC SQL DECLARE foo_bar CURSOR FOR
SELECT number, ascii FROM foo ORDER BY ascii;
EXEC SQL OPEN foo_bar;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar; EXEC SQL COMMIT;
```

27333. Управление транзакциями

В режиме по умолчанию операторы фиксируются только когда выполняется EXEC SQL COMMIT. Интерфейс встраиваемого SQL также поддерживает автофиксацию транзакций (так работает libpq по умолчанию); она включается аргументом командной строки -t программы ecpg либо оператором EXEC SQL SET AUTOCOMMIT TO ON. В режиме автофиксации каждая команда фиксируется автоматически, если только она не помещена в явный блок транзакции. Этот режим можно выключить явным образом, выполнив EXEC SQL SET AUTOCOMMIT TO OFF.

Поддерживаются следующие команды управления транзакциями:

`EXEC SQL COMMIT` – Зафиксировать текущую транзакцию.

`EXEC SQL ROLLBACK` – Откатить текущую транзакцию.

`EXEC SQL PREPARE TRANSACTION` *`ид_транзакции`* – Подготовить текущую транзакцию для двухфазной фиксации.

`EXEC SQL COMMIT PREPARED ид_транзакции` – Зафиксировать транзакцию в подготовленном состоянии.

`EXEC SQL ROLLBACK PREPARED ид_транзакции` – Откатить транзакцию в подготовленном состоянии.

`EXEC SQL SET AUTOCOMMIT TO ON` – Включить режим автофиксации.

`EXEC SQL SET AUTOCOMMIT TO OFF` – Отключить режим автофиксации. По умолчанию он отключён.

27334. Подготовленные операторы

Когда значения, передаваемые SQL-оператору, неизвестны во время компиляции, или один и тот же оператор будет использоваться многократно, могут использоваться подготовленные операторы.

Оператор подготавливается командой PREPARE. Вместо значений, которые ещё неизвестны, вставляются местозаполнители «?»:

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid, datname FROM pg_database
WHERE oid = ?";
```

Если оператор возвращает одну строку, приложение может вызвать EXECUTE после PREPARE для выполнения этого оператора, указав фактические значения для местозаполнителей в предложении USING:

```
EXEC SQL EXECUTE stmt1 INTO :dboid, :dbname USING 1;
```

Если оператор возвращает несколько строк, приложение может использовать курсор, объявленный на базе подготовленного оператора. Чтобы привязать входные параметры, курсор нужно открыть с предложением USING:

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid,datname FROM pg_database
WHERE oid > ?"; EXEC SQL DECLARE foo_bar CURSOR FOR stmt1;
/* по достижении конца набора результатов прервать цикл while */
EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL OPEN foo_bar USING 100;
...
while (1)
{
EXEC SQL FETCH NEXT FROM foo_bar INTO :dboid, :dbname;
...
}
EXEC SQL CLOSE foo_bar;
```

Когда подготовленный оператор больше не нужен, его следует освободить:

```
EXEC SQL DEALLOCATE PREPARE имя;
```

2734. Использование переменных среды

При выполнении SQL-операторов в программе со встраиваемым SQL некоторые из этих операторов использовали только фиксированные значения и не давали возможности вставлять в операторы произвольные значения или обрабатывать значения, возвращённые запросом. В этом разделе описывается, как можно передавать данные между программой на C и встраиваемыми операторами SQL, используя простой механизм, так называемые *переменные среды*. В программе со встраиваемым SQL мы считаем SQL-операторы *внедрёнными* в код программы на C, *языке среды*. Таким образом, переменные программы на C называются *переменными среды*.

Ещё один способ передать значения данных между сервером PG360 и приложениями ECPG заключается в использовании дескрипторов SQL.

27341. Обзор

Для передачи данных между программой C и SQL-операторами во встраиваемом SQL необходимо записать имя переменной C в SQL-операторе, предварив его двоеточием. Например:

```
EXEC SQL INSERT INTO sometable VALUES (:v1, 'foo', :v2);
```

Этот оператор обращается к двум переменным C с именами v1 и v2 и также использует обычную строковую константу SQL, показывая тем самым, что можно свободно сочетать разные виды данных.

Этот метод включения переменных C в SQL-операторы работает везде, где SQL-оператор принимает выражение значения.

27342. Секции объявлений

Чтобы передать данные из программы в базу данных, например, в виде параметров запроса, либо получить данные из базы данных в программе, переменные C, которые должны содержать эти данные, нужно объявить в специально помеченных секциях, чтобы препроцессор встраиваемого SQL знал о них.

Секция начинается с:

```
EXEC SQL BEGIN DECLARE SECTION;
```

и заканчивается командой:

```
EXEC SQL END DECLARE SECTION;
```

Между этими строками должны располагаться обычные объявления переменных C, например:

```
int x = 4;
char foo[16], bar[16];
```

Как здесь показано, переменной можно присвоить начальное значение. Область видимости переменной определяется расположением секции, в которой она объявляется в программе. Можно объявить переменную следующим образом (при этом неявно создаётся секция объявлений):

```
EXEC SQL int i = 4;
```

Можно включать в программу столько секций объявлений, сколько необходимо.

Эти объявления выводятся в результирующий файл как объявления обычных переменных C, так что эти переменные не нужно объявлять снова. Переменные, которые не предназначены для использования в командах SQL, можно объявить как обычно вне этих специальных секций.

Определение структуры или объединения тоже должно размещаться в секции DECLARE. В противном случае препроцессор не сможет воспринять эти типы, так как не будет знать их определения.

27343. Получение результатов запроса

Для получения результатов запроса во встраиваемом SQL есть особые вариации обычных команд SELECT и FETCH. У этих команд есть специальное предложение INTO, определяющее, в какие переменные среды будут помещены получаемые значения. SELECT используется для запросов, возвращающих только одну строку, а FETCH применяется с курсором для запросов, возвращающих несколько строк.

27344. Сопоставление типов

Когда приложения ECPG передают данные между сервером PG360 и программой на C, например, получая результаты запроса с сервера или выполняя операторы SQL с входными параметрам, эти данные должны преобразовываться из типов PG360 в типы переменных языка среды (а именно типы языка C) и наоборот. Одно из главных качеств ECPG состоит в том, что в большинстве случаев он делает это автоматически.

В этом отношении можно выделить два вида типов данных. К первому относятся простые типы данных PG360, такие как integer и text, которые приложение может непосредственно читать и писать. С другими типами данных, такими как timestamp и numeric, можно работать только через специальные функции.

В Таблице 2 показано, как типы данных PG360 соответствуют типам данных C. Когда нужно передать или получить значение определённого типа данных PG360, необходимо объявить переменную C соответствующего типа C в секции объявлений.

Таблица 2. Соответствие между типами данных PG360 и типами переменных C

Тип данных PG360	**Тип переменной среды C**
smallint	short
integer	int
bigint	long long int
decimal	decimal[a]
numeric	numeric[a]
real	float
double precision	double
smallserial	short
serial	int
bigserial	long long int
oid	unsigned int
character(*n*), varchar(*n*), text	char[*n*+1], VARCHAR[*n*+1]
name	char[NAMEDATALEN]
timestamp	timestamp[a]
interval	interval[a]
date	date[a]
boolean	bool[b]
bytea	char *, bytea[*n*]

273441. Работа с символьными строками

Для обработки типов символьных строк SQL, таких как varchar и text, предлагаются два варианта объявления переменных среды.

Первый способ заключается в использовании char[], массива char, как чаще всего и представляются символьные данные в C.

```
EXEC SQL BEGIN DECLARE SECTION;
char str[50];
EXEC SQL END DECLARE SECTION;
```

В качестве другого подхода можно использовать специальный тип VARCHAR, представленный в ECPG. Определение массива типа VARCHAR преобразуется в структуру (struct) с собственным именем для каждой переменной. Объявление вида:

```
VARCHAR var[180];
```

преобразуется в:

```
struct varchar_var { int len; char arr[180]; } var;
```

Член структуры arr содержит строку, включающую завершающий нулевой байт. Таким образом, чтобы сохранить строку в переменной типа VARCHAR, эта переменная должна быть объявлена с длиной, учитывающей завершающий нулевой байт. Член структуры len содержит длину строки, сохранённой в arr, без завершающего нулевого байта. Когда на вход запросу подаётся переменная среды C, у которой strlen(arr) отличается от len, применяется наименьшее значение.

VARCHAR можно записать в верхнем или нижнем регистре, но не в смешанном.

Переменные char и VARCHAR также могут содержать значения других типов SQL в их строковом представлении.

273442. Обработка специальных типов данных

ECPG представляет некоторые особые типы, которые позволяют оперировать некоторыми специальными типами данных PG360. В частности, в нём реализована поддержка типов numeric, decimal, date, timestamp и interval. Для этих типов нельзя подобрать полезное соответствие с примитивными типами среды (например, int, long long int или char[]), так как они имеют сложную внутреннюю структуру. Приложения, работающие с этими типами, должны объявлять переменные особых типов и работать с ними, применяя функции из библиотеки pgtypes. Эта библиотека содержит базовые функции для оперирования этими типами, чтобы вам не требовалось, например, передавать запрос SQL-серверу, когда нужно просто добавить интервал к значению времени.

Эти особые типы данных описаны в следующих подразделах.

timestamp, date

Для работы с переменными timestamp в приложении ECPG применяется следующая схема.

Сначала в программу нужно включить заголовочный файл, чтобы получить определение типа:

```
timestamp:
#include <pgtypes_timestamp.h>
```

Затем объявить в секции объявлений переменную типа timestamp:

```
EXEC SQL BEGIN DECLARE SECTION;
timestamp ts;
EXEC SQL END DECLARE SECTION;
```

Прочитав значение в эту переменную, выполнить действия с ним, используя функции в библиотеке pgtypes.

Таким же образом можно работать и с типом DATE. В программу нужно включить pgtypes_date.h, объявить переменную типа date, и затем можно будет преобразовать значение DATE в текстовый вид, используя функцию PGTYPESdate_to_asc().

interval

Принцип работы с типом interval тот же, что и с типами timestamp и date, однако для значения типа interval нужно явно выделить память. Блок памяти для этой переменной должен размещаться в области кучи, а не в стеке.

numeric, decimal

Типы numeric и decimal обрабатываются так же, как и тип interval: необходимо определить указатель, выделить некоторое пространство памяти в куче и обращаться к переменной, используя функции в библиотеке pgtypes.

Для типа decimal никакие специальные функции не реализованы. Для дальнейшей обработки приложение должно преобразовать его в переменную numeric, применив функцию из библиотеки pgtypes.

bytea

Работа с типом bytea организуется так же, как и с VARCHAR. Определение каждой переменной– массива типа bytea преобразуется в именованную структуру.

Двоичные данные формата помещаются в поле arr. В отличие от типа VARCHAR, в данных bytea могут быть восприняты значения '\0'. Эти данные записываются/считываются и переводятся из шестнадцатеричного формата и обратно средствами ecpglib.

2735. Библиотека pgtypes

Библиотека pgtypes сопоставляет типы базы данных PG360 с их эквивалентами в C, которые можно использовать в программах на C. Она также предлагает функции для выполнения простых вычислений с этими типами в C, то есть без помощи сервера PG360.

27351. Символьные строки

Некоторые функции, в частности PGTYPESnumeric_to_asc, возвращают указатель на строку в выделенной для неё памяти. Их результаты должны освобождаться функцией PGTYPESchar_free, а не free.

27352. Тип numeric

Тип numeric позволяет производить вычисления с произвольной точностью. Ввиду того, что переменная имеет произвольную точность, она должна расширяться и сжиматься динамически. Поэтому такие переменные можно создавать только в области кучи, используя функции PGTYPESnumeric_new и PGTYPESnumeric_free. Тип decimal подобен numeric, но имеет ограниченную точность, и поэтому может размещаться и в области кучи, и в стеке.

Для работы с типом numeric можно использовать следующие функции:

– PGTYPESnumeric_new – Запрашивает указатель на новую переменную, размещённую в памяти.

```
numeric *PGTYPESnumeric_new(void);
```

– PGTYPESnumeric_free – Освобождает переменную типа numeric, высвобождая всю её память.

```
void PGTYPESnumeric_free(numeric *var);
```

– PGTYPESnumeric_from_asc – Разбирает числовой тип из строковой записи.

```
numeric *PGTYPESnumeric_from_asc(char *str, char **endptr);
```

Допускаются в частности следующие форматы: -2, .794, +3.44, 592.49E07 и -32.84e-4. Если значение удаётся разобрать успешно, возвращается действительный указатель, в противном случае указатель NULL. ECPG всегда разбирает строку до конца, так что эта функция не может вернуть адрес первого недопустимого символа в *endptr. Поэтому в endptr свободно можно передать NULL.

– PGTYPESnumeric_to_asc – Возвращает указатель на строку, выделенную функцией malloc и содержащую строковое представление значения num числового типа.

```
char *PGTYPESnumeric_to_asc(numeric *num, int dscale);
```

Числовое значение будет выводиться с заданным в dscale количеством цифр после запятой, округлённое при необходимости. Результат нужно освободить функцией PGTYPESchar_free().

– PGTYPESnumeric_add – Суммирует две числовые переменные и возвращает результат в третьей.

```
int PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result);
```

Эта функция суммирует переменные var1 и var2 в результирующую переменную result. Функция возвращает 0 в случае успеха и -1 при ошибке.

– PGTYPESnumeric_sub – Вычисляет разность двух числовых переменных и возвращает результат в третьей.

```
int PGTYPESnumeric_sub(numeric *var1, numeric *var2, numeric *result);
```

Эта функция вычитает переменную var2 из var1. Результат операции помещается в переменную result. Функция возвращает 0 в случае успеха и -1 при ошибке.

– PGTYPESnumeric_mul – Перемножает две числовые переменные и возвращает результат в третьей.

```
int PGTYPESnumeric_mul(numeric *var1, numeric *var2, numeric *result);
```

Эта функция перемножает переменные var1 и var2. Результат операции сохраняется в переменной result. Функция возвращает 0 в случае успеха и -1 при ошибке.

– PGTYPESnumeric_div – Вычисляет частное двух числовых переменных и возвращает результат в третьей.

```
int PGTYPESnumeric_div(numeric *var1, numeric *var2, numeric *result);
```

Эта функция делит переменную var1 на var2. Результат операции сохраняется в переменной result. Функция возвращает 0 в случае успеха и -1 при ошибке.

– PGTYPESnumeric_cmp – Сравнивает две числовые переменные.

```
int PGTYPESnumeric_cmp(numeric *var1, numeric *var2)
```

Эта функция производит сравнение двух числовых переменных. При ошибке возвращается INT_MAX. В случае успеха функция возвращает одно из трёх возможных значений:

1, если var1 больше var2

-1, если var1 меньше var2

0, если var1 и var2 равны

– PGTYPESnumeric_from_int – Преобразует переменную int в переменную numeric.

```
int PGTYPESnumeric_from_int(signed int int_val, numeric *var);
```

Эта функция принимает целочисленную переменную со знаком типа signed int и сохраняет её значение в переменной var типа numeric. Функция возвращает 0 в случае успеха и -1 при ошибке.

– PGTYPESnumeric_from_long – Преобразует переменную long int в переменную numeric.

```
int PGTYPESnumeric_from_long(signed long int long_val, numeric *var);
```

Эта функция принимает целочисленную переменную со знаком типа signed long int и сохраняет её значение в переменной var типа numeric. Функция возвращает 0 в случае успеха и -1 при ошибке.

– PGTYPESnumeric_copy – Копирует одну числовую переменную в другую.

```
int PGTYPESnumeric_copy(numeric *src, numeric *dst);
```

Эта функция копирует значение переменной, на которую указывает src, в переменную, на которую указывает dst. Она возвращает 0 в случае успеха и -1 при ошибке.

– PGTYPESnumeric_from_double – Преобразует переменную типа double в переменную numeric.

```
int PGTYPESnumeric_from_double(double d, numeric *dst);
```

Эта функция принимает переменную типа double и сохраняет преобразованное значение в переменной, на которую указывает dst. Она возвращает 0 в случае успеха и -1 при ошибке.

– PGTYPESnumeric_to_double – Преобразует переменную типа numeric в переменную double.

```
int PGTYPESnumeric_to_double(numeric *nv, double *dp)
```

Эта функция преобразует значение типа numeric переменной, на которую указывает nv, в переменную типа double, на которую указывает dp. Она возвращает 0 в случае успеха и -1 при ошибке, в том числе при переполнении. Если происходит переполнение, в глобальной переменной errno дополнительно устанавливается значение PGTYPES_NUM_OVERFLOW.

– PGTYPESnumeric_to_int – Преобразует переменную типа numeric в переменную int.

```
int PGTYPESnumeric_to_int(numeric *nv, int *ip);
```

Эта функция преобразует значение типа numeric переменной, на которую указывает nv, в целочисленную переменную, на которую указывает ip. Она возвращает 0 в случае успеха и -1 при ошибке, в том числе при переполнении. Если происходит переполнение, в глобальной переменной errno дополнительно устанавливается значение PGTYPES_NUM_OVERFLOW.

– PGTYPESnumeric_to_long – Преобразует переменную типа numeric в переменную long.

```
int PGTYPESnumeric_to_long(numeric *nv, long *lp);
```

Эта функция преобразует значение типа numeric переменной, на которую указывает nv, в целочисленную переменную типа long, на которую указывает lp. Она возвращает 0 в случае успеха

и -1 при ошибке, в том числе при переполнении. Если происходит переполнение, в глобальной переменной errno дополнительно устанавливается значение PGTYPES_NUM_OVERFLOW.

– PGTYPESnumeric_to_decimal – Преобразует переменную типа numeric в переменную decimal.

```
int PGTYPESnumeric_to_decimal(numeric *src, decimal *dst);
```

Эта функция преобразует значение типа numeric переменной, на которую указывает src, в переменную типа decimal, на которую указывает dst. Она возвращает 0 в случае успеха и -1 при ошибке, в том числе при переполнении. Если происходит переполнение, в глобальной переменной errno дополнительно устанавливается значение PGTYPES_NUM_OVERFLOW.

– PGTYPESnumeric_from_decimal – Преобразует переменную типа decimal в переменную numeric.

```
int PGTYPESnumeric_from_decimal(decimal *src, numeric *dst);
```

Эта функция преобразует значение типа decimal переменной, на которую указывает src, в переменную типа numeric, на которую указывает dst. Она возвращает 0 в случае успеха и -1 при ошибке. Так как тип decimal реализован как ограниченная версия типа numeric, при таком преобразовании переполнение невозможно.

27353. Тип date

Для работы с типом date можно использовать следующие функции:

– PGTYPESdate_from_timestamp – Извлекает часть даты из значения типа timestamp.

```
date PGTYPESdate_from_timestamp(timestamp dt);
```

Эта функция получает в единственном аргументе значение времени типа timestamp и возвращает извлечённую из него дату.

– PGTYPESdate_from_asc – Разбирает дату из её текстового представления.

```
date PGTYPESdate_from_asc(char *str, char **endptr);
```

Эта функция получает строку C char* str и указатель на строку C char* endptr. На данный момент ECPG всегда разбирает строку до конца, так что эта функция не может вернуть адрес первого недопустимого символа в *endptr. Поэтому в endptr свободно можно передать NULL.

– PGTYPESdate_to_asc – Возвращает текстовое представление переменной типа date.

```
char *PGTYPESdate_to_asc(date dDate);
```

Эта функция получает в качестве единственного параметра дату dDate и выводит её в виде 1999-01-18, то есть в формате YYYY-MM-DD. Результат необходимо освободить функцией PGTYPESchar_free().

– PGTYPESdate_julmdy – Извлекает значения дня, месяца и года из переменной типа date.

```
void PGTYPESdate_julmdy(date d, int *mdy);
```

Эта функция получает дату d и указатель на 3 целочисленных значения mdy. Имя переменной указывает на порядок значений: в mdy[0] записывается номер месяца, в mdy[1] – номер дня, а в mdy[2] – год.

– PGTYPESdate_mdyjul – Образует значение даты из массива 3 целых чисел, задающих день, месяц и год даты.

```
void PGTYPESdate_mdyjul(int *mdy, date *jdate);
```

Эта функция получает в первом аргументе массив из 3 целых чисел (mdy), а во втором указатель на переменную типа date, в которую будет помещён результат операции.

– PGTYPESdate_dayofweek – Возвращает число, представляющее день недели для заданной даты.

```
int PGTYPESdate_dayofweek(date d);
```

Эта функция принимает в единственном аргументе переменную d типа date и возвращает целое число, выражающее день недели для этой даты.

0 – Воскресенье

1 – Понедельник

2 – Вторник

3 – Среда

4 – Четверг

5 – Пятница

6 – Суббота

– PGTYPESdate_today – Выдаёт текущую дату.

```
void PGTYPESdate_today(date *d);
```

Эта функция получает указатель на переменную (d) типа date, в которую будет записана текущая дата.

– PGTYPESdate_fmt_asc – Преобразует переменную типа date в текстовое представление по маске формата.

```
int PGTYPESdate_fmt_asc(date dDate, char *fmtstring, char *outbuf);
```

Эта функция принимает дату для преобразования (dDate), маску формата (fmtstring) и строку, в которую будет помещено текстовое представление даты (outbuf).

В случае успеха возвращается 0, а в случае ошибки – отрицательное значение.

В строке формата можно использовать следующие коды полей:

dd – Номер дня в месяце.

mm – Номер месяца в году.

yy – Номер года в виде двух цифр.

yyyy – Номер года в виде четырёх цифр.

ddd – Название дня недели (сокращённое).

mmm – Название месяца (сокращённое).

– PGTYPESdate_defmt_asc – Преобразует строку C char* в значение типа date по маске формата.

```
int PGTYPESdate_defmt_asc(date *d, char *fmt, char *str);
```

Эта функция принимает указатель на переменную типа date (d), в которую будет помещён результат операции, маску формата для разбора даты (fmt) и строку C char*, содержащую текстовое представление даты (str). Текстовое представление будет соответствовать маске формата. Однако это соответствие не обязательно должно быть точным. Данная функция анализирует только порядок элементов и ищет в нём подстроки yy или yyyy, обозначающие позицию года, подстроку mm, обозначающую позицию месяца, и dd, обозначающую позицию дня.

27354. Тип timestamp

Для работы с типом timestamp можно использовать следующие функции:

– PGTYPEStimestamp_from_asc – Разбирает значение даты/времени из текстового представления в переменную типа timestamp.

```
timestamp PGTYPEStimestamp_from_asc(char *str, char **endptr);
```

Эта функция получает строку (str), которую нужно разобрать, и указатель на строку C char* (endptr). На данный момент ECPG всегда разбирает строку до конца, так что эта функция не может вернуть адрес первого недопустимого символа в *endptr. Поэтому в endptr можно передать NULL.

В случае успеха эта функция возвращает разобранное время, а в случае ошибки возвращается PGTYPESInvalidTimestamp и в errno устанавливается значение PGTYPES_TS_BAD_TIMESTAMP.

– PGTYPEStimestamp_to_asc – Преобразует значение даты в строку C char*.

```
char *PGTYPEStimestamp_to_asc(timestamp tstamp);
```

Эта функция принимает в качестве единственного аргумента tstamp значение типа timestamp и возвращает размещённую в памяти строку, содержащую текстовое представление даты/ времени. Результат необходимо освободить функцией PGTYPESchar_free().

– PGTYPEStimestamp_current – Получает текущее время.

```
void PGTYPEStimestamp_current(timestamp *ts);
```

Эта функция получает текущее время и сохраняет его в переменной типа timestamp, на которую указывает ts.

– PGTYPEStimestamp_fmt_asc – Преобразует переменную типа timestamp в строку C char* по маске формата.

```
int PGTYPEStimestamp_fmt_asc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

Эта функция получает в первом аргументе (ts) указатель на переменную типа timestamp, а в последующих указатель на буфер вывода (output), максимальную длину строки, которую может принять буфер (str_len), и маску формата, с которой будет выполняться преобразование (fmtstr).

В случае успеха возвращается 0, а в случае ошибки – отрицательное значение.

В маске формата можно использовать коды формата, перечисленные ниже. Эти же коды принимает функция strftime из библиотеки libc. Любые символы, не относящиеся к кодам формата, будут просто скопированы в буфер вывода.

%A – заменяется локализованным представлением полного названия дня недели.

%a – заменяется локализованным представлением сокращённого названия дня недели.

%B – заменяется локализованным представлением полного названия месяца.

%b – заменяется локализованным представлением сокращённого названия месяца.

%C – заменяется столетием (год / 100) в виде десятичного числа; одиночная цифра предваряется нулём.

%c – заменяется локализованным представлением даты и времени.

%D – равнозначно %m/%d/%y.

%d – заменяется днём месяца в виде десятичного числа (01–31).

%E* %O* – расширения локали POSIX. Последовательности %Ec %EC %Ex %EX %Ey %EY%Od %Oe %OH %OI %Om %OM %OS %Ou %OU %OV %Ow %OW %Oy должны выводить альтернативные представления.

Кроме того, альтернативные названия месяцев представляет код формата %OB

(используется отдельно, без упоминания дня).

%e – заменяется днём в виде десятичного числа (1–31); одиночная цифра предваряется пробелом.

%F – равнозначно %Y-%m-%d.

%G – заменяется годом в виде десятичного числа (со столетием). При этом годом считается тот, что содержит наибольшую часть недели (дни недели начинаются с понедельника).

%g – заменяется тем же годом, что и %G, но в виде десятичного числа без столетия (00–99).

%H – заменяется часами (в 24-часовом формате) в виде десятичного числа (00–23).

%h – равнозначно %b.

%I – заменяется часами (в 12-часовом формате) в виде десятичного числа (01–12).

%j – заменяется днём года в виде десятичного числа (001–366).

%k – заменяется часами (в 24-часовом формате) в виде десятичного числа (0–23); одиночная цифра предваряется пробелом.

%l – заменяется часами (в 12-часовом формате) в виде десятичного числа (1–12); одиночная цифра предваряется пробелом.

%M – заменяется минутами в виде десятичного числа (00–59).

%m – заменяется номером месяца в виде десятичного числа (01–12).

%n – заменяется символом новой строки.

%O* – равнозначно %E*.

%p – заменяется локализованным представлением «до полудня» или «после полудня» в зависимости от времени.

%R – равнозначно %H:%M.

%r – равнозначно %I:%M:%S %p.

%S – заменяется секундами в виде десятичного числа (00–60).

%s – заменяется числом секунд с начала эпохи, по мировому времени (UTC).

%T – равнозначно %H:%M:%S

%t – заменяется символом табуляции.

%U – заменяется номером недели в году (первым днём недели считается воскресенье) в виде десятичного числа (00–53).

%u – заменяется номером дня недели (первым днём недели считается понедельник) в виде десятичного числа (1–7).

%V – заменяется номером недели в году (первым днём недели считается понедельник) в виде десятичного числа (01–53). Если к недели, включающей 1 января, относятся 4 или больше дней нового года, она считается неделей с номером 1; в противном случае это последняя неделя предыдущего года, а неделей под номером 1 будет следующая.

%v – равнозначно %e-%b-%Y.

%W – заменяется номером недели в году (первым днём недели считается понедельник) в виде десятичного числа (00–53).

%w – заменяется номером дня недели (первым днём недели считается воскресенье) в виде десятичного числа (0–6).

%X – заменяется локализованным представлением времени.

%x – заменяется локализованным представлением даты.

%Y – заменяется годом со столетием в виде десятичного числа.

%y – заменяется годом без столетия в виде десятичного числа (00–99).

%Z – заменяется названием часового пояса.

%z – заменяется смещением часового пояса от UTC; ведущий знак плюс обозначает смещение к востоку от UTC, а знак минус – к западу, часы и минуты задаются парами цифр без разделителя между ними (эта форма установлена для даты в RFC 822).

%+ – заменяется локализованным представлением даты и времени.

%-* – расширение GNU libc. Отключает дополнение чисел по ширине при выводе.

$_* – расширение GNU libc. Явно включает дополнение пробелами.

%0* – расширение GNU libc. Явно включает дополнение нулями.

%% – заменяется символом %.

– PGTYPEStimestamp_sub – Вычитает одно значение времени из другого и сохраняет результат в переменной типа interval.

```
int PGTYPEStimestamp_sub(timestamp *ts1, timestamp *ts2, interval *iv);
```

Эта функция вычитает значение типа timestamp, на которое указывает ts2, из значения timestamp, на которое указывает ts1, и сохраняет результат в переменной типа interval, на которую указывает iv.

В случае успеха возвращается 0, а в случае ошибки – отрицательное значение.

– PGTYPEStimestamp_defmt_asc – Разбирает значение типа timestamp из текстового представления с заданной маской формата.

```
int PGTYPEStimestamp_defmt_asc(char *str, char *fmt, timestamp *d);
```

Эта функция получает текстовое представление даты/времени в переменной str, а также маску формата для разбора в переменной fmt. Результат будет сохранён в переменной, на которую указывает d.

Если вместо маски формата fmt передаётся NULL, эта функция переходит к стандартной маске форматирования, а именно: %Y-%m-%d %H:%M:%S.

– PGTYPEStimestamp_add_interval – Добавляет переменную типа interval к переменной типа timestamp.

```
int PGTYPEStimestamp_add_interval(timestamp *tin, interval *span, timestamp *tout);
```

Эта функция получает указатель на переменную tin типа timestamp и указатель на переменную span типа interval. Она добавляет временной интервал к значению даты/времени и сохраняет полученную дату/время в переменной типа timestamp, на которую указывает tout.

В случае успеха возвращается 0, а в случае ошибки – отрицательное значение.

– PGTYPEStimestamp_sub_interval – Вычитает переменную типа interval из переменной типа timestamp.

```
int PGTYPEStimestamp_sub_interval(timestamp *tin, interval *span, timestamp *tout);
```

Эта функция вычитает значение типа interval, на которое указывает span, из значения типа timestamp, на которое указывает tin, и сохраняет результат в переменной, на которую указывает tout.

В случае успеха возвращается 0, а в случае ошибки – отрицательное значение.

27355. Тип interval

Тип interval, реализованный в C, позволяет программам работать с данными типа interval в SQL.

Для работы с типом interval можно использовать следующие функции:

– PGTYPESinterval_new – Возвращает указатель на новую переменную interval, размещённую в памяти.

```
interval *PGTYPESinterval_new(void);
```

– PGTYPESinterval_free – Освобождает место, занимаемое ранее размещённой в памяти переменной типа interval.

```
void PGTYPESinterval_free(interval *intvl);
```

– PGTYPESinterval_from_asc – Разбирает значение типа interval из его текстового представления.

```
interval *PGTYPESinterval_from_asc(char *str, char **endptr);
```

Эта функция разбирает входную строку str и возвращает указатель на размещённую в памяти переменную типа interval. На данный момент ECPG всегда разбирает строку до конца, так что эта функция не может вернуть адрес первого недопустимого символа в *endptr. Поэтому в endptr свободно можно передать NULL.

– PGTYPESinterval_to_asc – Преобразует переменную типа interval в текстовое представление.

```
char *PGTYPESinterval_to_asc(interval *span);
```

Эта функция преобразует переменную типа interval, на которую указывает span, в строку C char*. Её вывод выглядит следующим образом: @ 1 day 12 hours 59 mins 10 secs. Результат необходимо освободить функцией PGTYPESchar_free().

– PGTYPESinterval_copy – Копирует переменную типа interval.

```
int PGTYPESinterval_copy(interval *intvlsrc, interval *intvldest);
```

Эта функция копирует переменную типа interval, на которую указывает intvlsrc, в переменную, на которую указывает intvldest. Для целевой переменной необходимо предварительно выделить память.

27356. Тип decimal

Тип decimal похож на тип numeric, однако его максимальная точность ограничена 30 значащими цифрами. В отличие от типа numeric, который можно создать только в области кучи, тип decimal можно создать и в стеке, и в области кучи (посредством функций PGTYPESdecimal_new и PGTYPESdecimal_free).

Для работы с типом decimal можно использовать следующие функции (содержащиеся не в библиотеке libcompat).

– PGTYPESdecimal_new – Запрашивает указатель на новую переменную decimal, размещённую в памяти.

```
decimal *PGTYPESdecimal_new(void);
```

– PGTYPESdecimal_free – Освобождает переменную типа decimal, высвобождая всю её память.

```
void PGTYPESdecimal_free(decimal *var);
```

Значения errno, которые устанавливает pgtypeslib:

– PGTYPES_NUM_BAD_NUMERIC – Аргумент должен содержать переменную типа numeric (либо указывать на переменную типа numeric), но представление этого типа в памяти оказалось некорректным.

– PGTYPES_NUM_OVERFLOW – Произошло переполнение. Так как тип numeric может принимать значения практически любой точности, при преобразовании этого типа в другие типы возможно переполнение.

– PGTYPES_NUM_UNDERFLOW – Произошло антипереполнение. Так как тип numeric может принимать значения практически любой точности, при преобразовании переменной этого типа в другие типы возможно антипереполнение.

– PGTYPES_NUM_DIVIDE_ZERO – Имела место попытка деления на ноль.

– PGTYPES_DATE_BAD_DATE – Функции PGTYPESdate_from_asc передана некорректная строка даты.

– PGTYPES_DATE_ERR_EARGS – Функции PGTYPESdate_defmt_asc переданы некорректные аргументы.

– PGTYPES_DATE_ERR_ENOSHORTDATE – В строке, переданной функции PGTYPESdate_defmt_asc, оказался неправильный компонент даты.

– PGTYPES_INTVL_BAD_INTERVAL – Функции PGTYPESinterval_from_asc передана некорректная строка, задающая интервал, либо функции PGTYPESinterval_to_asc передано некорректное значение интервала.

– PGTYPES_DATE_ERR_ENOTDMY – Обнаружено несоответствие при выводе компонентов день/месяц/год в функции PGTYPESdate_defmt_asc.

– PGTYPES_DATE_BAD_DAY – Функция PGTYPESdate_defmt_asc обнаружила некорректное значение дня месяца.

– PGTYPES_DATE_BAD_MONTH – Функция PGTYPESdate_defmt_asc обнаружила некорректное значение месяца.

– PGTYPES_TS_BAD_TIMESTAMP – Функции PGTYPEStimestamp_from_asc передана некорректная строка даты/времени, либо функции PGTYPEStimestamp_to_asc передано некорректное значение типа timestamp.

– PGTYPES_TS_ERR_EINFTIME – Значение типа timestamp, представляющее бесконечность, получено в недопустимом контексте.

Специальные константы pgtypeslib:

– PGTYPESInvalidTimestamp – Значение типа timestamp, представляющее недопустимое время. Это значение возвращает функция PGTYPEStimestamp_from_asc при ошибке разбора.

Вследствие особенности внутреннего представления типа timestamp, значение PGTYPESInvalidTimestamp в то же время представляет корректное время (1899-12-31 23:59:59). Поэтому для выявления ошибок необходимо, чтобы приложение не только сравнивало результат функции с PGTYPESInvalidTimestamp, но и проверяло условие errno != 0 после каждого вызова PGTYPEStimestamp_from_asc.

2736. Использование областей дескрипторов

Области дескрипторов SQL дают возможности для более сложной обработки результатов операторов SELECT, FETCH и DESCRIBE. Область дескриптора SQL объединяет в одной структуре данные одной строки и элементы метаданных. Эти метаданные используются при выполнении динамических SQL-операторов, когда характер результирующих столбцов может быть неизвестен заранее. PG360 предлагает два подхода к использованию областей дескрипторов: именованные области SQL-дескрипторов и области SQLDA в структурах C.

27361. Именованные области SQL-дескрипторов

Именованная область SQL-дескриптора состоит из заголовка, содержащего сведения обо всём дескрипторе, и одного или нескольких дескрипторов элементов, которые по сути описывают отдельные столбцы в строке результата.

Для использования области SQL-дескриптора, её нужно выделить:

```
EXEC SQL ALLOCATE DESCRIPTOR идентификатор;
```

Заданный идентификатор играет роль «имени переменной» области дескриптора. Когда дескриптор оказывается ненужным, его следует освободить:

```
EXEC SQL DEALLOCATE DESCRIPTOR идентификатор;
```

Чтобы воспользоваться областью дескриптора, её нужно указать в качестве целевого объекта в предложении INTO, вместо перечисления переменных среды:

```
EXEC SQL FETCH NEXT FROM mycursor INTO SQL DESCRIPTOR mydesc;
```

Если набор результатов пуст, в области дескриптора будут тем не менее содержаться метаданные из запроса, то есть имена полей.

Получить метаданные набора результатов для ещё не выполненных подготовленных запросов можно, воспользовавшись оператором DESCRIBE:

```
EXEC SQL BEGIN DECLARE SECTION;
char *sql_stmt = "SELECT * FROM table1"; EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
```

В операторах DESCRIBE и FETCH ключевые слова INTO и USING действуют примерно одинаково: они указывают вывести набор результатов и метаданные в область дескриптора.

Для получения данных из области дескриптора, нужно получить значение поля из заголовка и сохранить его в переменной среды C, для этого выполнить команду:

```
EXEC SQL GET DESCRIPTOR имя :переменная_среды = поле;
```

В настоящее время определено только одно поле заголовка: *COUNT*, которое говорит, сколько областей дескрипторов элементов существует (то есть, сколько столбцов содержится в

результате). Переменная среды C должна иметь целочисленный тип. Чтобы получить поле из области дескриптора элемента, нужно выполнить команду:

```
EXEC SQL GET DESCRIPTOR имя VALUE номер :переменная_среды = поле;
```

В качестве *num* можно задать обычное целое или переменную среды C, содержащую целое число. Допустимые поля:

CARDINALITY (integer) – число строк в наборе результатов;

DATA – элемент данных (тип данных поля зависит от запроса);

DATETIME_INTERVAL_CODE (целое) – Когда TYPE равно 9, DATETIME_INTERVAL_CODE содержит значение 1 для DATE, 2 для TIME, 3 для TIMESTAMP, 4 для TIME WITH TIME ZONE, либо 5 для TIMESTAMP WITH TIME ZONE.

DATETIME_INTERVAL_PRECISION (целое) – не реализовано;

INDICATOR (целое) – индикатор (отмечающий значение NULL или усечение значения);

KEY_MEMBER (целое) – не реализовано;

LENGTH (целое) – длина данных в символах;

NAME (строка) – имя столбца;

NULLABLE (целое) не реализовано;

OCTET_LENGTH (целое) – длина символьного представления данных в байтах

PRECISION (целое) – точность (для типа numeric);

RETURNED_LENGTH (целое) – длина данных в символах;

RETURNED_OCTET_LENGTH (целое) – длина символьного представления данных в байтах;

SCALE (целое) – масштаб (для типа numeric);

TYPE (целое) – числовой код типа данных столбца.

В операторах EXECUTE, DECLARE и OPEN ключевые слова INTO и USING действуют по-разному. Область дескриптора также можно сформировать вручную, чтобы передать входные параметры запросу или курсору, а команда USING SQL DESCRIPTOR *имя* даёт возможность передать входные аргументы параметризованному запросу. Оператор, формирующий именованную область SQL-дескриптора, выглядит следующим образом:

```
EXEC SQL SET DESCRIPTOR имя VALUE номер поле = :переменная_среды;
```

PG360 поддерживает выборку сразу нескольких записей в одном операторе FETCH и может сохранить их данные в переменной среды C, если эта переменная – массив. Например:

```
EXEC SQL BEGIN DECLARE SECTION;
int id[5];
EXEC SQL END DECLARE SECTION;
EXEC SQL FETCH 5 FROM mycursor INTO SQL DESCRIPTOR mydesc; EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :id = DATA;
```

27362. Области дескрипторов SQLDA

Область дескриптора SQLDA представляет собой структуру языка C, в которую можно получить набор результатов и метаданные запроса. Одна такая структура содержит одну запись из набора данных.

```
EXEC SQL include sqlda.h; sqlda_t *mysqlda;
EXEC SQL FETCH 3 FROM mycursor INTO DESCRIPTOR mysqlda;
```

Общая схема использования SQLDA выглядит следующим образом:

– Подготовить запрос и объявить курсор для него.

– Объявить SQLDA для строк результата.

– Объявить SQLDA для входных параметров и инициализировать их (выделить память, задать параметры).

– Открыть курсор с входной SQLDA.

– Выбрать строки из курсора и сохранить их в выходной SQLDA.

– Прочитать значения из выходной SQLDA в переменные среды (и преобразовать при необходимости).

– Закрыть курсор.

– Освободить область памяти, выделенную для входной SQLDA.

273621. Структура данных SQLDA

Для SQLDA используются три типа данных: sqlda_t, sqlvar_t и struct sqlname.

273622. Структура sqlda_t

Тип структуры sqlda_t представляет тип SQLDA. Эта структура описывает одну запись. Две или более структур sqlda_t могут объединяться в связанный список по указателям в поле desc_next, и таким образом образовывать упорядоченный набор строк. Поэтому, когда выбираются две или более строк, приложение может прочитать их, проследуя по указателям desc_next во всех узлах sqlda_t.

Тип sqlda_t определяется следующим образом:

```
struct sqlda_struct
{
char sqldaid[8];
long sqldabc;
short     sqln;
short     sqld;
struct sqlda_struct *desc_next; struct sqlvar_struct sqlvar[1];
};
typedef struct sqlda_struct sqlda_t;
```

Его поля имеют следующее назначение:

– sqldaid – Содержит строковую константу "SQLDA ".

– sqldabc – Содержит размер выделенного пространства в байтах.

– sqln – Содержит число входных параметров для параметризованного запроса, когда передаётся в операторы OPEN, DECLARE или EXECUTE с ключевым словом USING. В структуре, выводимой операторами SELECT, EXECUTE или FETCH, данное значение совпадает с sqld.

– sqld – Содержит число полей в наборе результатов.

– desc_next – Если запрос выдаёт несколько записей, возвращается несколько связанных структур SQLDA, а desc_next содержит указатель на следующую запись в списке.

– sqlvar – Это массив столбцов в наборе результатов.

273623. Структура sqlvar_t

Тип структуры sqlvar_t содержит значение столбца и метаданные, в частности, тип и длину. Эта структура определяется следующим образом:

```
struct sqlvar_struct
{
short sqltype;
short sqllen;
char *sqldata;
short *sqlind; struct sqlname sqlname;
};
typedef struct sqlvar_struct sqlvar_t;
```

Её поля имеют следующее назначение:

– sqltype – Содержит идентификатор типа данного поля. Возможные значения перечислены в enum ECPGttype в ecpgtype.h.

– sqllen – Содержит двоичную длину поля, например 4 байта для ECPGt_int.

– sqldata – Указывает на данные.

– sqlind – Указывает на индикатор NULL. 0 соответствует значению не NULL, -1 – NULL.

– sqlname – Имя поля.

273624. Структура struct sqlname

Структура struct sqlname содержит имя столбца. Она включена в sqlvar_t в качестве члена. Эта структура определена следующим образом:

```
#define NAMEDATALEN 64

struct sqlname
{
short length;
char data[NAMEDATALEN];
};
```

Её поля имеют следующее назначение:

– length – Содержит длину имени поля.

– data – Содержит имя поля.

27363. Получение набора результатов с применением SQLDA

Чтобы получить набор результатов запроса через SQLDA, нужно выполнить следующие действия:

– объявить структуру sqlda_t для получения набора результатов;

– выполнить команды FETCH/EXECUTE/DESCRIBE для обработки запроса с указанной SQLDA;

– определить число записей в наборе результатов, прочитав sqln, член структуры sqlda_t;

– получить значения каждого столбца из элементов sqlvar[0], sqlvar[1] и т. д., составляющих массив, включённый в структуру sqlda_t;

– перейти к следующей строке (структуре sqlda_t) по указателю desc_next, члену структуры sqlda_t;

– при необходимости повторить эти действия.

27364. Передача значений параметров через SQLDA

Чтобы передать параметры подготовленному запросу через SQLDA, нужно выполнить следующие действия:

– создать подготовленный запрос (подготовленный оператор);

– объявить структуру sqlda_t в качестве входной SQLDA;

– выделить область памяти (структуру sqlda_t) для входной SQLDA;

– установить (скопировать) входные значения в выделенной памяти;

– открыть курсор, указав входную SQLDA.

2737. Обработка ошибок

В этом разделе описывается, как можно обрабатывать исключительные условия и предупреждения в программе со встраиваемым SQL. Для этого предназначены два средства, которые могут дополнять друг друга.

Можно настроить функции-обработчики для обработки предупреждений и ошибок, воспользовавшись командой WHENEVER.

Подробную информацию об ошибке или предупреждении можно получить через переменную sqlca.

27371. Установка обработчиков

Один простой метод перехвата ошибок и предупреждений заключается в назначении определённого действия, которое будет выполняться при некотором условии. В общем виде:

```
EXEC SQL WHENEVER условие действие;
```

Здесь *условие* может быть следующим:

SQLERROR – Указанное действие вызывается, когда при выполнении SQL-оператора происходит ошибка.

SQLWARNING – Указанное действие вызывается, когда при выполнении SQL-оператора выдаётся предупреждение.

NOT FOUND – Указанное действие вызывается, когда SQL-оператор получает или обрабатывает ноль строк.

действие может быть следующим:

CONTINUE – Означает, что условие игнорируется. Это поведение по умолчанию.

GOTO *метка,* GO TO *метка* – Перейти к указанной метке (используя оператор goto языка C).

SQLPRINT – Вывести сообщение в устройство стандартного вывода.

STOP – Вызвать exit(1), что приведёт к завершению программы.

DO BREAK – Выполнить оператор break языка C. Этот вариант следует использовать только в циклах или операторах switch.

DO CONTINUE – Выполнить оператор continue языка C. Этот вариант следует использовать только в циклах. Данный оператор передаёт управление в начало цикла.

CALL *имя* (*аргументы*) DO *имя* (*аргументы*) – Вызвать указанные функции C с заданными аргументами. (Эти вызовы имеют смысловые отличия от CALL и DO в обычной грамматике PG360.)

sqlca

Для более гибкой обработки ошибок в интерфейсе встраиваемого SQL представлена глобальная переменная с именем sqlca (SQL Communication Area, Область сведений SQL), имеющая следующую структуру:

```
struct
{
char sqlcaid[8];
long sqlabc; long sqlcode; struct
{
int sqlerrml;
char sqlerrmc[SQLERRMC_LEN];
} sqlerrm;
char sqlerrp[8]; long sqlerrd[6]; char sqlwarn[8]; char
sqlstate[5];
} sqlca;
```

(В многопоточной программе каждый поток автоматически получает собственную копию sqlca. Это работает подобно стандартной в C глобальной переменной errno.)

Структура sqlca покрывает и предупреждения, и ошибки. Если в процессе выполнения оператора выдаётся несколько предупреждений или ошибок, sqlca будет содержать сведения только о последнем(ей) из них.

Если последний оператор SQL выполняется без ошибки, sqlca.sqlcode будет содержать 0, а sqlca.sqlstate – "00000". Если выдаётся предупреждение или ошибка, в sqlca.sqlcode будет содержаться отрицательное число, а sqlca.sqlstate будет отличаться от "00000". Положительное значение sqlca.sqlcode устанавливается при нейтральном событии, например, когда последний запрос возвращает ноль строк. Поля sqlcode и sqlstate представляют две различные схемы кодов ошибок; подробнее они описаны ниже.

Если последний SQL-оператор был успешным, в sqlca.sqlerrd[1] содержится OID обработанной строки (если это уместно), а в sqlca.sqlerrd[2] количество обработанных или возвращённых строк (если это уместно для команды).

В случае ошибки или предупреждения sqlca.sqlerrm.sqlerrmc будет содержать строку, описывающую ошибку. Поле sqlca.sqlerrm.sqlerrml содержит длину сообщения об ошибке, которое хранится в sqlca.sqlerrm.sqlerrmc (результат функции strlen(), который не очень интересен для программиста C). Некоторые сообщения могут не умещаться в массив sqlerrmc фиксированного размера; они будут обрезаться.

В случае предупреждения, в sqlca.sqlwarn[2] записывается символ W. (Во всех других случаях значение будет отличным от W.) Символ W в sqlca.sqlwarn[1] показывает, что значение было обрезано при сохранении в переменной среды C. W в sqlca.sqlwarn[0] устанавливается, если предупреждение отмечается в каком-либо другом элементе массива.

2738. Директивы препроцессора

Препроцессор ecpg поддерживает ряд директив, которые позволяют управлять разбором и обработкой исходных файлов.

27381. Включение файлов

Для включения внешнего файла в программу со встраиваемым SQL, используется конструкция:

```
EXEC SQL INCLUDE имя_файла;
EXEC SQL INCLUDE <имя_файла>;
EXEC SQL INCLUDE "имя_файла";
```

Встретив такую директиву, препроцессор встраиваемого SQL будет искать файл *имя_файла*.h, обрабатывать его и включать в выходной код C. В результате встраиваемые SQL-операторы во включённом таким образом файле будут обработаны корректно.

Препроцессор ecpg будет искать указанный файл в нескольких каталогах в следующем порядке:

– текущий каталог /usr/local/include;

– каталог включаемых файлов PG360, определённый во время сборки (например, /usr/local/pgsql/include);

– /usr/include.

Но когда используется форма EXEC SQL INCLUDE "*имя_файла*", просматривается только текущий каталог.

В каждом каталоге препроцессор будет сначала искать файл с заданным именем, а если не обнаружит его, попытается найти файл с добавленным расширением .h (если только заданное имя файла уже не содержит это расширение).

Команда EXEC SQL INCLUDE *не* равнозначна включению:

#include <*имя_файла*.h>

так как во втором случае включаемый файл не проходит через препроцессор SQL-команд. Естественно, директиву C #include можно по-прежнему применять для включения других заголовочных файлов.

27382. Директивы define и undef

Во встраиваемом SQL есть конструкция, подобная директиве #define, известной в C:

```
EXEC SQL DEFINE имя;
EXEC SQL DEFINE имя значение;
```

Используя её, можно определить имя:

```
EXEC SQL DEFINE HAVE_FEATURE;
```

И также можно определить константы:

```
EXEC SQL DEFINE MYNUMBER 12; EXEC SQL DEFINE MYSTRING 'abc';
```

Удалить предыдущее определение позволяет команда undef:

```
EXEC SQL UNDEF MYNUMBER;
```

В программе со встраиваемым SQL можно продолжать использовать версии #define и #undef языка C. Отличие состоит в том, когда вычисляются определяемые значения. Когда

применяется команда EXEC SQL DEFINE, вычислять определения и подставлять значения будет препроцессор ecpg. Например, если написать:

```
EXEC SQL DEFINE MYNUMBER 12;
...
EXEC SQL UPDATE Tbl SET col = MYNUMBER;
```

подстановку выполнит ecpg и компилятор C никогда не увидит имени или идентификатора MYNUMBER.

27383. Директивы ifdef, ifndef, elif, else и endif

Для условной компиляции блоков кода можно использовать следующие указания:

`EXEC SQL ifdef` *`имя;`* – Проверяет *имя* и обрабатывает последующие строки, если *имя* было определено командой EXEC SQL define *имя*.

`EXEC SQL ifndef` *`имя;`* – Проверяет *имя* и обрабатывает последующие строки, если *имя не* было определено командой: EXEC SQL define *имя*.

`EXEC SQL elif` *`имя;`* – Начинает необязательный альтернативный блок после указания EXEC SQL ifdef *имя* или EXEC SQL ifndef *имя*. Количество блоков elif может быть любым. Строки, следующие за elif, будут обрабатываться, если *имя* определено *и* при этом не был обработан ни один из блоков той же конструкции ifdef/ifndef...endif.

`EXEC SQL else;` – Начинает необязательный заключительный блок после указания EXEC SQL ifdef *имя* или EXEC SQL ifndef *имя*. Последующие строки будут обрабатываться, если не был обработан ни один из блоков той же конструкции ifdef/ifndef...endif.

`EXEC SQL endif;` – Завершает конструкцию ifdef/ifndef...endif. Последующие строки обрабатываются обычным образом.

Конструкции ifdef/ifndef...endif могут быть вложенными, на глубину до 127 уровней. Так будет скомпилирована только одна из трёх команд SET TIMEZONE:

```
EXEC SQL ifdef TZVAR;
EXEC SQL SET TIMEZONE TO TZVAR; EXEC SQL elif TZNAME;
EXEC SQL SET TIMEZONE TO TZNAME;
EXEC SQL else;
EXEC SQL SET TIMEZONE TO 'GMT';
EXEC SQL endif;
```

2739. Компиляция программ со встраиваемым SQL

Прежде чем компилировать код C, необходимо пропустить исходный файл через препроцессор встраиваемого SQL в C, который преобразует записанные вами операторы SQL в вызовы специальных функций. После компиляции полученный объектный код нужно скомпоновать со специальной библиотекой, содержащей необходимые функции. Эти функции получают информацию из аргументов, выполняют команды SQL через интерфейс libpq, и помещают результат в аргументы, заданные для вывода.

Программа препроцессора называется ecpg и входит в состав обычной инсталляции PG360. Программам со встраиваемым SQL, как правило, даются имена с расширением .pgc. Созданный код программы в файле prog1.pgc, можно обработать, просто выполнив:

```
ecpg prog1.pgc
```

При этом будет создан файл prog1.c. Если имена входных файлов не следуют этому соглашению, имя выходного файла можно задать явно в аргументе -o.

Обработанный препроцессором файл можно скомпилировать обычным образом, например:

```
cc -c prog1.c
```

В сгенерированные исходные файлы C включаются заголовочные файлы из инсталляции PG360, поэтому если PG360 установлен так, что соответствующий каталог не просматривается по умолчанию, необходимо добавить указание вида -I/usr/local/pgsql/ include в командную строку компиляции.

Чтобы скомпоновать программу со встраиваемым SQL, необходимо подключить библиотеку libecpg:

```
cc -o myprog prog1.o prog2.o ... -lecpg
```

Возможно, и для этого понадобится добавить в командную строку указание вида -L/usr/local/ pgsql/lib.

Чтобы узнать пути инсталляции, можно воспользоваться командой pg_configили pkg– config(в качестве имени пакета нужно указать libecpg).

Если процесс сборки большого проекта организается с применением make, может быть удобно включить в сборочные файлы следующее неявное правило:

```
ECPG = ecpg

%.c: %.pgc
$(ECPG) $<
```

Библиотека ecpg по умолчанию потокобезопасна. Однако для компиляции клиентского кода могут потребоваться параметры командной строки для настройки многопоточности.

27.3.10. Библиотечные функции

Библиотека libecpg в основном содержит «скрытые» функции, применяемые для реализации функциональности, выражаемой встраиваемыми командами SQL. Но есть также некоторые функции, которые можно вызывать напрямую.

– ECPGdebug(int *вкл,* FILE **поток*) с первым аргументом, отличным от нуля, включает вывод отладочных сообщений в заданный *поток*. Журнал сообщений, полученный таким образом, будет содержать все операторы SQL с заданными входными переменными и результаты, выданные сервером PG360.

– ECPGget_PGconn(const char **имя_подключения*) возвращает указатель на подключение к базе данных, имеющее заданное имя. Если аргумент *имя_подключения* равен NULL, возвращается указатель на текущее подключение. Если определить подключение не удаётся, возвращается NULL. Полученный указатель на подключение, если требуется, можно использовать при вызове любых других функций libpq.

– ECPGtransactionStatus(const char **имя_подключения*) возвращает состояние текущей транзакции для подключения, на которое указывает *имя_подключения*.

– ECPGstatus(int *номер_строки*, const char* *имя_подключения*) возвращает true при наличии подключения к базе данных и false в противном случае. В аргументе *имя_подключения* можно передать NULL, если применяется одно подключение.

27311. Приложения на C++

ECPG обеспечивает поддержку языка C++ в ограниченном объёме.

Препроцессор ecpg принимает входной файл, написанный на C (или языке, подобном C) со встраиваемыми командами SQL, преобразует встроенные команды SQL в конструкции языка C и в результате формирует файл .c. Объявления библиотечных функций, вызываемых в конструкциях C, которые генерирует ecpg, заворачиваются в блоки extern "C" { ... } при использовании C++, так что они должны прозрачно работать в C++.

Препроцессор ecpg понимает только C; он не воспринимает особый синтаксис и зарезервированные слова языка C++. Поэтому какой-то код SQL, встроенный в код приложения на C++, в котором используются сложные особенности C++, может корректно не обработаться препроцессором или не работать как ожидается.

Надёжный подход к применению внедрённого кода SQL в приложении на C++ заключается в том, чтобы скрыть вызовы ECPG в модуле C, который будет вызываться приложением на C++ для работы с базой данных и который будет скомпонован с остальным кодом C++.

27312. Разработка приложения на C++ с внешним модулем на C

Для использования ECPG в приложениях на C++ лучше связывать код C с кодом C++ на стадии компоновки, а не внедрять команды SQL непосредственно в код на C++. В данном разделе показывается, как отделить встраиваемые команды SQL от кода приложения на C++, на примере. В этом примере приложение реализуется на C++, а взаимодействие с сервером PG360 построено на C и ECPG.

Для сборки нужно создать три типа файлов: файл на C (*.pgc), заголовочный файл и файл на C++:

test_mod.pgc – Модуль подпрограмм будет выполнять SQL-команды, встроенные в C. Этот код нужно будет преобразовать в test_mod.c с помощью препроцессора.

```
     #include "test_mod.h" #include <stdio.h>
     void db_connect()
     {
     EXEC SQL CONNECT TO testdb1;
     EXEC SQL SELECT pg_catalog.set_config('search_path', '', false);
EXEC SQL COMMIT;
     }
     void db_test()
     {
     EXEC SQL BEGIN DECLARE SECTION;
     char dbname[1024];
     EXEC SQL END DECLARE SECTION;
     EXEC SQL SELECT current_database() INTO :dbname;
printf("current_database = %s\n", dbname);
     }
     void db_disconnect()
     {
     EXEC SQL DISCONNECT ALL;
     }
```

test_mod.h – Заголовочный файл с объявлениями функций в модуле на языке C (test_mod.pgc). Он включается в test_cpp.cpp. Объявления в этом файле должны заключаться в блок extern "C", так как он будет связываться с модулем C++.

```
#ifdef cplusplus extern "C" { #endif

void db_connect(); void db_test();
void db_disconnect();

#ifdef cplusplus
}
#endif
```

test_cpp.cpp – Основной код приложения, содержащий функцию main, а также, в данном примере, класс C++.

```
#include "test_mod.h"
class TestCpp
{
public: TestCpp(); void test();
~TestCpp();
};
TestCpp::TestCpp()
{
db_connect();
}
void TestCpp::test()
{
db_test();
}
TestCpp::~TestCpp()
{
db_disconnect();
}
int main(void)
{
TestCpp *t = new TestCpp();
t->test(); return 0;
}
```

Для сборки приложения необходимо:

– преобразовать test_mod.pgc в test_mod.c с помощью ecpg:

```
ecpg -o test_mod.c test_mod.pgc
```

– получить test_mod.o, скомпилировав test_mod.c компилятором C:

```
cc -c test_mod.c -o test_mod.o
```

– получить test_cpp.o, скомпилировав test_cpp.cpp компилятором C++:

```
c++ -c test_cpp.cpp -o test_cpp.o
```

– связать полученные объектные файлы, test_cpp.o и test_mod.o, в один исполняемый файл, выполнив компоновку под управлением компилятора C++:

```
c++ test_cpp.o test_mod.o -lecpg -o test_cpp
```

27313. Команды встраиваемого SQL

Команды, предназначенные специально для встраиваемого SQL:

– ALLOCATE DESCRIPTOR – выделить область SQL-дескриптора.

Синтаксис:

```
ALLOCATE DESCRIPTOR имя
```

Описание:

ALLOCATE DESCRIPTOR выделяет новую именованную область SQL-дескриптора, через которую можно обмениваться данными между сервером PG360 и программой на C.

После использования области дескрипторов должны освобождаться командой DEALLOCATE DESCRIPTOR.

Параметры:

– *имя* – Имя SQL-дескриптора, задаётся с учётом регистра. Это может быть идентификатор SQL или переменная среды C.

– CONNECT – установить подключение к базе данных.

Синтаксис:

```
CONNECT TO цель_подключения [ AS имя_подключения ] [ USER пользователь_подключения ] CONNECT TO DEFAULT
CONNECT пользователь_подключения
DATABASE цель_подключения
```

Описание:

Команда CONNECT устанавливает подключение клиента к серверу PG360. Параметры:

– *цель_подключения*

цель_соединения задаёт целевой сервер и базу для подключения в одной из нескольких форм.

[*имя_бд*] [*@сервер*] [:*порт*] – Подключение по TCP/IP.

unix:postgresql://*сервер* [:*порт*] / [*имя_бд*] [?*параметр_подключения*] – Подключение через Unix-сокеты.

tcp:postgresql://*сервер* [:*порт*] / [*имя_бд*] [?*параметр_подключения*] – Подключение по TCP/IP.

Строковая константа SQL – содержащая значение в одной из показанных выше форм.

переменная среды C – переменная среды C типа char[] или VARCHAR[], содержащая значение в одной из показанных выше форм

– *имя_подключения* – Необязательный идентификатор подключения, позволяющий обращаться к этому подключению в других командах. Это может быть идентификатор SQL или переменная среды C.

– *пользователь_подключения* – Имя пользователя для подключения к базе данных.

В этом параметре также можно передать имя и пароль одним из следующих способов: *имя_пользователя*/*пароль*, *имя_пользователя* IDENTIFIED RU *пароль* или *имя_пользователя* USING *пароль*.

В качестве имени пользователя и пароля можно задать идентификаторы SQL, строковые константы или переменные среды.

– DEFAULT – Использовать все параметры подключения по умолчанию, которые определены библиотекой libpq.

– DEALLOCATE DESCRIPTOR – освободить область SQL-дескриптора.

Синтаксис:

```
DEALLOCATE DESCRIPTOR имя
```

Описание:

DEALLOCATE DESCRIPTOR освобождает область именованного SQL-дескриптора.

Параметры:

– *имя* – Имя дескриптора, подлежащего освобождению, задаётся с учётом регистра. Это может быть идентификатор SQL или переменная среды C.

– DECLARE – определить курсор.

Синтаксис:

```
DECLARE имя_курсора [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH |WITHOUT } HOLD ] FOR подготовленный_оператор
DECLARE имя_курсора [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH | WITHOUT } HOLD ] FOR запрос
```

Описание:

DECLARE объявляет курсор для прохода по набору результатов подготовленного оператора. Эта команда несколько отличается от обычной SQL-команды DECLARE: тогда как последняя выполняет запрос и подготавливает набор результатов для получения, встраиваемая SQL-команда просто объявляет имя в качестве «переменной цикла» для прохода по набору результатов запроса; фактически запрос выполнится, когда курсор будет открыт командой OPEN.

Параметры:

– *имя_курсора* – Имя курсора, задаётся с учётом регистра. Это может быть идентификатор SQL или переменная среды C.

– *подготовленный_оператор* – Имя подготовленного запроса, задаваемое SQL-идентификатором или переменной среды.

– *запрос* – Команда SELECT или VALUES, выдающая строки, которые будут получены через курсор.

– DESCRIBE – получить информацию о подготовленном операторе или наборе результатов.

Синтаксис:

```
DESCRIBE [ OUTPUT ] подготовленный_оператор USING [ SQL ] DESCRIPTOR имя_дескриптора DESCRIBE [ OUTPUT ] подготовленный_оператор INTO [ SQL ] DESCRIPTOR имя_дескриптора DESCRIBE [ OUTPUT ] подготовленный_оператор INTO имя_sqlda
```

Описание:

DESCRIBE получает метаданные о результирующих столбцах, содержащихся в подготовленном операторе, не считывая собственно строки результата.

Параметры:

– *подготовленный_оператор* – Имя подготовленного оператора. Это может быть идентификатор SQL или переменная среды C.

– *имя_дескриптора* – Имя дескриптора, задаётся с учётом регистра. Это может быть идентификатор SQL или переменная среды C.

– *имя_sqlda* – Имя переменной SQLDA.

– DISCONNECT – закрыть подключение к базе данных.

Синтаксис:

```
DISCONNECT имя_подключения DISCONNECT [ CURRENT ] DISCONNECT DEFAULT DISCONNECT ALL
```

Описание:

DISCONNECT закрывает подключение (или все подключения) к базе данных.

Параметры:

– *имя_подключения* – Имя подключения к базе данных устанавливается командой CONNECT.

– CURRENT – Закрывает «текущее» подключение, то есть подключение, открытое последним, либо установленное командой SET CONNECTION. Текущее подключение подразумевается по умолчанию, если DISCONNECT выполняется без аргументов.

– DEFAULT – Закрывает подключение по умолчанию.

– ALL – Закрывает все открытые подключения.

– EXECUTE IMMEDIATE – динамически подготовить и выполнить оператор.

Синтаксис:

```
EXECUTE IMMEDIATE строка
```

Описание:

EXECUTE IMMEDIATE немедленно подготавливает и выполняет динамически задаваемый SQL– оператор, не получая при этом строки результата.

Параметры;

– *строка* – Строковая константа C или переменная среды C, содержащая SQL-оператор, который нужно выполнить.

– GET DESCRIPTOR – получить информацию из области дескриптора SQL.

Синтаксис:

```
GET DESCRIPTOR имя_дескриптора :cvariable = элемент_заголовка_дескриптора [, ... ]
GET DESCRIPTOR имя_дескриптора VALUE номер_столбца :cvariable = элемент_дескриптора [, ... ]
```

Описание:

GET DESCRIPTOR получает информацию о наборе результатов запроса из области дескриптора SQL и сохраняет её в переменные среды. Область дескриптора обычно предварительно заполняется командами FETCH или SELECT, чтобы из неё можно было перенести сопутствующую информацию в переменные среды.

Эта команда имеет две формы: первая форма выдаёт элементы из «заголовка» дескриптора, который относится ко всему набору результатов в целом. Например, это число строк. Другая форма, требующая указания в дополнительном параметре номера столбца, выдаёт информацию о конкретном столбце строки. В качестве примеров можно привести имя столбца и фактическое значение в этом столбце.

Параметры:

– *имя_дескриптора* – Имя дескриптора.

– *элемент_заголовка_дескриптора* – Идентификатор, определяющий, какой элемент заголовка нужно получить. В настоящее время поддерживается только COUNT, позволяющий получить число столбцов в наборе результатов.

– *номер_столбца* – Номер столбца, информацию о котором нужно получить. Нумерация начинается с 1.

– *элемент_дескриптора* – Идентификатор, определяющий, какой элемент информации о столбце нужно получить.

– *cvariable* – Переменная среды C, в которую будут сохранены данные, полученные из области дескриптора.

– OPEN – открыть динамический курсор.

Синтаксис:

```
OPEN имя_курсора
OPEN имя_курсора USING значение [, ... ]
OPEN имя_курсора USING SQL DESCRIPTOR имя_дескриптора
```

Описание:

OPEN открывает курсор и в дополнение может связывать фактические значения с местозаполнителями в объявлении курсора. Курсор должен быть предварительно объявлен командой DECLARE. Команда OPEN запускает выполнение запроса на сервере.

Параметры:

– *имя_курсора* – Имя открываемого курсора. Этот может быть идентификатор SQL или переменная среды.

– *значение* – Значение, связываемое с местозаполнителем в курсоре. Это может быть константа SQL, переменная среды C или переменная среды с индикатором.

– *имя_дескриптора* – Имя дескриптора, содержащего значения, которые должны быть связаны с местозаполнителями в курсоре. Это может быть идентификатор SQL или переменная среды C.

– PREPARE – подготовить оператор к выполнению.

Синтаксис:

```
PREPARE имя FROM строка
```

Описание:

Команда PREPARE подготавливает к выполнению динамический оператор, задаваемый в виде строки. Она отличается от обычного SQL-оператора PREPARE, который также можно использовать во встраиваемых командах. Для обоих типов подготовленных операторов применяется команда EXECUTE.

Параметры

– *подготовленный_оператор* – Идентификатор для подготовленного запроса.

– *строка* – Строковая константа C или переменная среды C, содержащая подготавливаемый оператор: SELECT, INSERT, UPDATE или DELETE.

– SET AUTOCOMMIT – установить режим автофиксации для текущего сеанса.

Синтаксис:

```
SET AUTOCOMMIT { = | TO } { ON | OFF }
```

Описание:

SET AUTOCOMMIT устанавливает режим автофиксации для текущего сеанса использования базы данных. По умолчанию программы со встраиваемым SQL работают *не* в режиме автофиксации, так что в определённые моменты нужно явно выполнять COMMIT. Эта команда может переключить сеанс в режим автофиксации, когда неявно фиксируется каждый отдельный оператор.

– SET CONNECTION – выбрать подключение к базе данных.

Синтаксис:

```
SET CONNECTION [ TO | = ] имя_подключения
```

Описание:

SET CONNECTION устанавливает «текущее» подключение к базе данных, которое будет использоваться командами, не задающими подключение явно.

Параметры:

– *имя_подключения* – Имя подключения к базе данных устанавливается командой CONNECT.

– DEFAULT – Устанавливает заданное подключение подключением по умолчанию.

– SET DESCRIPTOR – внести информацию в область дескриптора SQL.

Синтаксис:

```
SET DESCRIPTOR имя_дескриптора элемент_заголовка_дескриптора = значение [, ... ] SET DESCRIPTOR имя_дескриптора VALUE номер элемент_дескриптора = значение [, ...]
```

Описание:

SET DESCRIPTOR заполняет область SQL-дескриптора значениями. Заполненная область дескриптора обычно применяется для привязывания параметров при выполнении подготовленного запроса.

Эта команда имеет две формы: первая применяется к «заголовку» дескриптора, который не зависит от конкретных данных. Вторая форма устанавливает значения для определённых полей по номерам.

Параметры:

– *имя_дескриптора* – Имя дескриптора.

– *элемент_заголовка_дескриптора* – Идентификатор, определяющий, какой элемент заголовка нужно задать. В настоящее время поддерживается только COUNT, позволяющий задать число элементов в дескрипторе.

– *номер* – Номер элемента дескриптора, для которого задаётся значение. Нумерация начинается с 1.

– *элемент_дескриптора* – Идентификатор, определяющий, какой элемент нужно установить в дескрипторе.

– *значение* – Значение, которое нужно поместить в элемент дескриптора. Это может быть константа SQL или переменная среды C.

– TYPE – создать новый тип данных.

Синтаксис:

```
TYPE имя_типа IS тип_C
```

Описание:

Команда TYPE определяет новый тип C. Она равнозначна добавлению typedef в секции объявлений. Эта команда принимается, только когда ecpg запускается с параметром -c.

Параметры

– *имя_типа* – Имя нового типа. Это имя должно быть допустимым для типа в языке C.

– *тип_C* – Определение типа C.

– VAR – определить переменную.

Синтаксис:

```
VAR имя_переменной IS тип_C
```

Описание:

Команда VAR назначает переменной среды C новый тип данных C. Эта переменная среды должна быть объявлена ранее в секции объявлений.

Параметры

– *имя_переменной* – Имя переменной C.

– *тип_C* – Определение типа C.

– WHENEVER – определить действие, которое должно выполняться, когда при обработке SQL– оператора возникает определённое условие.

Синтаксис:

```
WHENEVER { NOT FOUND | SQLERROR | SQLWARNING } действие
```

Описание:

Устанавливает поведение в случае определённых условий (строки не найдены, выданы предупреждения или ошибки SQL и т. д.), возникающих в ходе выполнения SQL.

Параметры:

NOT FOUND – Указанное действие вызывается, когда SQL-оператор получает или обрабатывает ноль строк.

SQLERROR – Указанное действие вызывается, когда при выполнении SQL-оператора происходит ошибка.

SQLWARNING – Указанное действие вызывается, когда при выполнении SQL-оператора выдаётся предупреждение.

– действие – может быть следующим:

CONTINUE – Это фактически означает, что условие игнорируется. Это поведение по умолчанию.

GOTO метка GO TO метка Перейти к указанной метке (используя оператор goto языка C).

SQLPRINT – Вывести сообщение в устройство стандартного вывода.

STOP – Вызвать exit(1), что приведёт к завершению программы.

DO BREAK – Выполнить оператор break языка C. Этот вариант следует использовать только в циклах или операторах switch.

DO CONTINUE – Выполнить оператор continue языка C. Этот вариант следует использовать только в циклах. Данный оператор передаёт управление в начало цикла.

CALL имя (аргументы),

DO имя (аргументы) – Вызвать указанные функции C с заданными аргументами.

2.7.4. Информационная схема

Информационная схема состоит из набора представлений, содержащих информацию об объектах, определённых в текущей базе данных.

Информационная схема – это схема с именем information_schema. Данная схема автоматически доступна во всех базах данных. Владельцем этой схемы является начальный пользователь баз данных в кластере, и этот пользователь имеет все права в ней, включая возможность её удалить.

По умолчанию информационная схема отсутствует в пути поиска схем, так что ко всем объектам в ней нужно обращаться по полным именам.

Столбцы в представлениях информационной схемы имеют специальные типы данных, определённые в информационной схеме. Они определены как простые домены поверх обычных встроенных типов. Задействовать эти типы вне информационной схемы не следует.

Специальные типы данных, определённые в столбцах информационной схемы:

– cardinal_number – неотрицательное целое;

– character_data – строка символов (без определённого ограничения по длине);

– sql_identifier – строка символов. Этот тип применяется для идентификаторов SQL, тогда как тип character_data для всех остальных видов текстовых данных;

– time_stamp – домен на базе типа timestamp with time zone;

– yes_or_no – домен символьной строки, который принимает либо YES, либо NO. Этот домен применяется для представления булевых данных (истина/ложь, true/false) в информационной схеме.

Все столбцы в информационной схеме имеют один из этих пяти типов.

Подробное описание информационных схем PG360 приведено в Приложении А.

2.8. Серверное программирование

Эта часть документации посвящена расширению функциональности сервера путём реализации собственных функций, типов данных, триггеров и т. д. Также описываются языки программирования на стороне сервера, поддерживаемые дистрибутивом PG360, и рассматриваются общие вопросы, связанные с программированием на стороне сервера.

2.8.1. Расширение SQL

Далее описано, как в PG360 можно расширять язык запросов SQL, добавляя собственные:

– функции;
– агрегатные функции;
– типы данных;
– операторы;
– классы операторов для индексов;
– пакеты связанных объектов.

2.8.1.1. Как реализована расширяемость

PG360 является расширяемым благодаря тому, что его работа управляется каталогами. Эти каталоги представляются пользователю в виде таблиц, подобных любым другим, но СУБД ведёт в них свои внутренние записи. Ключевое отличие PG360 от обычных реляционных СУБД состоит в том, что PG360 хранит в этих каталогах намного больше информации: информацию не только о таблицах и столбцах, но также о типах данных, функциях, методах доступа и т.д. Эти таблицы могут быть изменены пользователями, а так как PG360 в своих действиях руководствуется этими таблицами, это означает, что пользователи могут расширять PG360. Обычные же СУБД можно расширять, только модифицируя жёстко запрограммированные процедуры в исходном коде или загружая модули, специально разработанные производителем СУБД.

Кроме того, сервер PG360 может динамически загружать в свой процесс код, написанный пользователем. То есть, пользователь может подключить файл с объектным кодом (например, разделяемую библиотеку), который реализует новый тип или функцию, а PG360 загрузит его по мере надобности. Код, написанный на SQL, добавляется на сервер ещё проще. Эта способность менять своё поведение «на лету» делает PG360 исключительно подходящим для быстрого прототипирования новых приложений и структур хранения.

2.8.1.2. Система типов PG360

Типы данных PG360 делятся на базовые, типы-контейнеры, составные, доменные и псевдотипы.

Базовые типы

Базовые типы — это типы, такие как integer, которые реализуются ниже уровня языка SQL (обычно на низкоуровневом языке, например C). В общих чертах они соответствуют так называемым абстрактным типам данных. PG360 может работать с такими типами только

через функции, предоставленные пользователем, и понимать их поведение только в той степени, в какой его опишет пользователь.

Типы-перечисления (enum) можно считать подкатегорией базовых типов. Они отличаются от других типов тем, что их можно создавать просто командами SQL, обходясь без низкоуровневого программирования.

Типы-контейнеры

В PG360 есть три вида «типов-контейнеров», то есть типов, которые могут содержать в себе несколько значений других типов. Это массивы, составные типы и диапазоны.

Массивы могут содержать множество значений, имеющих один тип. Тип массива автоматически создаётся для каждого базового и составного типа, диапазона и домена, но не для массивов — массивы массивов не существуют. Для системы типов многомерные массивы не отличаются от одномерных.

Составные типы, или типы строк, образуются при создании любой таблицы. С помощью команды CREATE TYPE также можно определить «независимый» составной тип, не связанный с таблицей. Составной тип представляет собой просто список типов с определёнными именами полей. Значением составного типа является строка таблицы или запись из значений полей.

Диапазонный тип может содержать два значения одного типа, которые определяют нижнюю и верхнюю границу диапазона. Диапазонные типы создаются пользователем, хотя существует и несколько встроенных.

Домены

Домен основывается на определённом нижележащем типе и во многих аспектах взаимозаменяем с ним. Однако домен может иметь ограничения, уменьшающие множество допустимых для него значений относительно нижележащего типа. Домены создаются SQL-командой CREATE DOMAIN.

Псевдотипы

Для специальных целей существует также несколько «псевдотипов». Псевдотипы нельзя задействовать в столбцах таблицы или в типах-контейнерах, но их можно использовать в объявлениях аргументов и результатов функций. Это даёт возможность выделить в системе типов специальные классы функций.

Полиморфные типы

Особый интерес представляет подмножество псевдотипов, *полиморфные типы*, которые применяются в объявлениях *полиморфных функций*. Используя такие типы, можно объявить всего одну функцию, которая будет работать с разными типами данных, определяя конкретные типы в зависимости от того, значения каких типов были переданы ей при вызове.

Таблица 3. Полиморфные типы

Имя	**Семейство**	**Описание**
anyelement	Простое	Указывает, что функция принимает любой тип
anyarray	Простое	Указывает, что функция принимает любой тип массива
anynonarray	Простое	Указывает, что функция принимает любой тип, отличный от массива
anyenum	Простое	Указывает, что функция принимает любой тип-перечисление
anyrange	Простое	Указывает, что функция принимает любой диапазонный тип
anycompatible	Общее	Указывает, что функция принимает любой тип, с автоматическим приведением нескольких аргументов к общему типу
anycompatiblearray	Общее	Указывает, что функция принимает любой тип массива, с автоматическим приведением нескольких аргументов к общему типу
anycompatiblenonarray	Общее	Указывает, что функция принимает любой тип, отличный от массива, с автоматическим приведением нескольких аргументов к общему типу
anycompatiblerange	Общее	Указывает, что функция принимает любой диапазонный тип, с автоматическим приведением нескольких аргументов к общему типу

Полиморфные аргументы и результаты связаны друг с другом и сводятся к конкретным типам данным при разборе запроса, вызывающего полиморфную функцию. Когда полиморфных аргументов несколько, фактические типы данных входных значений должны совмещаться, как описано далее. Если тип результата функции полиморфный или у неё имеются выходные параметры полиморфных типов, фактические типы этих результатов выводятся из типов полиморфных входных значений, как описано ниже.

Для «простого» семейства полиморфных типов действуют следующие правила совмещения и выведения типов:

В каждой позиции (в аргументах или возвращаемом значении), объявленной как anyelement, может передаваться любой фактический тип данных, но в каждом конкретном вызове все эти фактические типы должны быть *одинаковыми*. Аналогичным образом, в каждой позиции, объявленной как anyarray, может передаваться любой тип данных массива, но все фактические типы должны совпадать. Так же и во всех позициях, объявленных как anyrange, должен передаваться одинаковый диапазонный тип. Более того, если некоторые позиции объявлены как anyarray, а другие как anyelement, то фактическим типом в позициях anyarray должен быть массив, элементы которого имеют тот же тип, что и значения в позициях anyelement. Подобным образом, если одни позиции объявлены как anyrange, а другие как anyelement или anyarray, фактическим типом в позициях anyrange должен быть диапазон, подтип которого совпадает с типом элементов в позициях anyelement и с типом, передаваемым в позициях anyarray. Псевдотип anynonarray обрабатывается так же, как anyelement, но с дополнительным ограничением — фактический тип не должен быть типом массива. Псевдотип anyenum тоже обрабатывается как anyelement, но его фактические типы ограничиваются перечислениями.

Таким образом, когда с полиморфным типом объявлено несколько аргументов, в итоге допускаются только определённые комбинации фактических типов. Например, функция, объявленная как equal(anyelement, anyelement), примет в аргументах любые два значения, но только если их типы данных совпадают.

Когда с полиморфным типом объявлено возвращаемое значение функции, так же полиморфным должен быть минимум один аргумент, и фактический тип результата при конкретном вызове определится по типу фактически переданного полиморфного аргумента (или аргументов).

В большинстве случаев при разборе функции фактический тип данных для полиморфного результата может быть выведен из аргументов, имеющих другой полиморфный тип из того же семейства; например, подтип anyarray может выводиться из anyelement и наоборот. Исключение представляет полиморфный результат типа anyrange — для него требуется аргумент типа anyrange; вывести его фактический тип из типа аргументов anyarray или anyelement нельзя. Это объясняется тем, что на одном подтипе могут базироваться несколько диапазонных типов.

Типы anynonarray и anyenum представляют не отдельные типы переменных; это те же типы, что и anyelement, но с дополнительными ограничениями. Например, объявление функции f(anyelement, anyenum) равнозначно объявлению f(anyenum, anyenum): оба фактических аргумента должны быть одинаковыми типами-перечислениями.

Для «общего» семейства полиморфных типов работают примерно та же правила совмещения и выведения типов, что и для «простого» семейства, но есть одно важно отличие: фактические типы аргументов не должны совпадать, если они могут быть неявно приведены к некоторому общему типу. Этот общий тип выбирается по тем же правилам, что применяются в UNION и подобных конструкциях. При выборе общего типа учитываются фактические типы аргументов anycompatible и anycompatiblenonarray, типы элементов в аргументах anycompatiblearray и подтипы диапазонов в аргументах anycompatiblerange. Если присутствует тип anycompatiblenonarray, общим типом не должен быть тип массива. После того как общий тип определён, аргументы anycompatible и anycompatiblenonarray автоматически приводятся к этому типу, а аргументы anycompatiblearray приводятся к типу-массиву с элементами этого типа.

Так как выбрать диапазонный тип, зная только его подтип, невозможно, в случае использования anycompatiblerange требуется, чтобы все аргументы, объявленные с этим типом, имели один диапазонный тип, а его подтип согласовывался с выбранным общим типом, что позволяет обойтись без приведения типов для диапазонных значений. Как и anyrange, anycompatiblerange можно применить в качестве типа результата функции, только если у неё имеется аргумент такого же типа (anycompatiblerange).

Полиморфные семейства «простое» и «общее» представляют два независимых набора переменных типов.

Функции с переменным числом аргументом тоже могут быть полиморфными: для этого их последний параметр описывается как VARIADIC anyarray или VARIADIC anycompatiblearray. Для целей сопоставления аргументов и определения фактического типа результата такая функция представляется так же, как если бы в ней явно объявлялось нужное число параметров anynonarray или anycompatiblenonarray.

28.1.3. Пользовательские функции

В PG360 представлены функции четырёх видов:

– функции на языке запросов (функции, написанные на SQL);

– функции на процедурных языках (функции, написанные, например, на PL/pgSQL или PL/Tcl);

– внутренние функции;

– функции на языке C.

Функции любых видов могут принимать в качестве аргументов (параметров) базовые типы, составные типы или их сочетания. Кроме того, любые функции могут возвращать значения базового или составного типа. Также можно определить функции, возвращающие наборы базовых или составных значений.

Функции многих видов могут также принимать или возвращать определённые псевдотипы (например, полиморфные типы), но доступные средства для работы с ними различаются.

Многие концепции, касающиеся функций на SQL, затем распространятся и на другие виды функций.

28.1.4. Пользовательские процедуры

Процедура — объект базы данных, подобный функции, но имеющий следующие отличия:

– Процедуры определяются командой CREATE PROCEDURE, а не CREATE FUNCTION.

– Процедуры, в отличие от функций, не возвращают значение; поэтому в CREATE PROCEDURE отсутствует предложение RETURNS. Однако процедуры могут выдавать данные в вызывающий код через выходные параметры.

– Функции вызываются как часть запроса или команды DML, а процедуры вызываются отдельно командой CALL.

– Процедура, в отличие от функции, может фиксировать или откатывать транзакции во время её выполнения (а затем автоматически начинать новую транзакцию), если вызывающая команда CALL находится не в явном блоке транзакции.

Некоторые атрибуты функций (например, STRICT) неприменимы к процедурам. Эти атрибуты влияют на вызов функций в запросах и не имеют отношения к процедурам.

Далее описано, как определять пользовательские функции, что также применимо к процедурам, с учётом вышеперечисленных особенностей.

Функции и процедуры в совокупности также называются *подпрограммами*. Существуют команды, такие как ALTER ROUTINE и DROP ROUTINE, которые способны работать и с функциями, и с процедурами, не требуя указания точного вида объекта.

28.1.5. Функции на языке запросов (SQL)

SQL-функции выполняют произвольный список операторов SQL и возвращают результат последнего запроса в списке. В простом случае (не с множеством) будет возвращена первая строка результата последнего запроса. (Понятие «первая строка» в наборе результатов с несколькими строками определено точно, только если присутствует ORDER BY.) Если последний запрос вообще не вернёт строки, будет возвращено значение NULL.

Кроме того, можно объявить SQL-функцию как возвращающую множество (то есть, несколько строк), указав в качестве возвращаемого типа функции SETOF *некий_тип*, либо объявив её с указанием RETURNS TABLE(*столбцы*). В этом случае будут возвращены все строки результата последнего запроса. Подробнее это описывается ниже.

Тело SQL-функции должно представлять собой список SQL-операторов, разделённых точкой с запятой. Точка с запятой после последнего оператора может отсутствовать. Если только функция не объявлена как возвращающая void, последним оператором должен быть SELECT, либо INSERT, UPDATE или DELETE с предложением RETURNING.

Любой набор команд на языке SQL можно скомпоновать вместе и обозначить как функцию. Помимо запросов SELECT, эти команды могут включать запросы, изменяющие данные (INSERT, UPDATE и DELETE), а также другие SQL-команды. (В SQL-функциях нельзя использовать команды управления транзакциями, например COMMIT, SAVEPOINT, и некоторые вспомогательные команды, в частности VACUUM.) Однако последней командой должна быть SELECT или команда с предложением RETURNING, возвращающая результат с типом возврата функции. Если необходимо определить функцию SQL, выполняющую действия, но не возвращающую значение, можно объявить её как возвращающую тип void.

Синтаксис команды CREATE FUNCTION требует, чтобы тело функции было записано как строковая константа. Обычно для этого удобнее всего заключать строковую константу в доллары. Если есть необходимость использовать обычный синтаксис с заключением строки в апострофы, то необходимо дублировать апострофы (') и обратную косую черту (\) (предполагается синтаксис спецпоследовательностей) в теле функции.

28.15.1. Аргументы SQL-функций

К аргументам SQL-функции можно обращаться в теле функции по именам или номерам. Ниже приведены примеры обоих вариантов.

Чтобы использовать имя, необходимо объявить аргумент функции как именованный, а затем прописать это имя в теле функции. Если имя аргумента совпадает с именем какого-либо столбца в текущей SQL-команде внутри функции, имя столбца будет иметь приоритет. Чтобы перекрыть имя столбца, необходимо дополнить имя аргумента именем самой функции, то есть записать его в виде *имя_функции.имя_аргумента*. (Если и это имя будет конфликтовать с полным именем столбца, снова выиграет имя столбца. Неоднозначности в этом случае можно избежать, выбрав другой псевдоним для таблицы в SQL-команде.)

Старый подход с нумерацией позволяет обращаться к аргументам, применяя запись $*n*: $1 обозначает первый аргумент, $2 — второй и т. д. Это будет работать и в том случае, если данному аргументу назначено имя.

Если аргумент имеет составной тип, то для обращения к его атрибутам можно использовать запись с точкой, например: *аргумент.поле* или $1.*поле*. И опять же, при этом может потребоваться дополнить имя аргумента именем функции, чтобы сделать имя аргумента однозначным.

Аргументы SQL-функции могут использоваться только как значения данных, но не как идентификаторы.

28.1.6. Перегрузка функций

Вы можете определить несколько функций с одним именем SQL, если эти функции будут принимать разные аргументы. Другими словами, имена функций можно *перегружать*.

Независимо от того, использовать эту возможность или нет, она требует предосторожности при вызове функций в базах данных, где одни пользователи не доверяют другим. Когда выполняется запрос, сервер определяет, какую именно функцию вызывать, по количеству и типам представленных аргументов. Перегрузка может использоваться для имитации функций с переменным количеством аргументов, до какого-то конечного числа.

Функции, принимающей один аргумент составного типа, обычно не следует давать имя, совпадающее с именем какого-либо атрибута (поля) этого типа. Запись *атрибут*(*таблица*) считается равнозначной *таблица*.*атрибут*. В случае, когда возникает неоднозначность между функцией, принимающей составной тип, и атрибутом составного типа, всегда будет выбираться атрибут. Этот выбор можно переопределить, дополнив имя функции схемой (то есть, записав *схема*.*функция*(*таблица*)), но лучше избежать этой проблемы, подобрав разные имена.

Другой тип конфликта возможен между обычными функциями и функциями с переменными параметрами. Например, можно создать функции foo(numeric) и foo(VARIADIC numeric[]). В этом случае будет непонятно, какая функция должна выбираться при передаче одного числового аргумента, например foo(10.1). При разрешении этого конфликта предпочтение отдаётся функции, найденной первой по пути поиска, либо, если две функции находятся в одной схеме, выбирается функция с постоянными аргументами.

При перегрузке функций на языке C есть дополнительное ограничение: имя уровня C каждой функции в семействе перегруженных функций должно отличаться от имён уровня C всех других функций, как внутренних, так и загружаемых динамически. Если это правило нарушается, поведение зависит от среды. Можно получить ошибку компоновщика во время выполнения, либо будет вызвана не та функция (обычно внутренняя). Альтернативная форма предложения AS для SQL-команды CREATE FUNCTION позволяет отвязать имя SQL-функции от имени, определённого в исходном коде на C.

28.1.7. Категории изменчивости функций

Для каждой функции определяется характеристика *изменчивости*, с возможными вариантами: VOLATILE, STABLE и IMMUTABLE. Если эта характеристика не задаётся явно в команде CREATE FUNCTION, по умолчанию подразумевается VOLATILE. Категория изменчивости представляет собой обещание некоторого поведения функции для оптимизатора:

– Изменчивая функция (VOLATILE) может делать всё, что угодно, в том числе, модифицировать базу данных. Она может возвращать различные результаты при нескольких вызовах с одинаковыми аргументами. Оптимизатор не делает никаких предположений о поведении таких функций. В запросе, использующем изменчивую функцию, она будет вычисляться заново для каждой строки, когда потребуется её результат.

– Стабильная функция (STABLE) не может модифицировать базу данных и гарантированно возвращает одинаковый результат, получая одинаковые аргументы, для всех строк в одном операторе. Эта характеристика позволяет оптимизатору заменить множество вызовов этой функции одним. В частности, выражение, содержащее такую функцию, можно безопасно

использовать в условии поиска по индексу. (Так как при поиске по индексу целевое значение вычисляется только один раз, а не для каждой строки, использовать функцию с характеристикой VOLATILE в условии поиска по индексу нельзя.)

– Постоянная функция (IMMUTABLE) не может модифицировать базу данных и гарантированно всегда возвращает одинаковые результаты для одних и тех же аргументов. Эта характеристика позволяет оптимизатору предварительно вычислить функцию, когда она вызывается в запросе с постоянными аргументами.

Для наилучших результатов оптимизации, функции следует назначать самую строгую характеристику изменчивости, которой она соответствует.

Любая функция с побочными эффектами *должна* быть помечена как VOLATILE, чтобы обращения к ней не исключались при оптимизации. Даже если функция не имеет побочных эффектов, её нужно пометить как VOLATILE, если её значение может меняться при выполнении одного запроса; таковы функции random(), currval() и timeofday().

Другой важный пример представляет семейство функций current_timestamp, которые имеют характеристику STABLE, потому что их значения не меняются в рамках одной транзакции.

Характеристики STABLE и IMMUTABLE мало различаются, когда речь идёт о простых интерактивных запросах, которые планируются и сразу же выполняются; не имеет большого значения, будет ли функция выполнена однократно на этапе планирования или в начале выполнения. Существенное различие проявляется, когда план сохраняется и многократно используется позже. Если функция помечена как IMMUTABLE, тогда как на самом деле она не является постоянной, она может быть сведена к константе во время планирования, так что при последующих выполнениях плана вместо неё будет использоваться неактуальное значение. Это опасно при использовании подготовленных операторов или языков функций, кеширующих планы (например, PL/pgSQL).

У функций, написанных на SQL или на любом другом стандартном процедурном языке, есть ещё одно важное свойство, определяемое характеристикой изменчивости, а именно видимость изменений, произведённых командой SQL, которая вызывает эту функцию. Функция VOLATILE будет видеть такие изменения, тогда как STABLE и IMMUTABLE — нет. Это поведение реализуется посредством снимков в MVCC: STABLE и IMMUTABLE используют снимок, полученный в начале вызывающего запроса, тогда как функции VOLATILE получают свежий снимок в начале каждого запроса, который они выполняют.

Вследствие такой организации работы со снимками, функцию, содержащую только команды SELECT, можно безопасно пометить как STABLE, даже если она выбирает данные из таблиц, которые могут быть изменены параллельными запросами. PG360 выполнит все команды в функции STABLE со снимком, полученным для вызывающего запроса, так что они будут видеть одно представление базы данных на протяжении всего запроса.

То же самое поведение со снимками распространяется на команды SELECT в функциях IMMUTABLE. Вообще в функциях IMMUTABLE обычно неразумно выбирать данные из таблиц, так как «постоянство» функции будет нарушено, если содержимое таблиц изменится.

Одна из распространённых ошибок — помечать функцию как IMMUTABLE, при том, что её результаты зависят от параметра конфигурации. Например, функция, работающая с временем,

может выдавать результаты, зависящие от параметра TimeZone. Для надёжности такие функции следует помечать как STABLE.

28.1.8. Функции на процедурных языках

PG360 позволяет разрабатывать собственные функции и на языках, отличных от SQL и C. Эти другие языки в целом обычно называются *процедурными языками* (PL, Procedural Languages). Процедурные языки не встроены в сервер PG360; они предлагаются загружаемыми модулями.

28.1.9. Внутренние функции

Внутренние функции — это функции, написанные на языке C, и статически скомпонованные в исполняемый код сервера PG360. В «теле» определения функции задаётся имя функции на уровне C, которое не обязательно должно совпадать с именем, объявленным для использования в SQL. (Обратной совместимости ради, тело функции может быть пустым, что будет означать, что имя функции на уровне C совпадает с именем в SQL.)

Обычно все внутренние функции, представленные на сервере, объявляются в ходе инициализации кластера баз данных, но пользователь может воспользоваться командой CREATE FUNCTION и добавить дополнительные псевдонимы для внутренней функции. Внутренние функции объявляются в CREATE FUNCTION с именем языка internal.

28.1.10. Функции на языке C

Пользовательские функции могут быть написаны на C (или на языке, который может быть совместим с C, например C++). Такие функции компилируются в динамически загружаемые объекты (также называемые разделяемыми библиотеками) и загружаются сервером по требованию. Именно метод динамической загрузки отличает функции «на языке C» от «внутренних» функций — правила написания кода по сути одни и те же.

В настоящее время для функций на C применяется только одно соглашение о вызовах («версии 1»). Поддержка этого соглашения обозначается объявлением функции с макросом (PG_FUNCTION_INFO_V1), как показано ниже.

28.1.10.1. Динамическая загрузка

В первый раз, когда в сеансе вызывается пользовательская функция в определённом внешнем объектном файле, загрузчик динамических модулей загружает этот файл в память, чтобы можно было вызвать эту функцию. Таким образом, в команде CREATE FUNCTION, объявляющей пользовательскую функцию на языке C, необходимо определить две сущности для функции: имя загружаемого объектного файла и имя уровня C (символ для компоновки) заданной функции в этом объектном файле. Если имя уровня C не указано явно, предполагается, что оно совпадает с именем функции в SQL.

Для нахождения разделяемого объектного файла по имени, заданному в команде CREATE FUNCTION, применяется следующий алгоритм:

– Если имя задаётся абсолютным путём, загружается заданный файл.

– Если имя начинается со строки $libdir, эта часть пути заменяется путём к каталогу библиотек PG360, который определяется во время сборки.

– Если в имени не указывается каталог, поиск файла производится по пути, заданному конфигурационной переменной dynamic_library_path.

– В противном случае (файл не был найден в пути поиска, или в его имени указывается не абсолютный путь к каталогу), загрузчик попытается принять имя как есть, что, скорее всего, не увенчается успехом. (Полагаться на текущий рабочий каталог ненадёжно.)

Если эта последовательность не даёт положительный результат, к данному имени добавляется принятое на данной платформе расширение файлов библиотек (часто .so) и последовательность повторяется снова. Если и это не приводит к успеху, происходит сбой загрузки.

Для поиска разделяемых библиотек рекомендуется задавать либо путь относительно $libdir, либо путь динамических библиотек. Это упрощает обновление версии при перемещении новой инсталляции в другое место. Какой именно каталог подразумевается под $libdir, можно узнать с помощью команды pg_config --pkglibdir.

Пользователь, от имени которого работает сервер PG360, должен иметь возможность пройти путь к файлу, который требуется загрузить. Очень распространённая ошибка — когда сам файл или каталог верхнего уровня оказывается недоступным для чтения и/или исполнения для пользователя postgres.

В любом случае имя файла, заданное в команде CREATE FUNCTION, записывается в системные каталоги буквально, так что если этот файл потребуется загрузить ещё раз, та же процедура будет проделана снова.

Чтобы гарантировать, что динамически загружаемый объектный файл не будет загружен несовместимым сервером, PG360 проверяет, содержит ли этот файл «отличительный блок» с требуемым содержимым. Чтобы включить его в модуль, необходимо написать в одном (и только одном) из исходных файлов модуля, после включения заголовочного файла fmgr.h:

PG_MODULE_MAGIC;

После того как он был использован первый раз, динамически загружаемый объектный файл сохраняется в памяти. Следующие обращения в том же сеансе к функциям в этом файле повлекут только небольшие издержки, связанные с поиском в таблице символов. Если нужно принудительно перезагрузить объектный файл, например, после перекомпиляции, необходимо начать новый сеанс.

Динамически загружаемый файл может дополнительно содержать функции инициализации и завершения работы библиотеки. Если в файле находится функция с именем _PG_init, эта функция будет вызвана сразу после загрузки файла. Эта функция не принимает параметры и не должна ничего возвращать. Если в файле находится функция _PG_fini, эта функция будет вызвана непосредственно перед выгрузкой файла. Эта функция так же не принимает параметры и не должна ничего возвращать. _PG_fini будет вызываться только при выгрузке файла, но не при завершении процесса.

28.1.102. Базовые типы в функциях на языке C

Для написания функций на языке C, необходимо знать, как внутри PG360 представляются базовые типы данных и как их могут принимать и передавать функции. PG360 внутри воспринимает базовые типы как «блоки памяти». Пользовательские функции, устанавливаемые для типов, в свою очередь, определяют, как PG360 может работать с этими типами. То есть, PG360 только сохраняет и загружает данные с диска, а для ввода, обработки и вывода данных он использует определяемые вами функции.

Базовые типы могут иметь один из трёх внутренних форматов:

– передаётся по значению, фиксированной длины;

– передаётся по ссылке, фиксированной длины;

– передаётся по ссылке, переменной длины.

Типы, передаваемые по значению, могут иметь размер только 1, 2 или 4 байта (и 8 байт, если sizeof(Datum) равен 8 на ПК). Определяя собственные типы, следует позаботиться о том, чтобы они имели одинаковый размер (в байтах) во всех архитектурах. Например, тип long опасен, так как он имеет размер 4 байта на одних ПК, и 8 байт на других, тогда как тип int состоит из 4 байт в большинстве систем Unix.

В функции PG360 и из них могут передаваться только указатели на такие типы. Чтобы вернуть значение такого типа, необходимо выделить для него нужное количество памяти функцией palloc, заполнить выделенную память и вернуть указатель на неё.

Все типы переменной длины также должны передаваться по ссылке. Все типы переменной длины должны начинаться с обязательного поля длины размером ровно 4 байта, которая будет задаваться макросом SET_VARSIZE; нельзя устанавливать это поле вручную. Все данные, которые будут храниться в этом типе, должны размещаться в памяти непосредственно за этим полем длины. Поле длины содержит полную длину структуры, то есть включает размер самого поля длины.

Ещё один важный момент — стараться не оставлять неинициализированных байт в значениях данных; например, обнулять все возможные байты выравнивания, которые могут присутствовать в структурах. Если этого не делать, логически равные значения данных могут представляться неравными планировщику, что приведёт к построению неэффективных (хотя и корректных) планов.

В качестве примера можно определить тип text следующим образом:

```
typedef struct { int32 length;
char data[FLEXIBLE_ARRAY_MEMBER];
} text;
```

Запись [FLEXIBLE_ARRAY_MEMBER] означает, что действительная длина массива данных в этом объявлении не указывается.

Работая с типами переменной длины, необходимо аккуратно выделить нужный объём памяти и записать его размер в поле длины.

В Таблице 4 указано, какие типы языка C соответствуют типам SQL при написании функций на C с использованием встроенных типов PG360. В столбце «Определён в» указывается, какой заголовочный файл необходимо подключить, чтобы получить определение типа. (Фактическое определение может быть в другом файле, который подключается из

указанного, однако рекомендуется придерживаться обозначенного интерфейса.) В любом исходном файле первым всегда необходимо включать postgres.h, потому что в нём объявляется ряд вещей, которые нужны в любом случае, и потому что включение первым другого заголовочного файла может сделать код непереносимым.

Таблица 4. Типы C, эквивалентные встроенным типам SQL

Тип SQL	Тип C	Определён в
boolean	bool	postgres.h (может быть встроен в компиляторе)
box	BOX*	utils/geo_decls.h
bytea	bytea*	postgres.h
"char"	char	(встроен в компиляторе)
character	BpChar*	postgres.h
cid	CommandId	postgres.h
date	DateADT	utils/date.h
float4 (real)	float4	postgres.h
float8 (double precision)	float8	postgres.h
int2 (smallint)	int16	postgres.h
int4 (integer)	int32	postgres.h
int8 (bigint)	int64	postgres.h
interval	Interval*	datatype/timestamp.h
lseg	LSEG*	utils/geo_decls.h
name	Name	postgres.h
numeric	Numeric	utils/numeric.h
oid	Oid	postgres.h
oidvector	oidvector*	postgres.h
path	PATH*	utils/geo_decls.h
point	POINT*	utils/geo_decls.h
regproc	RegProcedure	postgres.h
text	text*	postgres.h
tid	ItemPointer	storage/itemptr.h
time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
timestamp	Timestamp	datatype/timestamp.h
timestamp with time zone	TimestampTz	datatype/timestamp.h
varchar	VarChar*	postgres.h
xid	TransactionId	postgres.h

28.1.103. Соглашение о вызовах версии 1

Соглашение о вызовах версии 1 полагается на макросы, скрывающие основную долю сложностей, связанных с передачей аргументов и результатов. По соглашению версии 1 функция на C должна всегда определяться следующим образом:

```
Datum funcname(PG_FUNCTION_ARGS)
```

В дополнение к этому, в том же исходном файле должен присутствовать вызов макроса:

```
PG_FUNCTION_INFO_V1(funcname);
```

(Обычно его принято записывать непосредственно перед функцией.) Этот вызов макроса не нужен для функций internal, так как PG360 предполагает, что все внутренние функции используют соглашении версии 1. Однако для функций, загружаемых динамически, этот макрос необходим.

В функции версии 1 каждый аргумент выбирается макросом PG_GETARG_*xxx*(), который соответствует типу данных аргумента. В нестрогих функциях этому вызову должна предшествовать проверка на NULL в аргументе с использованием PG_ARGISNULL() (см. ниже). Результат возвращается макросом PG_RETURN_*xxx*() для возвращаемого типа. PG_GETARG_*xxx*() принимает в качестве параметра номер выбираемого аргумента функции (нумерация начинается с 0). PG_RETURN_*xxx*() принимает фактическое значение, которое нужно возвратить.

28.1.104. Написание кода

Далее описаны некоторые правила написания кода функций на языке C для PG360. Хотя можно загружать в PG360 функции, написанные на языках, отличных от C, обычно это довольно сложно (когда вообще возможно), так как другие языки, например C++, FORTRAN или Pascal часто не следуют соглашениям, принятым в C. То есть другие языки могут передавать аргументы и возвращаемые значения между функциями разными способами. Поэтому далее предполагается, что функции на языке C действительно написаны на C.

Основные правила написания и компиляции функций на C таковы:

– Чтобы выяснить, где находятся заголовочные файлы сервера PG360, установленные в системе (или в системе, с которой будут работать пользователи), необходимо воспользоваться командой pg_config --includedir-server.

– Для компиляции и компоновки кода, который можно будет динамически загрузить в PG360, требуется указать специальные флаги.

– Определить «отличительный блок» для разделяемой библиотеки.

– Для выделения памяти использовать функцию PG360 palloc, а для освобождения pfree, вместо соответствующих функций библиотеки C malloc и free. Память, выделяемая функцией palloc, будет автоматически освобождаться в конце каждой транзакции, во избежание утечек памяти.

– Всегда обнулять байты структур, применяя memset (или сразу выделить память функцией palloc0). Даже если присвоить значение каждому полю структуры, в ней могут оставаться байты выравнивания (пустоты в структуре), содержащие случайные значения. Если исключить это требование, будет сложно поддерживать индексы или соединение по хешу, так как для вычисления хеша придётся выбирать только значащие биты из структуры данных. Планировщик также иногда полагается на побитовое сравнение констант, так что результаты планирования

могут оказаться неожиданными, если логически равные значения окажутся неравными на битовом уровне.

Большинство внутренних типов PG360 объявлены в postgres.h, тогда как интерфейс менеджера функций (PG_FUNCTION_ARGS и т. д.) определён в fmgr.h, так что потребуется подключить как минимум два этих файла. По соображениям портируемости, лучше включить postgres.h *первым*, до каких-либо других системных или пользовательских файлов заголовков. При подключении postgres.h автоматически также будут подключены elog.h и palloc.h.

Имена символов, определённые в объектных файлах, не должны конфликтовать друг с другом или с именами других символов, определённых в исполняемых файлах сервера PG360.

28.1.105. Компиляция и компоновка динамически загружаемых функций

Прежде чем использовать написанные на C функции, расширяющие возможности PG360, их необходимо скомпилировать и скомпоновать особым образом, чтобы сервер мог динамически загрузить полученный файл. Точнее говоря, вам необходимо создать *разделяемую библиотеку*.

Создание разделяемых библиотек не отличается от сборки исполняемых файлов: сначала исходные файлы компилируются в объектные, а затем объектные связываются вместе. Объектные файлы должны создаваться так, чтобы они содержали *позиционно-независимый* код (PIC, position-independent code), что означает, что при загрузке для выполнения этот код может быть помещён в любое место в памяти. (Объектные файлы, предназначенные для сборки непосредственно исполняемых файлов, обычно собираются не так.) Команда для компоновки разделяемой библиотеки принимает специальные флаги, что отличают её от компоновки исполняемого файла (по крайней мере в теории — в некоторых системах реальность не так прекрасна).

В следующем примере предполагается, что исходный код находится в файле foo.c и будет создаваться разделяемая библиотека foo.so. Промежуточный объектный файл будет называться foo.o, если не отмечено другое. Разделяемая библиотека может включать больше одного объектного файла.

В Linux для создания кода PIC компилятору передаётся флаг -fPIC. Для создания разделяемой библиотеки компилятору передаётся флаг -shared. Полный пример будет выглядеть следующим образом:

```
cc -fPIC -c foo.c
cc -shared -o foo.so foo.o
```

28.1.106. Аргументы составного типа

Составные типы не имеют фиксированного макета данных, как структуры C. В частности, экземпляры составного типа могут содержать поля NULL. Кроме того, в контексте наследования составные типы могут иметь разные поля для разных членов в одной иерархии наследования. Поэтому PG360 предоставляет функциям специальный интерфейс для обращения к полям составных типов из C.

28.1.10.7. Возврат строк (составных типов)

Чтобы вернуть строку или значение составного типа из функции на языке C, можно использовать специальный API, предоставляющий макросы и функции, скрывающие основную сложность формирования составных типов данных. Для использования этого API необходимо включить в исходный файл:

#include "funcapi.h"

Сформировать значение составного типа (далее «кортеж») можно двумя способами: его можно построить из массива значений Datum, или из массива строк C, которые будут переданы функциям преобразования ввода для типов столбцов кортежа. В любом случае сначала нужно получить или сконструировать дескриптор TupleDesc для структуры кортежа. Работая со значениями Datum, необходимо передать TupleDesc функции BlessTupleDesc, а затем вызвать heap_form_tuple для каждой строки. Работая со строками C, необходимо передать TupleDesc функции TupleDescGetAttInMetadata, а затем для каждой строки вызвать BuildTupleFromCStrings. В случае функции, возвращающей множество кортежей, все подготовительные действия можно выполнить один раз при первом вызове функции.

Для получения требуемого дескриптора TupleDesc предлагается несколько дополнительных функций. Рекомендованный способ возврата составных значений заключается в вызове функции:

```
TypeFuncClass get_call_result_type(FunctionCallInfo fcinfo,
Oid *resultTypeId, TupleDesc *resultTupleDesc)
```

При этом в fcinfo должна передаваться та же структура, что была передана самой вызывающей функции. (Для этого, конечно, необходимо использовать соглашения о вызовах версии 1.) В resultTypeId можно передать NULL или адрес локальной переменной, в которую будет записан OID типа результата функции. В resultTupleDesc должен передаваться адрес локальной переменной TupleDesc. Убедить, что функция возвратила результат TYPEFUNC_COMPOSITE; в этом случае в resultTupleDesc оказывается требуемая структура TupleDesc. (Если получен другой результат, можно получить ошибку с сообщением «функция, возвращающая запись, вызвана в контексте, не допускающем этот тип».)

Ранее для получения TupleDesc использовались теперь уже устаревшие функции:

TupleDesc RelationNameGetTupleDesc(const char *relname)

(возвращает TupleDesc для типа строк указанного отношения) и:

TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)

(возвращает TupleDesc для типа, задаваемого по OID). Применяя её, можно получить TupleDesc для базового или составного типа. Однако она не подойдёт для функции, возвращающей тип record, и не сможет разрешить полиморфные типы.

Получив TupleDesc, необходимо вызвать:

TupleDesc BlessTupleDesc(TupleDesc tupdesc)

если планируется работать со структурами Datum, либо:

AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)

если планируется работать со строками C. Если разрабатывается функция, возвращающая набор данных, можно сохранить результаты этих функций в структуре FuncCallContext, в поле tuple_desc или attinmeta, соответственно.

При работе со структурами Datum, необходимо воспользоваться функцией:

HeapTuple heap_form_tuple(TupleDesc tupdesc, Datum *values, bool *isnull)

Она формирует HeapTuple из переданных ей данных в форме Datum. При работе со строками C, необходимо воспользоваться функцией:

HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)

Она формирует HeapTuple из переданных ей данных в виде строк C. В параметре *values* ей передаётся массив строк C, по одной для каждого атрибута выходной строки. Каждая из этих строк должна иметь формат, принимаемый функцией ввода типа данных атрибута. Чтобы задать значение NULL для одного из этих атрибутов, вместо соответствующего указателя в массиве *values* нужно передать NULL. Эту функцию нужно вызывать для каждой возвращаемой строки.

Получив кортеж, который возвращается из функции, необходимо преобразовать его в тип Datum. Чтобы преобразовать HeapTuple в Datum, необходимо воспользоваться функцией:

HeapTupleGetDatum(HeapTuple tuple)

Полученный тип Datum можно вернуть непосредственно, если должна возвращаться только одна строка, либо использовать как текущее выдаваемое значение в функции, возвращающей набор строк.

28.1.10.8. Возврат множеств

Функции на языке C могут возвращать наборы данных (множества строк) двумя способами. Первый способ, который называется *ValuePerCall* (значение за вызов), заключается в многократном вызове функции (при этом ей каждый раз передаются одни и те же аргументы). Эта функция при очередном вызове должна возвращать следующую строку, пока не выдаст все строки, о чём она сообщает, возвращая NULL. Таким образом, возвращающая множество функция (Set- Returning Function, SRF) должна сохранять между вызовами своё состояние в достаточном объёме, чтобы помнить, какие данные уже были выданы, и возвращать следующие при очередном вызове. Второй вариант, *Materialize* (Материализация), заключается в формировании в SRF объекта tuplestore, содержащего сразу весь результирующий набор; единственный вызов производится для получения сразу всего результата, и никакое состояние между вызовами сохранять не нужно.

Реализуя режим ValuePerCall, важно не забывать, что выполнение запроса до полного завершения не гарантируется. Так, например, получив указание LIMIT, исполнитель запроса может перестать вызывать возвращающую множество функцию до получения всех строк. Это означает, что выполнять действия, связанные с очисткой, в последнем вызове небезопасно, так как он может вовсе не состояться. Поэтому для функций, которым нужно обращаться к внешним ресурсам, например, открывая файловые дескрипторы, рекомендуется использовать режим материализации результатов.

В продолжении этого раздела описываются несколько вспомогательных макросов, которые часто используются (хотя не являются обязательными) в SRF, реализующих метод ValuePerCall.

Для использования описанных макросов поддержки ValuePerCall необходимо подключить funcapi.h. Эти макросы работают со структурой FuncCallContext, содержащей состояние, которое требуется сохранять между вызовами. Указатель на FuncCallContext внутри вызываемой SRF сохраняется между вызовами в поле fcinfo->flinfo->fn_extra, которое макросы автоматически заполняют при первом использовании, рассчитывая прочесть из него тот же указатель при последующих вызовах.

Для SRF предоставляется ряд макросов, использующих эту инфраструктуру:

SRF_IS_FIRSTCALL() – макрос, чтобы определить, вызывается ли функция в первый раз. При первом вызове (но не при последующих) необходимо выполнить:

SRF_FIRSTCALL_INIT()

для того, чтобы инициализировать FuncCallContext. При каждом вызове функции, включая первый, необходимо выполнить:

SRF_PERCALL_SETUP()

для того, чтобы подготовиться к использованию FuncCallContext.

Если у функции есть данные, которые она должна выдать в текущем вызове, необходимо выполнить:

SRF_RETURN_NEXT(funcctx, result)

для того, чтобы передать их вызывающему. (Переменная result должна быть типа Datum, либо одним значением, либо кортежем, подготовленным как описано выше.) Когда функция закончила выдавать данные, необходимо выполнить:

SRF_RETURN_DONE(funcctx)

для того, чтобы провести очистку и завершить SRF.

Контекст памяти, в котором вызывается SRF, временный, он будет очищаться между вызовами. Это значит, что не нужно вызывать pfree для всех блоков памяти, полученных через palloc; они всё равно будут освобождены. Однако если необходимо выделить структуры данных, сохраняющиеся между вызовами, нужно разместить их где-то в другом месте. Для размещения данных, которые не должны уничтожаться, пока SRF не закончит работу, подходит контекст памяти, на который указывает multi_call_memory_ctx. В большинстве случаев это означает, что необходимо переключиться в контекст multi_call_memory_ctx в коде подготовки при первом вызове. Для сохранения указателя на такие долгоживущие структуры необходимо воспользоваться полем funcctx->user_fctx. (Память, которую получается в контексте multi_call_memory_ctx, будет освобождена автоматически при завершении запроса, так что и её освобождать вручную нет необходимости.)

28.1.109. Полиморфные типы аргументов и результата

Функции на языке C могут быть объявлены как принимающие и возвращающие полиморфные типы. Когда типы аргументов или результата определены как полиморфные, автор функции не может заранее знать, с какими типами данных она будет вызываться и какой возвращать. Чтобы функция на C в стиле версии 1 могла определить фактические типы данных своих аргументов и тип, который она должна вернуть, в fmgr.h предлагаются две функции. Они называются get_fn_expr_rettype(FmgrInfo *flinfo) и get_fn_expr_argtype(FmgrInfo *flinfo, int argnum) и возвращают соответственно OID типа результата и аргумента, либо InvalidOid, если информация о типе отсутствует. Структуру flinfo обычно можно получить по ссылке fcinfo->flinfo. Номер аргумента argnum задаётся, начиная с нуля. В качестве альтернативы get_fn_expr_rettype также можно использовать функции get_call_result_type. Кроме того, есть функция get_fn_expr_variadic, позволяющая определить, были ли переменные аргументы объединены в массив. Это используется в основном для функций VARIADIC "any", так как такое объединение всегда имеет место для функций с переменными аргументами, принимающих обычные типы.

Существует один вариант полиморфизма, которым могут пользоваться только функции на языке C: их можно объявить с параметрами типа "any". (Имя этого типа нужно заключать в двойные кавычки, так как это также зарезервированное слово в SQL.) Он работает так же, как anyelement, за исключением того, что он не требует, чтобы аргументы "any" имели одинаковый тип, и не помогает определить тип результата функции. Функцию на языке C можно также объявить с последним параметром VARIADIC "any". Ему будут соответствовать один или более фактических аргументов любого типа (не обязательно одинакового). Эти аргументы *не* будут собираться в массив, как это происходит с обычными функциями с переменными аргументами; они просто будут переданы функции по отдельности. Если применяется этот вариант, то чтобы определить число фактических аргументов и их типы, нужно использовать макрос PG_NARGS() и функции, описанные выше. Пользователи такой функции также могут пожелать использовать ключевое слово VARIADIC в вызове функции, ожидая, что функция обработает элементы массива как отдельные аргументы. При необходимости соответствующее поведение должна реализовывать сама функция, определив с помощью get_fn_expr_variadic, был ли фактический аргумент передан с указанием VARIADIC.

28.1.10.10. Разделяемая память и лёгкие блокировки

Модули расширений могут резервировать лёгкие блокировки и область в разделяемой памяти при запуске сервера. Чтобы библиотека модуля предварительно загружалась на этапе запуска сервера, нужно указать её в shared_preload_libraries. Чтобы зарезервировать разделяемую память, необходимо вызвать из функции _PG_init функцию:

```
void RequestAddinShmemSpace(int size)
```

Чтобы зарезервировать лёгкие блокировки, из _PG_init нужно вызвать:

```
void RequestNamedLWLockTranche(const char *tranche_name, int num_lwlocks)
```

В результате будет сформирован массив из num_lwlocks лёгких блокировок под именем tranche_name. Чтобы получить указатель на этот массив, необходимо воспользоваться функцией GetNamedLWLockTranche.

28.1.10.11. Использование C++ для расширяемости

Хотя код сервера PG360 написан на C, расширения для него можно писать и на C++, если соблюдать следующие правила:

– Все функции, к которым будет обращаться сервер, должны предоставлять ему интерфейс C; эти функции на C затем могут вызывать функции на языке C++. В частности, для функций, доступных серверу, необходимо указать extern C. Это также необходимо для всех функций, указатели на которые передаются между кодом сервера и подключаемым кодом на C++.

– Освобождать память, применяя для этого подходящий метод. Например, память сервера в основном выделяется функцией palloc(), так что освобождать её нужно, вызывая pfree(). Попытка использовать в таких случаях принятую в C++ операцию delete приведёт к ошибке.

– Не допускать распространения исключений в код C (добавить блок, перехватывающий все исключения, на верхнем уровне функций extern C). Это необходимо, даже если код на C++ не генерирует исключения явно, потому что исключения могут возникать, например, и при нехватке

памяти. Все исключения должны перехватываться, и в интерфейс C должны передаваться соответствующие ошибки. Если возможно, скомпилировать код C++ с указанием -fno-exceptions, чтобы полностью отключить исключения; в таких случаях необходимо выявлять исключительные ситуации в коде C++, например, проверять на NULL адрес, возвращённый new().

– Вызывая серверные функции из кода C++, убедиться, что в стеке вызова C++ содержатся только простые структуры данных. Это необходимо, потому что в случае ошибки сервера выполняется функция longjmp(), а она не отматывает стек вызовов C++ должным образом для объектов, отличных от простых структур.

28.11.11. Пользовательские агрегатные функции

Агрегатные функции в PG360 определяются в терминах *значений состояния* и *функций перехода состояния*. То есть агрегатная функция работает со значением состояния, которое меняется при обработке каждой последующей строки. Чтобы определить агрегатную функцию, нужно выбрать тип данных для значения состояния, начальное значение состояния и функцию перехода состояния. Функция перехода состояния принимает предыдущее значение состояния и входное агрегируемое значение для текущей строки и возвращает новое значение состояния. Также можно указать *функцию завершения*, на случай, если ожидаемый результат агрегатной функции отличается от данных, которые сохраняются в изменяющемся значении состояния. Функция завершения принимает конечное значение состояния и возвращает то, что она хочет вернуть в виде результата агрегирования. В принципе, функции перехода и завершения представляют собой просто обычные функции, которые также могут применяться вне контекста агрегирования. (На практике для большей производительности часто создаются специализированные функции перехода, которые работают, только когда вызываются при агрегировании.)

Таким образом, помимо типов данных аргументов и результата, с которыми имеет дело пользователь агрегатной функции, есть также тип данных внутреннего состояния, который может отличаться от этих типов.

Если определить агрегат, не использующий функцию завершения, агрегат будет вычислять бегущее значение функции по столбцам каждой строки. Примером такой агрегатной функции является sum. Вычисление sum начинается с нуля, а затем к накапливаемой сумме всегда прибавляется значение из текущей строки. Например, при необходимости сделать агрегатную функцию sum для комплексных чисел, потребуется только функция сложения для такого типа данных. Такая агрегатная функция может быть определена следующим образом:

```
CREATE AGGREGATE sum (complex) (
sfunc = complex_add, stype = complex, initcond = '(0,0)'
);
```

Использовать её можно будет следующим образом:

```
SELECT sum(a) FROM test_complex;

sum (34,53.9)
```

Определённая выше функция sum вернёт ноль (начальное значение состояния), если в наборе данных не окажется значений, отличных от NULL. У нас может возникнуть желание вернуть NULL в этом случае — стандарт SQL требует, чтобы sum работала так. Можно добиться

этого, просто опустив фразу initcond, так что начальным значением состояния будет NULL. Обычно это будет означать, что в sfunc придётся проверять входное значение состояния на NULL. Но для sum и некоторых других простых агрегатных функций, как max и min, достаточно вставить в переменную состояния первое входное значение не NULL, а затем начать применять функцию перехода со следующего значения не NULL. PG360 сделает это автоматически, если начальное значение состояние равно NULL и функция перехода помечена как «strict» (то есть не должна вызываться для аргументов NULL).

Ещё одна особенность поведения по умолчанию «строгой» функции перехода — предыдущее значение состояния остаётся без изменений, когда встречается значение NULL. Другими словами, значения NULL игнорируются.

Функция avg (вычисляющая среднее арифметическое) представляет собой более сложный пример агрегатной функции. Ей необходимы два компонента текущего состояния: сумма входных значений и их количество. Окончательный результат получается как частное этих величин. При реализации этой функции для значения состояния обычно используется массив. Например, встроенная реализация avg(float8) выглядит следующим образом:

```
CREATE AGGREGATE avg (float8) (
sfunc = float8_accum, stype = float8[], finalfunc = float8_avg, initcond = '{0,0,0}'
);
```

Вызовы агрегатных функций SQL допускают указания DISTINCT и ORDER BY, которые определяют, какие строки и в каком порядке будут поступать в функцию перехода агрегата. Это реализовано на заднем плане и непосредственно не затрагивает функции, поддерживающие работу агрегата.

28.1.11.1. Режим движущегося агрегата

Агрегатные функции могут дополнительно поддерживать *режим движущегося агрегата*, который позволяет значительно быстрее выполнять агрегатные функции в окнах со сдвигающимся началом рамки. Основная идея состоит в том, что помимо добавления обычной функции перехода «вперёд», для агрегатной функции задаётся *функция обратного перехода*, которая позволяет убирать строки из накапливаемого значения состояния, когда они покидают рамку окна. Например, для sum в качестве функции прямого перехода выполняется сложение, а в качестве функции обратного перехода выполняется вычитание. Без функции обратного перехода механизм оконных функций вынужден вычислять агрегат заново при каждом перемещении начала рамки, в результате чего время обработки оказывается пропорциональным количеству входных строк, помноженному на средний размер рамки. С функцией обратного перехода это время пропорционально только количеству входных строк.

Функции обратного перехода передаётся текущее значение состояния и агрегируемое входное значение(я) для строки, ранее учтённой в текущем состоянии. Она должна восстановить то значение состояния, которое было бы получено, если бы эта строка не агрегировалась, но агрегировались все последующие. Иногда для этого нужно, чтобы функция обратного перехода сохраняла больше информации о состоянии, чем это требуется для простого режима агрегирования. Таким образом, для режима движущегося агрегата используется реализация,

отличная от простого режима: для него определяется отдельный тип данных, отдельная функция прямого перехода и отдельная функция завершения, при необходимости. Они могут совпадать с типом данных и аналогичными функциями обычного режима, если в дополнительном состоянии необходимости нет.

Функции прямого перехода в режиме движущегося агрегата не разрешено возвращать NULL в качестве нового значения состояния. Если функция обратного перехода возвращает NULL, это воспринимается как признак того, что она не может восстановить предыдущее состояние для полученных данных, и значит, агрегатное вычисление нужно производить заново с текущей позиции начала рамки. Это соглашение позволяет применять режим движущегося агрегата и в ситуациях, когда прокручивать назад значение состояния непрактично. Функция обратного перехода может «спасовать» в таких случаях, но включаться в работу, насколько это возможно в большинстве случаев. Например, агрегатная функция, работающая с числами с плавающей точкой, может спасовать, когда от неё потребуется убрать значение NaN (не число, not a number) из текущего значения состояния.

Разрабатывая функции, реализующие режим движущегося агрегата, важно, чтобы функция обратного перехода могла восстановить в точности требуемое значение состояния. Иначе в результатах могут проявляться различия в зависимости от того, использовался ли режим движущегося агрегата.

28.1.12. Агрегатные функции с полиморфными и переменными аргументами

Агрегатная функция может использовать полиморфные функции перехода состояния или функции завершения, так что эти функции могут применяться для реализации нескольких агрегатов. Более того, сама агрегатная функция может описываться с полиморфными типами входных данных и состояния, так что одно определение агрегатной функции может служить для использования с разными типами данных. Пример полиморфного агрегата:

```
CREATE AGGREGATE array_accum (anyelement) (
sfunc = array_append, stype = anyarray, initcond = '{}'
);
```

Здесь фактическим типом состояния для любого конкретного агрегатного вызова будет массив, элементы которого будут иметь тип входных данных. Действие данного агрегата заключается в накоплении всех входных значений в массиве этого типа. (встроенная агрегатная функция array_agg обеспечивает подобную функциональность, но работает быстрее, чем могла бы функция с приведённым определением.)

Так будут выглядеть результаты с аргументами двух различных типов:

```
SELECT attrelid::regclass, array_accum(attname) FROM pg_attribute
WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass GROUP
BY attrelid;
attrelid  |    array_accum
--------------+---------------------------------------
pg_tablespace | {spcname,spcowner,spcacl,spcoptions} (1 row)
SELECT attrelid::regclass, array_accum(atttypid::regtype) FROM
pg_attribute
```

```
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass GROUP
BY attrelid;
    attrelid  |    array_accum
    ----------+---------------------------
    pg_tablespace | {name,oid,aclitem[],text[]} (1 row)
```

Обычно агрегатная функция с полиморфным типом результата имеет и полиморфный тип состояния. Это необходимо, так как иначе нельзя будет объявить функцию завершения: она должна будет иметь полиморфный тип результата, но не будет иметь полиморфного аргумента, что команда CREATE FUNCTION не примет на основании того, что тип результата нельзя будет определить при вызове. Но иметь полиморфный тип состояния не всегда удобно. Чаще всего эта проблема возникает, когда функции реализации агрегата пишутся на C и тип состояния должен объявляться как internal, так как для него нет соответствующего типа на уровне SQL. Чтобы решить эту проблему, можно объявить функцию завершения как принимающую дополнительные фиктивные аргументы, соответствующие входным аргументам агрегата. В этих фиктивных аргументах всегда передаются значения NULL, так как при вызове функции завершения какое- либо определённое значение отсутствует. Единственное их предназначение — позволить связать тип результата полиморфной функции завершения с типом входных данных агрегата. Например, определение встроенного агрегата array_agg выглядит следующим образом:

```
    CREATE FUNCTION array_agg_transfn(internal, anynonarray) RETURNS
internal ...;
    CREATE FUNCTION array_agg_finalfn(internal, anynonarray) RETURNS
anyarray ...;
    CREATE AGGREGATE array_agg (anynonarray) (
    sfunc = array_agg_transfn,
    stype = internal,
    finalfunc = array_agg_finalfn, finalfunc_extra
    );
```

Здесь параметр finalfunc_extra указывает, что функция завершения помимо значения состояния получит дополнительные фиктивные аргументы, соответствующие входным аргументам агрегата. Дополнительный аргумент anynonarray позволяет сделать объявление array_agg_finalfn допустимым.

Агрегатную функцию можно сделать принимающей переменное число аргументов, объявив её последний аргумент как массив VARIADIC, в том же ключе, как и обычную функцию. При этом у функций перехода агрегата их последний аргумент должен иметь тот же тип массива. Такие функции обычно также объявляются как VARIADIC, но строго это не требуется.

28.1.11.3. Сортирующие агрегатные функции

Описанные выше агрегатные функции были «обычными» агрегатами. Но PG360 также поддерживает *сортирующие агрегатные функции*, которые имеют два отличия от обычных. Во–первых, в дополнение к обычным агрегируемым аргументам, вычисляемых для каждой входной строки, сортирующий агрегат может иметь «непосредственные» аргументы, которые должны вычисляться в операции агрегирования только один раз. Во-вторых, для обычных агрегируемых аргументов порядок их сортировки задаётся явно, а сортирующий агрегат обычно

выполняет вычисления, зависящие от конкретного порядка строк, например, вычисляет ранг или процентиль, так что порядок сортировки критичен для каждого вызова. Например, встроенное определение функции percentile_disc равнозначно следующему:

```
CREATE FUNCTION ordered_set_transition(internal, anyelement) RETURNS internal ...;
CREATE FUNCTION percentile_disc_final(internal, float8, anyelement) RETURNS anyelement ...;
CREATE AGGREGATE percentile_disc (float8 ORDER BY anyelement) (
sfunc = ordered_set_transition, stype = internal,
finalfunc = percentile_disc_final, finalfunc_extra
);
```

Этот агрегат принимает непосредственный аргумент float8 (дробь процентиля) и агрегируемые данные, которые могут быть любого упорядочиваемого типа. Используя его, можно рассчитать средний семейный доход следующим образом:

```
SELECT percentile_disc(0.5) WITHIN GROUP (ORDER BY income) FROM households; percentile_disc
50489
```

В данном случае 0.5 — это непосредственный аргумент; если бы дробь процентиля менялась от строки к строке, это не имело бы смысла.

В отличие от случая с обычными агрегатами, сортировка входных строк для сортирующего агрегата *не* выполняется на заднем плане, а является задачей функций, реализующих агрегат. Типичная реализация такого агрегата заключается в сохранении ссылки на объект «tuplesort» в значении состояния агрегата, загрузке поступающих строк в этот объект, и собственно окончании сортировки и выдачи данных в функции завершения. При такой организации обработки функция завершения может выполнять специальные операции, в частности, вставлять дополнительные «гипотетические» строки в сортируемые данные. Тогда как обычные агрегаты часто реализуются функциями, написанными на PL/pgSQL или другом процедурном языке, сортирующие агрегатные функции, как правило, должны быть написаны на C, так как их значение состояния нельзя выразить каким-либо типом данных SQL. И вследствие того, что сортировку выполняет функция завершения, нельзя возобновить добавление входных строк, продолжая вызывать функцию перехода. Это означает, что функция завершения не может иметь характеристику READ_ONLY; она должна объявляться командой CREATE AGGREGATE с характеристикой READ_WRITE или SHAREABLE (если она позволяет при последующих вызовах функции завершения использовать уже отсортированное состояние).

Функция перехода состояния для сортирующего агрегата получает значение текущего состояния плюс агрегируемые входные данные для каждой строки и возвращает изменённое значение состояния. Это определение распространяется и на обычные агрегаты, но непосредственные аргументы (если они есть) при этом не передаются. Функция завершения же получает последнее значение состояния и значения непосредственных аргументов (если они есть), а также (если присутствует указание finalfunc_extra) значения NULL, соответствующие агрегируемым данным. С обычными агрегатами указание finalfunc_extra действительно полезно, только если агрегат полиморфный; тогда дополнительные фиктивные аргументы необходимы, чтобы связать тип результата функции завершения с типом(ами) входных данных агрегата.

В настоящее время сортирующие агрегаты не могут использоваться в качестве оконных функций, поэтому от них поддержка режима движущегося агрегата не требуется.

28.11.14. Частичное агрегирование

Дополнительно агрегатная функция может поддерживать *частичное агрегирование*. Идея такого агрегирования в том, чтобы вызывать функции перехода состояния для различных подмножеств входных данных независимо, а затем комбинировать значения состояния, вычисленные по этим подмножествам, и получать тот же результат, что был бы получен при сканировании сразу всех входных данных. Этот режим может применяться для параллельного агрегирования, когда разные рабочие процессы сканируют различные части таблицы. При этом каждый рабочий процесс выдаёт частичное значение состояния, а в конце эти значения комбинируются вместе и получается окончательное значение состояния. (В будущем этот режим может также применяться, например для комбинированного агрегирования локальных и удалённых таблиц, но пока это не реализовано.)

Для поддержки частичного агрегирования в определении агрегатной функции должна задаваться *комбинирующая функция*, принимающая два значения типа состояния агрегата (представляющие результаты агрегирования по двум подмножествам входных строк) и выдающая новое значение типа состояния, представляющее то состояние, которое было бы получено при агрегировании совокупности этих подмножеств строк. При этом относительный порядок входных строк в этих двух множествах не оговаривается. Это значит, что для агрегатных функций, зависящих от порядка входных строк, обычно невозможно определить осмысленную комбинирующую функцию.

В качестве простого примера, частичное агрегирование могут поддерживать функции MAX и MIN, если задать в качестве комбинирующей соответственно функцию сравнения значений большее– из-двух или меньшее-из-двух, ту же, что они используют и как функцию перехода. Для SUM комбинирующей функцией будет просто функция сложения. (И это опять же функция перехода, если только значение состояния не выходит за рамки типа входных данных.)

Комбинирующая функция задействуется практически так же, как функция перехода, но принимает в качестве второго аргумента значение типа состояния, а не нижележащего входного типа. В частности, на неё распространяются те же правила строгости функции и передачи значений NULL. Если в определении агрегатной функции задаётся отличное от NULL значение initcond, оно будет задавать начальное состояние не только для каждого прохода частичного агрегирования, но также и начальное состояние для комбинирующей функции, которая будет вызываться для комбинирования каждого частичного результата в этом состоянии.

Если типом состояния агрегатной функции выбран internal, комбинирующая функция отвечает за то, чтобы её результат был помещён в контекст памяти, подходящий для значений агрегатного состояния. В частности это значит, что, получив в первом аргументе NULL, нельзя просто возвратить второй аргумент, так как это значение окажется в неверном контексте и не просуществует достаточное время.

Когда типом состояния агрегатной функции выбран internal, обычно в определении агрегатной функции также уместно задать *функцию сериализации* и *функцию десериализации*, которые позволяют копировать значение состояния из одного процесса в другой. Без этих

функций параллельное агрегирование невозможно, а также вероятно не будут работать такие будущие приложения, как локальное/удалённое агрегирование.

Функция сериализации должна принимать один аргумент типа internal и возвращать результат типа bytea, представляющий значение состояния, упакованное в плоский набор байтов. Функция десериализации, напротив, обращает это преобразование. Она должна принимать два аргумента типов bytea и internal и возвращать результат типа internal. (Второй её аргумент не используется и всегда равен нулю, но он требуется из соображений типобезопасности.) Результат функции десериализации следует просто разместить в текущем контексте памяти, так как в отличие от результата комбинирующей функции он недолговечен.

Также стоит заметить, что для выполнения агрегатной функции в параллельном режиме она должна иметь характеристику PARALLEL SAFE (безопасная для распараллеливания). Характеристики допустимости распараллеливания её опорных функций значения не имеют.

28.11.15. Вспомогательные функции для агрегатов

Функция, написанная на C, может определить, была ли она вызвана как вспомогательная функция агрегирования, вызвав AggCheckCallContext, например:

```
if (AggCheckCallContext(fcinfo, NULL))
```

Смысл такой проверки в том, что в случае положительного её результата первым входным аргументом является временное значение состояния, которое можно безопасно модифицировать на месте, не создавая новую копию. (Хотя агрегатные функции перехода всегда могут изменить непосредственно переходное значение, агрегатные функции завершения должны избегать этого; если они это делают, такое поведение должно отмечаться при создании агрегата. За дополнительными подробностями необходимо обратиться к функции CREATE AGGREGATE.)

Второй аргумент AggCheckCallContext можно использовать, чтобы получить контекст памяти, в котором содержатся значения агрегатного состояния. Это полезно для функций перехода, которые желают использовать «развёрнутые» объёкты в качестве значений состояния. При первом вызове такая функция перехода должна возвратить развёрнутый объект в контексте памяти, относящемся к контексту состояния агрегата, а затем продолжать возвращать тот же объект при последующих вызовах. Например, эту логику можно увидеть в функции array_append(). (Функция array_append() не используется в качестве перехода никаким встроенным агрегатом, но она написана так, чтобы работать эффективно в таком качестве в дополнительном агрегате.)

Ещё одна вспомогательная подпрограмма, предназначенная для агрегатных функций, написанных на C, называется AggGetAggref. Эта функция возвращает узел разбора Aggref, описывающий вызов агрегата. Это используется для сортирующих агрегатов, которые могут исследовать структуру узла Aggref и выяснить, какой порядок сортировки они должны реализовать.

28.11.2. Пользовательские типы

PG360 может расширяться и поддерживать новые типы данных. В этом разделе описывается, как определить новые базовые типы, то есть типы данных, описанные ниже уровня языка SQL. Для создания нового базового типа необходимо реализовать функции, работающие с этим типом, на языке низкого уровня, обычно C.

Пользовательский тип должен всегда иметь функции ввода и вывода. Эти функции определяют, как тип будет выглядеть в строковом виде (при вводе и выводе для пользователя) и как этот тип размещается в памяти. Функция ввода принимает в качестве аргумента строку символов, заканчивающуюся нулём, и возвращает внутреннее представление типа (в памяти). Функция вывода принимает в качестве аргумента внутреннее представление типа и возвращает строку символов, заканчивающуюся нулём.

Дополнительно пользовательский тип может предоставлять функции для ввода и вывода в двоичном виде. Двоичный ввод/вывод обычно работает быстрее, но хуже портируется, чем текстовый. Как и с текстовым представлением, выбор, каким будет двоичное представление, остаётся за вами. Многие встроенные типы данных стараются обеспечить двоичное представление, независимое от машинной архитектуры.

Написав функции ввода/вывода и скомпилировав их в разделяемую библиотеку, можно определить тип complex в SQL. Сначала необходимо объявить его как тип-пустышку:

```
CREATE TYPE complex;
```

Это позволит ссылаться на этот тип, определяя для него функции ввода/вывода. Теперь определить функции ввода/вывода:

```
CREATE FUNCTION complex_in(cstring) RETURNS complex
AS 'имя_файла'
LANGUAGE C IMMUTABLE STRICT;
CREATE FUNCTION complex_out(complex) RETURNS cstring
AS 'имя_файла'
LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal) RETURNS complex
AS 'имя_файла'
LANGUAGE C IMMUTABLE STRICT;
CREATE FUNCTION complex_send(complex) RETURNS bytea
AS 'имя_файла'
LANGUAGE C IMMUTABLE STRICT;
```

Затем предоставить полное определение типа данных:

```
CREATE TYPE complex ( internallength = 16, input = complex_in, output = complex_out, receive = complex_recv, send = complex_send, alignment = double
);
```

Когда определяется новый базовый тип, PG360 автоматически обеспечивает поддержку массивов с элементами такого типа. Тип массива обычно получает имя по имени базового типа с добавленным спереди символом подчёркивания (_).

Когда тип данных определён, можно объявить дополнительные функции для выполнения операций с этим типом. Затем поверх этих функций могут быть определены операторы, а если потребуется, и классы операторов, для поддержки индексации этого типа. Эти дополнительные уровни обсуждаются в следующих разделах.

Если внутреннее представление типа данных имеет переменную длину, оно должно соответствовать стандартной схеме данных переменной длины: первые четыре байта должно

занимать поле char[4], к которому никогда не следует обращаться напрямую (по обыкновению названное vl_len_). Чтобы сохранить в этом поле размер элемента (включая длину самого поля), необходимо использовать макрос SET_VARSIZE(), а чтобы получить его — макрос VARSIZE(). (Эти макросы нужны, потому что поле длины может кодироваться по-разному на разных платформах.)

28.1.12.1. Особенности TOAST

Если значения типа данных могут быть разного размера (во внутренней форме), обычно для такого типа желательно реализовать поддержку TOAST. Это следует делать, даже если значения слишком малы для сжатия или внешнего хранения, так как TOAST позволяет сэкономить пространство и с данными маленького размера, сокращая издержки в заголовке.

Для поддержки хранения TOAST функции на C, работающие с таким типом данных, должны позаботиться о распаковке поступивших им данных, используя макрос PG_DETOAST_DATUM. (Эту внутреннюю особенность обычно скрывает дополнительный, определяемый для типа макрос GETARG_DATATYPE_P.) Затем, выполняя команду CREATE TYPE, необходимо указать в качестве внутренней длины variable и выбрать подходящий вариант хранения (не plain).

Если выравнивание данных не имеет значения (либо только для некоторой функции, либо потому что для типа данных в любом случае применяется выравнивание по байтам), некоторых издержек, связанных с макросом PG_DETOAST_DATUM, можно избежать. Вместо него можно использовать PG_DETOAST_DATUM_PACKED (его обычно скрывает определяемый для типа макрос GETARG_DATATYPE_PP) и воспользоваться макросами VARSIZE_ANY_EXHDR и VARDATA_ANY для обращения к потенциально сжатым данным. Стоит ещё раз отметить, что данные, возвращаемые этими макросами, не выравниваются, даже если выравнивание задано в определении типа. Если выравнивание важно, необходимо задействовать обычный интерфейс PG_DETOAST_DATUM.

Поддержка TOAST даёт также возможность иметь *развёрнутое* представление данных в памяти, работать с которым будет удобнее, чем с форматом хранения на диске. Обычный или «плоский» формат хранения varlena в конце концов представляет собой просто набор байт; он не может содержать указатели, так как эти байты могут быть скопированы в другие адреса. Для сложных типов данных работать с плоским форматом данных может быть довольно дорого, так что PG360 даёт возможность «развернуть» плоский формат в представление, более подходящее для вычислений, и затем передавать эту структуру в памяти функциям, работающим с этим типом.

Для использования развёрнутого хранения тип данных должен определять развёрнутый формат по правилам, описанным в src/include/utils/expandeddatum.h, и предоставлять функции для «разворачивания» плоского значения в этот формат, а также для «заворачивания» этого формата опять в обычное представление varlena. Затем надо добиться, чтобы все функции на C могли принимать любое представление, возможно выполняя преобразование одного в другое непосредственно при получении. Для этого не требуется исправлять сразу все существующие функции для этого типа данных, так как имеющийся стандартный макрос PG_DETOAST_DATUM способен преобразовывать развёрнутые входные данные в обычный плоский формат. Таким

образом, все существующие функции, работающие с плоским форматом varlena продолжат работать, хотя и не очень эффективно, с развёрнутыми входными данными; их необязательно переделывать, пока не потребуется оптимизировать производительность.

Функции на C, умеющие работать с развёрнутым представлением, обычно делятся на две категории: те, что могут работать с развёрнутым форматом, и те, что могут принимать и развёрнутые, и плоские данные varlena. Первые проще написать, но они могут быть менее эффективными в целом, так как преобразование плоского значения в развёрнутую форму для использования только одной функцией может стоить больше, чем сэкономится при обработке данных в развёрнутом формате. Когда нужно работать только с развёрнутым форматом, преобразование плоских значений в развёрнутую форму можно скрыть в макросе, извлекающем аргументы, чтобы функция была не сложнее, чем работающая с традиционными входными данными varlena. Чтобы принимать оба варианта входных значений, необходимо написать функцию извлечения аргументов, которая будет распаковывать значения с сокращённым заголовком, а также внешние и сжатые, но не развёрнутые данные. Такую функцию можно определить как возвращающую указатель на объединение плоского формата varlena и развёрнутого формата. Какой формат получен фактически, вызывающий код может определить, вызвав макрос VARATT_IS_EXPANDED_HEADER().

Инфраструктура TOAST позволяет не только отличить обычные значения varlena от развёрнутых значений, но и различить указатели «для чтения/записи» и «только для чтения» на развёрнутые значения. Функции на C, которым нужно читать развёрнутое значение, или которые будут менять его безопасным и невидимым извне образом, могут не обращать внимания на тип полученного указателя. Если же функции на C выдают изменённую версию входного значения, они могут изменять развёрнутые входные данные на месте, только когда получают указатель для чтения/записи, но не когда получен указатель только для чтения. В последнем случае они должны сначала скопировать значение и получить новое значение, допускающее изменение. Функция на C, создающая новое развёрнутое значение, должна всегда возвращать указатель на него для чтения/записи. Кроме того, функция, изменяющая развёрнутое значение непосредственно по указателю для чтения/записи должна позаботиться о том, чтобы это значение осталось в приемлемом состоянии, если она отработает не полностью.

28.11.3. Пользовательские операторы

Прежде чем создать оператор, необходимо создать нижележащую функцию. Оператор несёт и дополнительную информацию, помогающую планировщику запросов оптимизировать запросы с этим оператором.

PG360 поддерживает левые унарные, правые унарные и бинарные операторы. Операторы могут быть перегружены; то есть одно имя оператора могут иметь различные операторы с разным количеством и типами операндов. Когда выполняется запрос, система определяет, какой оператор вызвать, по количеству и типам предоставленных операндов.

COMMUTATOR

Предложение COMMUTATOR, если представлено, задаёт оператор, коммутирующий для определяемого. Оператор A является коммутирующим для оператора B, если (x A y) равняется (y

B x) для всех возможных значений x, y. B также будет коммутирующим для A. Например, операторы < и > для конкретного типа данных обычно являются коммутирующими друг для друга, а оператор + — коммутирующий для себя. Но традиционный оператор – коммутирующего не имеет.

Тип левого операнда оператора должен совпадать с типом правого операнда коммутирующего для него оператора, и наоборот.

NEGATOR

Предложение NEGATOR, если присутствует, задаёт оператор, обратный к определяемому. Оператор A является обратным к оператору B, если они оба возвращают логический результат и (x A y) равняется NOT (x B y) для всех возможных x, y. B так же является обратным к A. Например, операторы < и >= составляют пару обратных друг к другу для большинства типов данных. Никакой оператор не может быть обратным к себе же.

В отличие от коммутирующих операторов, два унарных оператора вполне могут быть обратными к друг другу; это будет означать, что (A x) равняется NOT (B x) для всех x (и для правых унарных операторов аналогично).

У оператора, обратного данному, типы левого и/или правого операнда должны соответствовать типам данного оператора, так же как и с предложением COMMUTATOR; отличие только в том, что имя оператора задаётся в предложении NEGATOR.

Указание обратного оператора может использоваться для оптимизатора запросов, так как это позволяет упростить выражение вида NOT (x = y) до x <> y. Такие выражения не так редки, как может показаться, так как операции NOT могут добавляться автоматически в результате реорганизаций выражений.

Пару обратных операторов можно определить теми же способами, что были описаны ранее для пары коммутирующих.

RESTRICT

Предложение RESTRICT, если представлено, определяет функцию оценки избирательности ограничения для оператора. Предложения RESTRICT имеют смысл только для бинарных операторов, возвращающих boolean. Идея оценки избирательности ограничения заключается в том, чтобы определить, какой процент строк в таблице будет удовлетворять условию WHERE вида:

```
column OP constant
```

для текущего оператора и определённого значения константы. Это помогает оптимизатору примерно определить, сколько строк будет исключено предложениями WHERE такого вида.

Можно использовать один из стандартных системных оценщиков для большинства дополнительных операторов. Стандартные оценщики ограничений следующие:

```
eqsel для = neqsel для <> scalarltsel для <
scalarlesel для <= scalargtsel для > scalargesel для >=
```

Часто можно обойтись функциями eqsel и neqsel для операторов с очень высокой или низкой избирательностью, даже если это не операторы собственно равенства или неравенства. Например, геометрические операторы приблизительного равенства используют eqsel в

предположении, что соответствующие (равные) элементы будут составлять только небольшой процент от всех записей таблицы.

Функции scalarltsel, scalarlesel, scalargtsel и scalargesel можно использовать для сравнений с типами данных, которые могут быть каким-либо осмысленным образом преобразованы в числовые скалярные значения для сравнения диапазонов. Если возможно, необходимо добавить свой тип данных в число типов, которые понимает функция convert_to_scalar(), реализованная в src/backend/utils/adt/selfuncs.c.

Есть ещё одна функция оценки избирательности, matchingsel, которая будет работать практически с любым бинарным оператором, если для его входных типов данных собирается статистика MCV и/или строится гистограмма. По умолчанию эта оценка в два раза больше той, что выдаёт eqsel, таким образом, данная функция наиболее полезна для операторов сравнения, более избирательных, чем оператор равенства. (Также можно вызвать нижележащую функцию generic_restriction_selectivity, передав ей другую оценку по умолчанию.)

JOIN

Предложение JOIN, если представлено, определяет функцию оценки избирательности соединения для оператора. Предложения JOIN имеют смысл только для бинарных операторов, возвращающих boolean. Идея оценки избирательности соединения заключается в том, чтобы угадать, какой процент строк в паре таблиц будет удовлетворять условию WHERE следующего вида:

```
table1.column1 OP table2.column2
```

для текущего оператора. Как и RESTRICT, это предложение очень помогает оптимизатору, позволяя ему выяснить, какой из возможных вариантов соединения скорее всего окажется выгоднее.

Для функции оценивания избирательности соединения можно использовать один из подходящих стандартных оценщиков:

```
eqjoinsel для =
neqjoinsel для <>
scalarltjoinsel для <
scalarlejoinsel для <=
scalargtjoinsel для >
scalargejoinsel для >=
matchingjoinsel для типовых операторов сопоставления
areajoinsel для сравнений областей в плоскости
positionjoinsel для сравнения положений в плоскости
contjoinsel для проверки на включение в плоскости
```

HASHES

Предложение HASHES, если присутствует, говорит системе, что для соединений с применением этого оператора допустимо использовать метод соединения по хешу. HASHES имеет смысл только для бинарного оператора, который возвращает boolean, и на практике этот оператор должен выражать равенство значений некоторого типа данных или пары типов данных.

Соединение по хешу базируется на том предположении, что оператор соединения возвращает истину только для таких пар значений слева и справа, для которых получается

одинаковый хеш. Если два значения оказываются в разных ячейках хеша, операция соединения никогда не будет сравнивать их, неявно подразумевая, что результат оператора соединения в этом случае должен быть ложным. Поэтому не имеет никакого смысла указывать HASHES для операторов, которые не представляют какую-любо форму равенства. В большинстве случаев практический смысл в поддержке хеширования есть только для операторов, принимающих один тип данных с обеих сторон. Однако иногда возможно разработать хеш-функции, совместимые сразу с несколькими типами данных; то есть, функции, которые будут выдавать одинаковые хеш- коды для «равных» значений, несмотря на то, что эти значения будут представлены по-разному. Например, довольно легко функции с такой особенностью реализуются для хеширования целых чисел различного размера.

Чтобы оператор соединения имел характеристику HASHES, он должен входить в семейство операторов индексирования по хешу. Это требование откладывается, когда оператор только создаётся, ведь нужное семейство операторов, разумеется, ещё не может существовать. Но при попытке использовать такой оператор для соединения по хешу, возникнет ошибка во время выполнения, если такого семейства не окажется. Системе необходимо знать семейство операторов, чтобы найти функции для хеширования типа(ов) входных данных оператора. Необходимо также определить подходящие функции хеширования, прежде чем сможете создать семейство операторов.

При подготовке функции хеширования обязательно необходимо позаботиться о том, чтобы она всегда выдавала нужный результат, вне зависимости от особенностей машинной архитектуры. Например, если тип данных представлен в структуре, в которой есть незначащие дополняющие биты, нельзя просто передать всю структуру функции hash_any. (Это возможно, только если все операторы и функции гарантированно очищают незначащие биты, что является рекомендуемой стратегией.) В качестве другого примера можно привести типы с плавающей точкой в стандарте IEEE, в которых отрицательный ноль и положительный ноль — различные значения (отличаются на уровне битов), но при сравнении они считаются равными. Если значение с плавающей точкой может содержать отрицательный ноль, требуются дополнительные действия, чтобы для него выдавался тот же хеш, что и для положительного нуля.

Оператор соединения по хешу должен иметь коммутирующий (это может быть тот же оператор, если у него два операнда одного типа, либо связанный оператор равенства, в противном случае), относящийся к тому же семейству операторов. В случае его отсутствия, при попытке использования оператора возможны ошибки планировщика. Также желательно (хотя это строго не требуется), чтобы в семействе операторов хеширования, поддерживающем несколько типов данных, определялись операторы равенства для всех комбинаций этих типов данных; это способствует лучшей оптимизации.

MERGES

Предложение MERGES, если присутствует, говорит системе, что для соединений с применением этого оператора допустимо использовать метод соединения слиянием. MERGES имеет смысл только для бинарного оператора, который возвращает boolean, и на практике этот оператор должен выражать равенство значений некоторого типа данных или пары типов данных.

Идея объединения слиянием заключается в упорядочивании таблиц слева и справа и затем параллельном сканировании их. Поэтому оба типа данных должны поддерживать сортировку в полном объёме, а оператор соединения должен давать положительный результат только для пар значений, оказавшихся в «одном месте» при определённом порядке сортировки. На практике это означает, что оператор соединения должен работать как проверка на равенство. Но при этом возможно объединить слиянием два различных типа данных, если они совместимы логически. Например, оператор проверки равенства smallint и integer может применяться для соединений слиянием; понадобятся только операторы сортировки, приводящие оба типа данных в логически совместимые последовательности.

Чтобы оператор соединения имел характеристику MERGES, он должен являться членом семейства операторов индекса btree, реализующим равенство. Это требование откладывается, когда оператор только создаётся, ведь нужное семейство операторов, разумеется, ещё не может существовать. Но этот оператор не будет фактически применяться для соединений слиянием, пока не будет найдено соответствующее семейство операторов. Таким образом, флаг MERGES только подсказывает планировщику, что стоит обратиться к соответствующему семейству.

Оператор соединения слиянием должен иметь коммутирующий (это может быть тот же оператор, если у него два операнда одного типа, либо связанный оператор равенства, в противном случае), относящийся к тому же семейству операторов. В случае его отсутствия, при попытке использования оператора возможны ошибки планировщика. Также желательно (хотя это строго не требуется), чтобы в семействе операторов btree, поддерживающем несколько типов данных, определялись операторы равенства для всех комбинаций этих типов данных; это способствует лучшей оптимизации.

28.1.14 Интерфейсы расширений для индексов

Описанные до этого процедуры позволяли определять новые типы, функции и операторы. Однако можно определить индекс по столбцу нового типа данных. Для этого потребуется создать *класс операторов* для нового типа данных.

Классы операторов могут объединяться в *семейства операторов*, выражающие зависимости между семантически совместимыми классами. Когда вводится один тип данных, достаточно класса операторов.

28.1.14.1. Методы индексов и классы операторов

В системном каталоге есть таблица pg_am, содержащая записи для каждого метода индекса (внутри называемого методом доступа). Поддержка обычного доступа к таблицам встроена в PG360, но все методы доступа описываются в pg_am. Система позволяет добавлять новые методы доступа — для этого нужно написать необходимый код, а затем добавить запись в pg_am.

Процедуры метода индекса непосредственно ничего не знают о типах данных, с которыми будет применяться этот метод. Вместо этого, набор операций, которые нужны методу индекса для работы с конкретным типом данных, определяется *классом операторов*. Классы операторов называются так потому, что они определяют множество операторов в предложении WHERE, которые могут использоваться с индексом (т. е. могут быть сведены к сканированию индекса). В классе операторов могут также определяться некоторые *опорные функции*, необходимые для

внутренних операций метода индекса, но они не соответствуют напрямую каким-либо операторам предложения WHERE, которые могут обрабатываться с индексом.

Для одного типа данных и метода индекса можно определить несколько классов операторов. Благодаря этому, для одного типа данных можно использовать несколько семантически разных вариантов индексирования. Например, индекс-B-дерево требует, чтобы для каждого типа данных, с которым он работает, определялся порядок сортировки. Для типа комплексных чисел может быть полезен класс операторов B-дерева, сортирующий данные по модулю комплексного числа, и ещё один, сортирующий по вещественной части, и т. п. Обычно предполагается, что один из классов операторов будет применяться чаще других, и тогда он помечается как класс по умолчанию для данного типа и метода индекса.

Одно и то же имя класса операторов может использоваться для разных методов индекса (например, для методов индекса-B-дерева или хеш-индекса применяются классы операторов int4_ops), но все такие классы являются независимыми и должны определяться отдельно.

28.1.142. Стратегии методов индексов

Операторам, которые связываются с классом операторов, назначаются «номера стратегий», определяющие роль каждого оператора в контексте его класса. Например, в B-дереве должен быть строгий порядок ключей с отношениями меньше/больше, так что в данном контексте представляют интерес операторы «меньше» и «больше или равно». Так как PG360 позволяет пользователям определять операторы произвольным образом, PG360 не может просто посмотреть на имя оператора (< или >=) и сказать, какое сравнение он выполняет. Вместо этого для метода индекса определяется набор «стратегий», которые можно считать обобщёнными операторами. Каждый класс операторов устанавливает, какие фактические операторы соответствуют стратегиям для определённого типа данных и интерпретации семантики индекса.

Для метода индекса-B-дерева определены пять стратегий, описанных в Таблице 5.

Таблица 5. Стратегии B-дерева

Операция	**Номер стратегии**
меньше	1
меньше или равно	2
равно	3
больше или равно	4
больше	5

Индексы по хешу поддерживают только сравнение на равенство, так что они используют только одну стратегию, показанную в Таблице 6.

Таблица 6. Стратегии хеша

Операция	**Номер стратегии**
равно	1

Индексы GiST более гибкие: для них вообще нет фиксированного набора стратегий. Вместо этого опорная процедура «согласованности» каждого конкретного класса операторов GiST интерпретирует номера стратегий как ей угодно. Например, некоторые из встроенных классов операторов для индексов GiST индексируют двумерные геометрические объекты, и реализуют

стратегии «R-дерева», показанные в Таблице 7. Четыре из них являются истинно двумерными проверками (overlaps, same, contains, contained by); другие четыре учитывают только ординаты, а ещё четыре проводят же проверки только с абсциссами.

Таблица 7. Стратегии двумерного «R-дерева» индекса GiST

Операция	**Номер стратегии**
строго слева от	1
не простирается правее	2
пересекается с	3
не простирается левее	4
строго справа от	5
одинаковы	6
содержит	7
содержится в	8
не простирается выше	9
строго ниже	10
строго выше	11
не простирается ниже	12

Индексы SP-GiST такие же гибкие, как и индексы GiST: для них не задаётся фиксированный набор стратегий. Вместо этого опорные процедуры каждого класса операторов интерпретируют номера стратегий в соответствии с определением класса операторов. В качестве примера, в Таблице 8 приведены номера стратегий, установленные для встроенных классов операторов для точек.

Таблица 8. Стратегии SP-GiST для точек

Операция	**Номер стратегии**
строго слева от	1
строго справа от	5
одинаковы	6
содержится в	8
строго ниже	10
строго выше	11

Индексы GIN такие же гибкие, как и индексы GiST и SP-GiST: для них не задаётся фиксированный набор стратегий. Вместо этого опорные процедуры каждого класса операторов интерпретируют номера стратегий в соответствии с определением класса операторов. В качестве примера, в Таблице 9 приведены номера стратегий, установленные для встроенного класса операторов для массивов.

Таблица 9. Стратегии GIN для массивов

Операция	**Номер стратегии**
пересекается с	**1**
содержит	**2**
содержится в	**3**
равно	**4**

Индексы BRIN такие же гибкие, как и индексы GiST, SP-GiST и GIN: для них не задаётся фиксированный набор стратегий. Вместо этого опорные процедуры каждого класса операторов интерпретируют номера стратегий в соответствии с определением класса операторов. В качестве примера, в Таблице 10 приведены номера стратегий, используемые встроенными классами операторов Minmax.

Таблица 10. Стратегии BRIN Minmax

Операция	**Номер стратегии**
меньше	1
меньше или равно	2
равно	3
больше или равно	4
больше	5

Все вышеперечисленные операторы возвращают булевы значения. На практике все операторы, определённые как операторы поиска для метода индекса, должны возвращать тип boolean, так как они должны находиться на верхнем уровне предложения WHERE, чтобы для них применялся индекс. (Некоторые методы доступа по индексу также поддерживают *операторы упорядочивания*, которые обычно не возвращают булевы значения)

28.1.14.3. Опорные процедуры метода индекса

Стратегии обычно не дают системе достаточно информации, чтобы понять, как использовать индекс. На практике, чтобы методы индекса работали, необходимы дополнительные опорные процедуры. Например, метод индекса-B-дерева должен уметь сравнивать два ключа и определять, больше, равен или меньше ли первый второго. Аналогично, метод индекса по хешу должен уметь сравнивать хеш-коды значений ключа. Эти операции не соответствуют операторам, которые применяются в условиях в командах SQL; это внутрисистемные подпрограммы, используемые методами индекса.

Так же, как и со стратегиями, класс операторов определяет, какие конкретные функции должны играть каждую из ролей для определённого типа данных и интерпретации семантики индекса. Для метода индекса определяется набор нужных ему функций, а класс оператора выбирает нужные функции для применения, назначая им «номера опорных функций», определяемые методом индекса.

Некоторые классы операторов дополнительно позволяют задать параметры, управляющие их поведением. У всех встроенных индексных методов доступа имеется необязательная опорная функция options, которая определяет набор параметров, поддерживаемых данным классом.

Для B-деревьев требуется опорная функция сравнения и могут предоставляться четыре дополнительные опорные функции по выбору разработчика класса операторов, описанные в

Таблице11.

Таблица 11. Опорные функции B-деревьев

Функция	**Номер опорной функции**
Сравнивает два ключа и возвращает целое меньше нуля, ноль или целое больше нуля, показывающее, что первый ключ меньше, равен или больше второго	**1**
Возвращает адреса вызываемых из C опорных функций (или функции) сортировки (необязательная)	**2**
Сравнивает проверяемое значение с базовым плюс/минус смещение и возвращает true или false в зависимости от результата сравнения (необязательная)	**3**
Определяет, может ли в индексах, использующих данный класс операторов, безопасно применяться реализованное в btree исключение дубликатов (необязательная)	**4**
Определяет набор параметров, относящихся к данному классу операторов (необязательная)	**5**

Для хеш-индексов требуется одна опорная функция, и ещё две могут задаваться по выбору разработчика класса операторов, как показано в Таблице 12.

Таблица 12. Опорные функции хеша

Функция	**Номер опорной функции**
Вычисляет 32-битное значение хеша для ключа	**1**
Вычисляет 64-битное значение хеша для ключа с заданной 64-битной солью; если значение соли равно 0, младшие 32 бита результата должны соответствовать значению, которое было бы вычислено функцией 1 (необязательная)	**2**
Определяет набор параметров, относящихся к данному классу операторов (необязательная)	**3**

Для индексов GiST предусмотрены десять опорных функций, три из которых необязательные; они описаны в Таблице 13.

Таблица 13. Опорные функции GiST

Функция	**Описание**	**Номер опорной функции**
consistent	определяет, удовлетворяет ли ключ условию запроса	1
union	вычисляет объединение набора ключей	2
compress	вычисляет сжатое представление ключа или индексируемого значения	3
decompress	вычисляет развёрнутое представление сжатого ключа	4
penalty	вычисляет стоимость добавления нового ключа в поддерево с заданным ключом	5
picksplit	определяет, какие записи страницы должны быть перемещены в новую страницу, и вычисляет ключи объединения для результирующих страниц	6
equal	сравнивает два ключа и возвращает true, если они равны	7

Функция	Описание	Номер опорной функции
distance	определяет дистанцию от ключа до искомого значения (необязательная)	8
fetch	вычисляет исходное представление сжатого ключа для сканирования только по индексу (необязательная)	9
options	Определяет набор параметров, относящихся к данному классу операторов (необязательная)	10

Для индексов SP-GiST предусмотрены шесть опорных функций, одна из которых необязательная; они описаны в Таблице 14.

Таблица 14. Опорные функции SP-GiST

Функция	Описание	Номер опорной функции
config	предоставляет основную информацию о классе операторов	1
choose	определяет, как вставить новое значение во внутренний элемент	2
picksplit	определяет, как разделить множество значений	3
inner_consistent	определяет, в каких внутренних ветвях нужно искать заданное значение	4
leaf_consistent	определяет, удовлетворяет ли ключ условию запроса	5
options	Определяет набор параметров, относящихся к данному классу операторов (необязательная)	6

Для индексов GIN предусмотрены семь опорных функций, четыре из которых необязательные; они описаны в Таблице 15.

Таблица 15. Опорные функции GIN

Функция	Описание	Номер опорной функции
compare	сравнивает два ключа и возвращает целое меньше нуля, ноль или целое больше нуля, показывающее, что первый ключ меньше, равен или больше второго	1
extractValue	извлекает ключи из индексируемого значения	2
extractQuery	извлекает ключи из условия запроса	3
consistent	определяет, соответствует ли значение условию запроса (логическая вариация) (не требуется, если присутствует опорная функция 6)	4

Функция	Описание	Номер опорной функции
comparePartial	сравнивает частичный ключ из запроса с ключом из индекса и возвращает целое число меньше нуля, ноль или больше нуля, показывающее, что GIN должен игнорировать эту запись индекса, принять её как соответствующую или прекратить сканирование индекса (необязательная)	5
triConsistent	определяет, соответствует ли значение условию запроса (троичная вариация) (не требуется, если присутствует опорная функция 4)	6
options	Определяет набор параметров, относящихся к данному классу операторов (необязательная)	7

Для индексов BRIN предусмотрены пять базовых опорных функций, перечисленных в Таблице 16. Для некоторых видов базовых функций может потребоваться предоставить дополнительные опорные функции.

Таблица 16. Опорные функции BRIN

Функция	Описание	Номер опорной функции
opcInfo	возвращает внутреннюю информацию, описывающую сводные данные по индексированным столбцам	1
add_value	добавляет новое значение в существующий сводный кортеж индекса	2
consistent	определяет, удовлетворяет ли значение условию запроса	3
union	вычисляет объединение двух обобщающих кортежей	4
options	Определяет набор параметров, относящихся к данному классу операторов (необязательная)	5

В отличие от операторов поиска, опорные функции возвращают тот тип данных, который ожидает конкретный метод индекса; например, функция сравнения для B-деревьев возвращает знаковое целое. Количество и типы аргументов для каждой опорной функции так же зависят от метода индекса. Для методов B-дерева и хеша функции сравнения и хеширования принимают те же типы данных, что и операторы, включённые в класс операторов, но для большинства опорных функций GiST, SP-GiST, GIN и BRIN это не так.

28.1.14.4. Семейства и классы операторов

До этого неявно полагалось, что класс операторов работает только с одним типом данных. Хотя в конкретном индексируемом столбце, определённо, может быть только один тип данных, часто необходимо индексировать операции, сравнивающие значение столбца со значением другого типа. Также, если в сочетании с классом операторов возможно применение оператора, работающего с двумя типами, для другого типа данных обычно тоже создаётся собственный класс. В таких случаях полезно установить явную связь между связанными классами, так как это

поможет планировщику оптимизировать SQL-запросы (особенно для классов операторов B- дерева, потому что планировщик хорошо знает, как работать с ними).

Для удовлетворения этих потребностей в PG360 введена концепция *семейства операторов*. Семейство операторов содержит один или несколько классов операторов и может также содержать индексируемые операторы и соответствующие опорные функции, принадлежащие к семейству в целом, но не к какому-то одному классу в нём. Такая связь операторов и функций с семейством является «слабой», в отличие от обычной связи с определённым классом. Как правило, классы содержат операторы с операндами одного типа, тогда как межтиповые операторы слабо связываются с семейством.

Все операторы и функции в семействе операторов должны иметь совместимую семантику; требования к совместимости устанавливаются методом индекса. Смысл классов операторов в том, что они определяют, какая часть семейства необходима для поддержки некоторого индекса. Если существует индекс, использующий класс операторов, этот класс нельзя будет удалить, не удалив индекс — но другие части семейства, а именно, другие классы операторов и слабосвязанные операторы, удалить можно. Таким образом, класс операторов должен определяться так, чтобы он содержал минимальный набор операторов и функций, обоснованно требующихся для работы с индексом по определённому типу данных, а несущественные операторы могут добавляться в качестве слабосвязанных членов в семейство операторов.

В определении семейства «перегружаются» номера стратегий операторов и опорных функций: каждый номер фигурирует в семействе неоднократно. Это допускается, если для каждого экземпляра определённого номера задаются свои типы данных. Экземпляры, у которых оба входных типа совпадают с входным типом класса операторов, являются первичными операторами и опорными функциями для этого класса, и в большинстве случаев они должны объявляться в составе класса операторов, а не быть слабосвязанными членами семейства.

В семействе операторов B-дерева все операторы должны быть совместимыми в контексте сортировки. Для каждого оператора в семействе должна существовать опорная функция, принимающая на вход те же два типа, что и оператор. Семейство рекомендуется делать полным, то есть включать в него все операторы для каждого сочетания типов данных. В классы операторов следует включать только однотиповые операторы и опорные функции для определённого типа данных.

Чтобы создать семейство операторов хеширования для нескольких типов данных, необходимо создать совместимые функции поддержки хеша для каждого типа данных, который будет поддерживать семейство. Здесь под совместимостью понимается гарантия получения одного хеш– кода для любых двух значений, которые операторы сравнения в этом семействе считают равными, даже если они имеют разные типы. Обычно это сложно осуществить, когда типы имеют разное физическое представление, но в некоторых случаях всё же возможно. Более того, преобразование значения одного типа данных, представленного в семействе операторов, к другому типу, также представленному в этом семействе, путём неявного или двоичного сведения не должно менять значение вычисляемого хеша. Единственная опорная функция задаётся для типа данных, а не для оператора равенства. Семейство рекомендуется делать полным, то есть включить в него оператор равенства для всех сочетаний типов данных. В классы операторов следует

включать только однотиповый оператор равенства и опорную функция для определённого типа данных.

В индексах GiST, SP-GiST и GIN межтиповые операции явно не выражены. Множество поддерживаемых операторов определяется только теми операциями, которые могут выполнять основные опорные функции заданного класса операторов.

В BRIN требования зависят от инфраструктуры, предоставляющей классы операторов. Для классов операторов, построенных на инфраструктуре minmax, требуется то же поведение, что и для семейств операторов B-дерева: все операторы в семействе должны поддерживать совместимый порядок, а приведения не должны влиять на установленный порядок сортировки.

28.1.145. Системные зависимости от классов операторов

PG360 использует классы операторов для наделения операторов такими свойствами, которые могут использоваться не только для индексов.

В частности, это касается SQL-конструкций ORDER BY и DISTINCT, для которых требуется сравнивать и упорядочивать значения. Чтобы эти конструкции работали с определённым пользователем типом данных, PG360 задействует класс операторов B-дерева по умолчанию для этого типа. Член «равно» этого класса определяет, как система будет понимать равенство значений для GROUP BY и DISTINCT, а порядок сортировки, задаваемый классом операторов, определяет порядок ORDER BY по умолчанию.

Если класс операторов B-дерева по умолчанию для типа данных не определён, система будет искать класс операторов хеширования по умолчанию. Но так как подобный класс поддерживает только равенство, с ним будет возможна только группировка, но не сортировка.

Если для типа не определён класс операторов по умолчанию, попытавшись использовать эти конструкции SQL с данным типом, получится ошибка вида «не удалось найти оператор сортировки».

Сортировка с нестандартным классом операторов B-дерева возможна, если указать в предложении

```
USING оператор «меньше или равно» в данном классе:
SELECT * FROM mytable ORDER BY somecol USING ~<~;
```

Также возможно выполнить сортировку в порядке по убыванию, если указать в USING оператор «больше или равно».

Сравнение массивов пользовательских типов также производится в зависимости от семантики, определённой классом операторов B-дерева. Если класс операторов B-дерева по умолчанию для данного типа не определён, но имеется класс операторов хеширования, то будет поддерживаться сравнение массивов, но не упорядочивание.

28.1.146. Операторы упорядочивания

Некоторые методы доступа индексов (в настоящее время только GiST и SP-GiST) поддерживают концепцию *операторов упорядочивания*. Операторы, которые обсуждались до этого, были *операторами поиска*. Оператором поиска называется такой оператор, для которого можно выполнить поиск по индексу и найти все строки, удовлетворяющие условию WHERE *индексированный_столбец оператор константа*. Оператор упорядочивания, напротив, не

ограничивает набор возвращаемых строк, но определяет их порядок. С таким оператором, просканировав индекс, можно получить строки в порядке, заданным указанием ORDER BY *индексированный_столбец оператор константа*. Такое определение объясняется тем, что оно поддерживает поиск ближайшего соседа, если этот оператор вычисляет расстояние.

Тогда как операторы поиска должны возвращать логические результаты, операторы упорядочивания обычно возвращают другой тип, например, float или numeric для расстояний. Этот тип, как правило, отличается от типа индексируемых данных. Чтобы избежать жёстко запрограммированных предположений о поведении различных типов данных, при объявлении оператора упорядочивания должно указываться семейство операторов B-дерева, определяющее порядок сортировки результирующего типа данных. Как было отмечено в предыдущем разделе, семейства операторов B-дерева определяют понятие упорядочивания для PG360, так что такое объявление оказывается естественным.

28.15. Упаковывание связанных объектов в расширение

Полезное расширение PG360 обычно включает несколько объектов SQL; например, с появлением нового типа данных могут потребоваться новые функции, новые операторы и новые классы операторов. Все эти объекты удобно собрать в один пакет, с тем чтобы упростить управление базой данных. В PG360 такие пакеты называются *расширениями*. Чтобы определить расширение, вам понадобится как минимум *файл скрипта* с командами SQL, создающими объекты расширения, и *управляющий файл*, в котором определяются несколько базовых свойств самого расширения. Если расширение написано на C, в него обычно также включается файл разделяемой библиотеки, содержащий скомпилированный код. Обеспечив наличие этих файлов, загрузить их в базу данных можно простой командой CREATE EXTENSION.

Основное преимущество расширений по сравнению с обычным SQL-скриптом, загружающим множество «разрозненных» объектов в базу данных, состоит в том, что PG360 будет понимать, что объекты расширения связаны вместе. При необходимости можно удалить все объекты одной командой DROP EXTENSION (разрабатывать отдельный скрипт «uninstall» не требуется). Утилита pg_dump знает, что не нужно выгружать отдельные объекты, составляющие расширение — вместо этого она просто включит в архивный файл команду CREATE EXTENSION. Это кардинально упрощает миграцию на новую версию расширения, которая может содержать новые или другие объекты по сравнению с предыдущей версией. При загрузке такого архива в базу данных обязательно наличие скрипта, управляющего файла и других файлов расширения.

PG360 не позволит удалить отдельный объект, содержащийся в расширении, кроме как при удалении всего расширения. Также можно изменить определение объекта, относящегося к расширению (например, командой CREATE OR REPLACE FUNCTION для функции), но изменённое определение не будет выгружено утилитой pg_dump. Такие изменения обычно разумны, только если они параллельно отражаются в файле скрипта расширения. В производственной среде обычно лучше создавать скрипт обновления расширения, который будет изменять относящиеся к расширению объекты.

Скрипт расширения может устанавливать права доступа для объектов, являющихся частью расширения, выполняя команды GRANT и REVOKE. Окончательный набор прав для каждого объекта (если они заданы) будет сохранён в системном каталоге pg_init_privs. При использовании pg_dump в выгружаемый скрипт будет выведена команда CREATE EXTENSION с последующими операторами GRANT и REVOKE, которые установят права, имевшие место в момент выгрузки.

PG360 в настоящее время не поддерживает скрипты расширений, выполняющие операторы CREATE POLICY или SECURITY LABEL. Ожидается, что такие команды будут выполняться после того, как расширение будет создано. Выгружая данные, pg_dump будет также включать в вывод все политики RLS и метки безопасности.

Механизм расширений также предоставляет средства для поддержки дополнительных скриптов, призванных изменять определение объектов SQL, содержащихся в расширении. Например, если версия расширения 1.1, по сравнению с версией 1.0, добавляет одну функцию и изменяет тело другой функции, автор расширения может предоставить *скрипт обновления*, который произведёт именно эти два изменения. Затем, воспользовавшись командой ALTER EXTENSION UPDATE, можно будет применить эти изменения и отследить, какая версия расширения фактически установлена в заданной базе данных.

Типы SQL-объектов, которые могут быть членами расширения, перечислены в описании ALTER EXTENSION. Не могут быть его членами, в частности, объекты уровня кластера, такие как базы данных, роли и табличные пространства, так как расширение существует только в рамках одной базы данных. (Скрипту расширения не запрещается создавать такие объекты, но если он сделает это, они не будут считаться частью расширения.) Несмотря на то, что таблица может быть членом расширения, её подчинённые объекты, такие как индексы, непосредственными членами расширения считаться не будут. Ещё один важный момент — схемы могут принадлежать расширениям, но не наоборот; поэтому расширение имеет неполное имя и не существует «внутри» какой-либо схемы. Однако объекты-члены расширения, будут относиться к схемам, если это уместно для их типов. Сами расширения могут иметь, а могут и не иметь основания владеть схемами, к которым относятся объекты-члены расширения.

Если скрипт расширения создаёт какие-либо временные объекты (например, временные таблицы), эти объекты будут считаться членами расширения до конца текущего сеанса, но удалятся автоматически в конце сеанса, как и должны временные объекты. Это является исключением из правила, запрещающего удаление объектов-членов расширения без удаления всего расширения.

28.1.15.1. Файлы расширений

Команда CREATE EXTENSION задействует управляющий файл расширения, который должен называться по имени расширения, с суффиксом .control, и должен быть помещён в каталог сервера SHAREDIR/extension. Должен быть также ещё минимум один SQL-скрипт, с именем, соответствующим шаблону *расширение--версия*.sql (например, foo--1.0.sql для версии 1.0 расширения foo). По умолчанию скрипт(ы) также помещается в каталог SHAREDIR/extension; но в управляющем файле можно задать и другой каталог.

Формат управляющего файла расширения не отличается от формата postgresql.conf, а именно представляет собой список присваиваний *имя_параметра* = *значение*, по одному в строке.

В нём также допускаются пустые строки и комментарии, начинающиеся с #. Все значения, отличные от единственного слова или числа, в нём должны заключаться в кавычки.

В управляющем файле могут устанавливаться следующие параметры:

– directory (string) – Каталог, содержащий SQL-скрипт(ы) расширения. Если только не задан абсолютный путь, это имя рассматривается относительно каталога сервера SHAREDIR. По умолчанию подразумевается указание directory = 'extension'.

– default_version (string) – Версия расширения по умолчанию (та, которая будет установлена, если в CREATE EXTENSION не будет указана никакая версия). Хотя этот параметр можно опустить, это приведёт к ошибке в CREATE EXTENSION без явного указания VERSION, что вряд ли будет желаемым поведением.

– comment (string) – Комментарий (произвольная строка) к расширению. Комментарий применяется при изначальном создании расширения, но не при обновлениях расширения (так как при этом мог бы заменяться комментарий, заданный пользователем). Комментарий расширения также можно задать посредством команды COMMENT в файле скрипта.

– encoding (string) – Кодировка символов, используемая в файлах скриптов. Её следует указать, если эти файлы содержат символы не из набора ASCII. По умолчанию предполагается, что эти файлы содержат текст в кодировке базы данных.

– module_pathname (string) – Значение этого параметра будет подставляться вместо каждого вхождения MODULE_PATHNAME в скриптах. Если этот параметр не задан, подстановка не производится. Обычно для этого параметра устанавливается значение $libdir/*имя_разделяемой_библиотеки*, а затем в командах CREATE FUNCTION для функций на языке C указывается MODULE_PATHNAME, чтобы в скриптах не приходилось жёстко задавать имя разделяемой библиотеки.

– requires (string) – Список имён расширений, от которых зависит данное, например, requires = 'foo, bar'. Эти расширения должны быть уже установлены, прежде чем можно будет установить данное.

– superuser (boolean) – Если этот параметр имеет значение true (по умолчанию), только суперпользователи смогут создать это расширение или обновить его до новой версии. Если он имеет значение false, для этого будет достаточно прав, необходимых для выполнения команд в установочном скрипте или скрипте обновления. Обычно значение true должно устанавливаться, если для выполнения какой-либо из команд в этих скриптах требуются права суперпользователя. Такие команды в любом случае не будут выполнены успешно, но лучше сообщить пользователю об ошибке заранее.

– trusted (boolean) – Если этот параметр имеет значение true (по умолчанию это не так), то расширение, для которого свойство superuser равно true, смогут устанавливать не только суперпользователи. А именно, установить его смогут любые пользователи, имеющие право CREATE в текущей базе данных. Когда пользователь, выполняющий CREATE EXTENSION, не является суперпользователем, но ему разрешена установка этого расширения посредством этого параметра, скрипт установки или обновления запускается от имени первоначального суперпользователя, а не от имени вызывающего пользователя. Этот параметр не играет роли, если свойство superuser равно false. Вообще говоря, этот параметр не следует устанавливать для расширений, которые могут открыть возможности, иначе доступные только суперпользователям,

например, предоставить доступ к файловой системе. Кроме того, если расширение помечается как доверенное, написание безопасных скриптов установки и обновления для него требует дополнительных усилий.

– relocatable (boolean) – Расширение является *перемещаемым*, если относящиеся к нему объекты после создания расширения можно переместить в другую схему. По умолчанию подразумевается false, то есть расширение не считается перемещаемым.

– schema (string) – Этот параметр может задаваться только для неперемещаемых расширений. Если он задан, расширение можно будет загрузить только в указанную схему и не в какую другую. Подробнее об этом рассказывается ниже. Параметр schema учитывается только при изначальном создании расширения, но не при его обновлении.

Помимо главного управляющего файла *расширение*.control, расширение может включать дополнительные управляющие файлы с именами вида *расширение*--*версия*.control. Если они присутствуют, они должны находиться в том же каталоге, что и основной скрипт. Дополнительные управляющие файлы имеют тот же формат, что и основной. Любые параметры, заданные в дополнительном управляющем файле, переопределяют параметры основного файла, когда выполняется установка этой версии расширения или обновление до неё. Однако параметры directory и default_version в дополнительных управляющих файлах задать нельзя.

SQL-скрипты расширений могут содержать любые команды SQL, за исключением команд управления транзакциями (BEGIN, COMMIT и т. д.) и команд, которые не могут выполняться внутри блока транзакции (например, VACUUM). Это объясняется тем, что эти скрипты неявно выполняются в блоке транзакции.

SQL-скрипты расширений также могут содержать строки, начинающиеся с \echo, и они будут игнорироваться (восприниматься как комментарии) механизмом расширений. Это часто используется для вывода ошибки в случае, если этот скрипт выполняется в psql, а не загружается командой CREATE EXTENSION. Если такое выполнение не предотвратить, пользователи могут случайно загрузить содержимое расширения как «разрозненные» объекты, а не как собственно расширение, и получить состояние, которое довольно сложно исправить.

Если скрипт расширения содержит строку @extowner@, она будет заменена именем (если требуется, заключённым в кавычки) пользователя, выполняющего команду CREATE EXTENSION или ALTER EXTENSION. Обычно это используется для доверенных расширений, в которых владельцем внутренних объектов назначается не первоначальный суперпользователь, а вызывающий пользователь. (Однако это следует делать с осторожностью. Например, если назначить обычного пользователя владельцем функции на языке C, это позволит ему повысить свои привилегии.)

Тогда как файлы скриптов могут содержать любые символы, допустимые в указанной кодировке, управляющие файлы могут содержать только ASCII-символы, так как указать кодировку этих файлов нет возможности. На практике это представляет проблему, только если использовать символы не из набора ASCII в комментарии расширения. В таких случаях рекомендуется не использовать параметр comment в управляющем файле, а вместо этого задать комментарий командой COMMENT ON EXTENSION в файле скрипта.

28.1.152. Перемещаемость расширений

У пользователей часто возникает желание загрузить объекты, содержащиеся в расширении, в схему, отличную от той, что выбрал автор расширения. Насколько это поддерживает расширение, описывается одним из трёх уровней:

– Полностью перемещаемое расширение может быть перемещено в другую схему в любое время, даже после того, как оно загружено в базу данных. Это осуществляется командой ALTER EXTENSION SET SCHEMA, которая автоматически переименовывает все объекты-члены расширения, перенося их в новую схему. Обычно это возможно, только если в расширении нет никаких внутренних предположений о том, в какой схеме находятся все его объекты. Кроме того, все объекты расширения должны находиться в одной исходной схеме (за исключением объектов, не принадлежащих схемам, как например, процедурные языки). Чтобы пометить расширение как полностью перемещаемое, необходимо установить relocatable = true в его управляющем файле.

– Расширение может быть перемещаемым в момент установки, но не после. Обычно это имеет место, когда скрипту расширения необходимо явно ссылаться на целевую схему, например, устанавливая свойства search_path для функций SQL. Для такого расширения нужно задать relocatable = false в его управляющем файле и обращаться к целевой схеме в скрипте по псевдоимени @extschema@. Все вхождения этого псевдоимени будут заменены именем выбранной целевой схемы перед выполнением скрипта. Пользователь может выбрать целевую схему в указании SCHEMA команды CREATE EXTENSION.

– Если расширение вовсе не поддерживает перемещение, необходимо установить в его управляющем файле relocatable = false, и также задать в параметре schema имя предполагаемой целевой схемы. Это предотвратит использование указания SCHEMA команды CREATE EXTENSION, если только оно задаёт не то же имя, что определёно в управляющем файле. Этот выбор обычно необходим, если в расширении делаются внутренние предположения об именах схемы, которые нельзя свести к использованию псевдоимени @extschema@. Механизм подстановки @extschema@ будет работать и в этом случае, хотя польза от него будет ограниченной, так как имя схемы определяется управляющим файлом.

В любом случае при выполнении файла скрипта параметр search_path изначально будет указывать на целевую схему; то есть, CREATE EXTENSION делает то же, что и:

SET LOCAL search_path TO @extschema@, pg_temp;

Это позволяет направить объекты, создаваемые скриптом, в целевую схему. Скрипт может изменить search_path, если пожелает, но обычно это нежелательно. Параметр search_path восстанавливает предыдущее значение по завершении CREATE EXTENSION.

Целевая схема определяется параметром schema (если он задан) в управляющем файле, либо указанием SCHEMA команды CREATE EXTENSION (если оно присутствует), а в противном случае выбирается текущая схема для создания объектов по умолчанию (первая указанная в параметре search_path вызывающего). Когда используется параметр управляющего файла schema, целевая схема будет создана, если она ещё не существует, но в двух других случаях она должна уже существовать.

Если в параметре requires в управляющем файле расширения указаны какие-либо расширения, необходимые для данного, их целевые схемы добавляются к начальному значению

search_path после целевой схемы нового расширения. Благодаря этому их объекты видны для скрипта нового расширения.

В целях безопасности схема pg_temp всегда автоматически добавляется в конец search_path.

Хотя неперемещаемое расширение может содержать объекты, распределяемые по нескольким схемам, обычно желательно поместить все объекты, предназначенные для внешнего использования, в одну схему, назначенную целевой схемой расширения. Такой порядок будет хорошо согласовываться со значением search_path по умолчанию в процессе создания зависимых расширений.

28.1.53. Конфигурационные таблицы расширений

Некоторые расширения включают конфигурационные таблицы, содержащие данные, которые могут быть добавлены или изменены пользователем после установки расширения. Обычно, если таблица является частью расширения, ни определение таблицы, ни её содержимое не будет выгружаться утилитой pg_dump. Но это поведение нежелательно для конфигурационных таблиц — изменения, внесённые в них пользователем, должны выгружаться; в противном случае расширение будет вести себя по-другому, когда будет загружено вновь.

Чтобы решить эту проблему, скрипт расширения может пометить созданную им таблицу или последовательность как конфигурационное отношение, в результате чего pg_dump включит в выгружаемые данные содержимое (но не определение) этой таблицы или последовательности. Для этого нужно вызвать функцию pg_extension_config_dump(regclass, text) после создания таблицы или последовательности, например следующим образом:

```
CREATE TABLE my_config (key text, value text); CREATE SEQUENCE my_config_seq;

SELECT pg_catalog.pg_extension_config_dump('my_config', ''); SELECT pg_catalog.pg_extension_config_dump('my_config_seq', '');
```

Так можно пометить любое число таблиц или последовательностей, в том числе последовательности, связанные со столбцами serial или bigserial.

Когда второй аргумент pg_extension_config_dump — пустая строка, pg_dump выгружает всё содержимое таблицы. Обычно это правильно, только если после создания скриптом расширения эта таблица изначально пуста. Если же в таблице оказывается смесь начальных данных и данных, добавленных пользователем, во втором аргументе pg_extension_config_dump передаётся условие WHERE, которое отфильтровывает данные, подлежащие выгрузке. Например, имея таблицу, созданную таким образом:

```
CREATE TABLE my_config (key text, value text, standard_entry boolean);

SELECT pg_catalog.pg_extension_config_dump('my_config', 'WHERE NOT standard_entry');
```

можно сделать так, чтобы поле standard_entry содержало true только для строк, создаваемых скриптом расширения.

Для последовательностей второй аргумент функции pg_extension_config_dump не имеет значения.

В более сложных ситуациях, когда пользователи могут модифицировать и изначально существовавшие строки, можно создать триггеры для конфигурационной таблицы, которые корректно пометят изменённые строки.

Условие фильтра, связанное с конфигурационной таблицей, можно изменить, повторно вызвав pg_extension_config_dump. (Обычно это находит применение в скрипте обновления расширения.) Единственный способ обозначить, что некоторая таблица более не является конфигурационной — разорвать её связь с расширением командой ALTER EXTENSION ... DROP TABLE.

Отношения внешних ключей между таблицами определяют порядок, в котором эти таблицы будет выгружать pg_dump. В частности, pg_dump попытается выгрузить сначала основную таблицу, а затем подчинённую. Так как отношения внешних ключей устанавливаются во время выполнения CREATE EXTENSION (до загрузки данных в таблицы), циклические зависимости не поддерживаются. Когда образуются циклические зависимости, данные тем не менее будут выгружены, но полученный архив нельзя будет восстановить обычным образом, потребуется вмешательство пользователя.

Последовательности, связанные со столбцами serial или bigserial, не обязательно помечать непосредственно, чтобы их состояние было сохранено. Для этой цели достаточно пометить только их родительское отношение.

282 Триггеры

Триггерные функции могут быть написаны на большинстве доступных процедурных языков, включая PL/pgSQL, PL/Tcl, PL/Perl и PL/Python.

Триггерные функции можно писать и на C, хотя большинство людей находит, что проще использовать один из процедурных языков. В настоящее время невозможно написать триггерную функцию на чистом SQL.

2821. Обзор механизма работы триггеров

Триггер является указанием, что база данных должна автоматически выполнить заданную функцию, всякий раз когда выполнен определённый тип операции. Триггеры можно использовать с таблицами (секционированными и обычными), с представлениями и с внешними таблицами.

Для обычных и сторонних таблиц можно определять триггеры, которые будут срабатывать до или после любой из команд INSERT, UPDATE или DELETE; либо один раз для каждой модифицируемой строки, либо один раз для оператора SQL. Триггеры на UPDATE можно установить так, чтобы они срабатывали, только когда в предложении SET оператора UPDATE упоминаются определённые столбцы. Также триггеры могут срабатывать для операторов TRUNCATE. Если происходит событие триггера, для обработки этого события в установленный момент времени вызывается функция триггера.

Для представлений триггеры могут быть определены для выполнения вместо операций INSERT, UPDATE и DELETE. Такие триггеры INSTEAD OF вызываются единожды для каждой строки, которая должна быть изменена в этом представлении. Именно функция триггера отвечает за то, чтобы произвести необходимые изменения в нижележащих базовых таблицах представления и должным образом возвращать изменённые строки, чтобы они появлялись в

представлении. Триггеры для представлений тоже могут быть определены так, что они будут выполняться единожды для всего оператора SQL, до или после операций INSERT, UPDATE или DELETE. Однако такие триггеры срабатывают, только если для представления определён триггер INSTEAD OF. В противном случае все операторы, обращающиеся к представлению, должны быть переписаны в виде операторов, обращающихся к нижележащим базовым таблицам, и тогда будут срабатывать триггеры, установленные для этих таблиц.

Триггерная функция должна быть создана до триггера. Она должна быть объявлена без аргументов и возвращать тип trigger. (Триггерная функция получает данные на вход посредством специально переданной структуры TriggerData, а не в форме обычных аргументов.)

После создания триггерной функции создаётся триггер с помощью CREATE TRIGGER. Одна и та же триггерная функция может быть использована для нескольких триггеров.

PG360 предлагает как *построчные*, так и *операторные* триггеры. В случае построчного триггера триггерная функция вызывается один раз для каждой строки, затронутой оператором, запустившим триггер. Операторный же триггер, напротив, вызывается только один раз при выполнении соответствующего оператора, независимо от количества строк, которые он затрагивает. В частности оператор, который не затрагивает никаких строк, всё равно приведёт к срабатыванию операторного триггера. Эти два типа триггеров также называют триггерами *уровня строк* и триггерами *уровня оператора*, соответственно. Триггеры на TRUNCATE могут быть определены только на уровне оператора, а не на уровне строк.

Триггеры также классифицируются в соответствии с тем, срабатывают ли они до, после или вместо операции. Они называются триггерами BEFORE, AFTER и INSTEAD OF, соответственно. Триггеры BEFORE уровня оператора срабатывают до того, как оператор начинает делать что- либо, тогда как триггеры AFTER уровня оператора срабатывают в самом конце работы оператора. Эти типы триггеров могут быть определены для таблиц, представлений или сторонних таблиц. Триггеры BEFORE уровня строки срабатывают непосредственно перед обработкой конкретной строки, в то время как триггеры AFTER уровня строки срабатывают в конце работы всего оператора (но до любого из триггеров AFTER уровня оператора). Эти типы триггеров могут определяться только для таблиц, в том числе сторонних, но не для представлений. Триггеры INSTEAD OF могут определяться только для представлений и только на уровне строк: они срабатывают для каждой строки сразу после того как строка представления идентифицирована как подлежащая обработке.

Оператор, нацеленный на родительскую таблицу в иерархии наследования или секционирования, не вызывает срабатывания триггеров уровня оператора для задействованных дочерних таблиц; срабатывать будут только такие триггеры для родительской таблицы. Однако если для этих дочерних таблиц установлены триггеры уровня строк, они будут срабатывать.

Если запрос INSERT содержит предложение ON CONFLICT DO UPDATE, возможно совместное применение и триггеров уровня строк BEFORE INSERT, и триггеров уровня строк BEFORE UPDATE, которое отразится в окончательном состоянии изменяемой строки, если в запросе задействуются столбцы EXCLUDED. При этом обращение к EXCLUDED не обязательно должно иметь место в обоих наборах триггеров BEFORE на уровне строк. Следует рассмотреть возможность получения неожиданного результата, когда имеются и триггеры BEFORE INSERT, и BEFORE UPDATE на уровне строки, и они вместе модифицируют добавляемую/изменяемую

строку (проблемы возможны, даже если изменения более или менее равнозначные, но при этом не идемпотентные). Триггеры UPDATE уровня оператора вызываются при ON CONFLICT DO UPDATE независимо от того, будут ли изменены какие-либо строки в результате UPDATE (и даже в случае, когда альтернативный путь UPDATE вообще не выбирается). При выполнении запроса INSERT с предложением ON CONFLICT DO UPDATE сначала выполняются триггеры BEFORE INSERT, затем триггеры BEFORE UPDATE, потом триггеры AFTER UPDATE и, наконец, AFTER INSERT (речь идёт о триггерах на уровне операторов).

Если оператор UPDATE в секционированной таблице должен переместить строку в другую секцию, это перемещение реализуется в результате выполнения DELETE в исходной секции и последующего INSERT в новой секции. При этом в исходной секции срабатывают все триггеры BEFORE UPDATE и BEFORE DELETE уровня строк. Затем в целевой секции срабатывают все триггеры BEFORE INSERT уровня строк. Следует иметь в виду, что в случаях, когда все эти триггеры модифицируют перемещаемую строку, полученный результат может быть неожиданным. Если рассматривать триггеры AFTER ROW, то применяться будут триггеры AFTER DELETE и AFTER INSERT, но не триггеры AFTER UPDATE, так как команда UPDATE заменяется на DELETE и INSERT. Если же рассматривать триггеры уровня операторов, ни триггеры DELETE, ни триггеры INSERT не будут срабатывать, даже если производится перемещение строк; сработают только триггеры UPDATE, установленные в целевой таблице оператора UPDATE.

Триггерные функции, вызываемые триггерами операторов, должны всегда возвращать NULL. Триггерные функции, вызываемые триггерами строк, могут вернуть строку таблицы (значение типа HeapTuple). У триггера уровня строки, срабатывающего до операции, есть следующий выбор:

– Можно вернуть NULL, чтобы пропустить операцию для текущей строки. Это указывает исполнителю запросов, что не нужно выполнять операцию со строкой вызвавшей триггер (вставку, изменение или удаление конкретной строки в таблице).

– Возвращаемая строка для триггеров INSERT или UPDATE будет именно той, которая будет вставлена или обновлена в таблице. Это позволяет триггерной функции изменять вставляемую или обновляемую строку.

Если в триггере BEFORE уровня строки не планируется использовать любой из этих вариантов, то нужно аккуратно вернуть в качестве результата ту же строку, которая была передана на вход (то есть строку NEW для триггеров INSERT и UPDATE, или строку OLD для триггеров DELETE).

Триггер уровня строки INSTEAD OF должен вернуть либо NULL, чтобы указать, что он не модифицирует базовые таблицы представления, либо он должен вернуть строку представления, полученную на входе (строку NEW для операций INSERT и UPDATE или строку OLD для операций DELETE). Отличное от NULL возвращаемое значение сигнализирует, что триггер выполнил необходимые изменения данных в представлении. Это приведёт к увеличению счётчика количества строк, затронутых командой. Для операций INSERT и UPDATE (и только для них) триггер может изменить строку NEW перед тем как её вернуть. В результате будут изменены данные, возвращаемые INSERT RETURNING или UPDATE RETURNING, при использовании, когда представление должно возвращать не те данные, что были получены.

Возвращаемое значение игнорируется для триггеров уровня строки, вызываемых после операции, поэтому они могут возвращать NULL.

Генерируемые столбцы заслуживают отдельного внимания. Сохраняемые генерируемые столбцы вычисляются после триггеров BEFORE и перед триггерами AFTER. Таким образом, в триггерах AFTER можно наблюдать сгенерированное значение. В триггерах BEFORE строка OLD, как можно было ожидать, содержит предыдущее значение, однако в строке NEW ещё не содержится новое сгенерированное значение, и обращаться к нему не следует. На уровне языка C содержимое столбца в этот момент считается неопределённым; более высокоуровневые языки должны блокировать обращения к сохраняемому генерируемому столбцу в строке NEW внутри триггера BEFORE. Изменённые в триггере BEFORE значения генерируемого столбца игнорируются и будут перезаписаны.

Если есть несколько триггеров на одно и то же событие для одной и той же таблицы, то они будут вызываться в алфавитном порядке по имени триггера. Для триггеров BEFORE и INSTEAD OF потенциально изменённая строка, возвращаемая одним триггером, становится входящей строкой для следующего триггера. Если любой из триггеров BEFORE или INSTEAD OF возвращает NULL, операция для этой строки прекращается и последующие триггеры (для этой строки) не срабатывают.

В определении триггера можно указать логическое условие WHEN, которое будет проверяться, чтобы посмотреть, нужно ли запускать триггер. В триггерах уровня строки в условии WHEN можно проверять старые и/или новые значения столбцов строки. (В триггерах уровня оператора также можно использовать условие WHEN.) В триггерах BEFORE условие WHEN вычисляется непосредственно перед тем, как триггерная функция будет выполнена, поэтому использование WHEN существенно не отличается от выполнения той же проверки в самом начале триггерной функции. Однако в триггерах AFTER условие WHEN вычисляется сразу после обновления строки и от этого зависит, будет ли поставлено в очередь событие запуска триггера в конце оператора или нет. Поэтому, когда условие WHEN в триггере AFTER не возвращает истину, не требуется ни постановка события в очередь, ни повторная выборка этой строки в конце оператора. Это может существенно ускорить работу операторов, изменяющих большое количество строк, с триггером, который должен сработать только для нескольких. В триггерах INSTEAD OF не поддерживается использование условий WHEN.

Как правило, триггеры BEFORE уровня строки используются для проверки или модификации данных, которые будут вставлены или изменены. Например, триггер BEFORE можно использовать для вставки текущего времени в столбец timestamp или проверки, что два элемента строки согласованы между собой. Триггеры AFTER уровня строки наиболее разумно использовать для каскадного обновления данных в других таблицах или проверки согласованности сделанных изменений с данными в других таблицах. Причина для такого разделения работы в том, что триггер AFTER видит окончательное значение строки, в то время как для триггера BEFORE это не так, ведь могут быть другие триггеры BEFORE, которые сработают позже. Если нет особых причин для выбора между триггерами BEFORE или AFTER, то триггер BEFORE предпочтительнее, так как не требует сохранения информации об операции до конца работы оператора.

Если триггерная функция выполняет команды SQL, эти команды могут заново запускать триггеры. Это известно как каскадные триггеры. Прямых ограничений на количество каскадных уровней не существует. Вполне возможно, что каскадные вызовы приведут к рекурсивному срабатыванию одного и того же триггера. Например, в триггере INSERT может выполняться команда, которая добавляет строку в эту же таблицу, тем самым опять вызывая триггер на INSERT. Обязанность программиста не допускать бесконечную рекурсию в таких случаях.

При определении триггера можно указывать аргументы. Цель включения аргументов в определение триггера в том, чтобы позволить разным триггерам с аналогичными требованиями вызывать одну и ту же функцию. В качестве примера можно создать обобщенную триггерную функцию, которая принимает два аргумента с именами столбцов и записывает текущего пользователя в первый аргумент и текущий штамп времени во второй. При правильном написании такая триггерная функция будет независима от конкретной таблицы, для которой она будет запускаться. Таким образом, одна и та же функция может использоваться при выполнении INSERT в любую таблицу с соответствующими столбцами, чтобы, например, автоматически отслеживать создание записей в транзакционной таблице. Для триггеров UPDATE аргументы также могут использоваться для отслеживания последних сделанных изменений.

У каждого языка программирования, поддерживающего триггеры, есть свой собственный метод доступа из триггерной функции к входным данным триггера. Входные данные триггера включают в себя тип события (например, INSERT или UPDATE), а также любые аргументы, перечисленные в CREATE TRIGGER. Для триггеров уровня строки входные данные также включают строку NEW для триггеров INSERT и UPDATE и/или строку OLD для триггеров UPDATE и DELETE.

Триггеры уровня оператора по умолчанию не имеют возможностей для проверки отдельных строк, модифицированных оператором. Но триггер AFTER STATEMENT может запросить создание для него *переходных таблиц,* чтобы ему были доступны наборы затрагиваемых операцией строк. Триггерам AFTER ROW также могут предоставляться переходные таблицы, чтобы они могли видеть все изменения в таблице, а не только изменения в отдельных строках, для которых они срабатывают. Метод обращения к переходным таблицам определяется применяемым языком программирования, но обычно переходные таблицы представляются как временные таблицы только для чтения, к которым в триггерной функции можно обращаться, выполняя SQL- команды.

2822. Видимость изменений в данных

Если в триггерной функции выполняются SQL-команды и эти команды обращаются к таблице, на которую создан триггер, то необходимо знать правила видимости данных, потому что они определяют, будут ли видеть эти SQL-команды изменения в данных, для которых сработал триггер. Кратко:

– Триггеры уровня оператора следуют простым правилам видимости: никакие из изменений, произведённых оператором, не видны в триггерах BEFORE, тогда как в триггерах AFTER видны все изменения.

– Изменение данных (вставка, обновление или удаление), заставляющее сработать триггер, *не видно* для команд SQL, выполняемых в триггере BEFORE уровня строки, потому что это изменение ещё не произошло.

– Тем не менее команды SQL, выполняемые в триггере BEFORE уровня строки, *будут* видеть изменения данных в строках, которые уже были обработаны в этом операторе. Это требует осторожности, так как порядок обработки строк в целом непредсказуемый; команда SQL, обрабатывающая множество строк, может делать это в любом порядке.

– Аналогично, триггер INSTEAD OF уровня строки увидит изменения данных, внесённые при предыдущих вызовах триггера INSTEAD OF для этой же внешней команды.

– Когда срабатывает триггер AFTER уровня строки, все изменения сделанные оператором уже выполнены и видны в вызываемой триггерной функции.

Если триггерная функция написана на одном из стандартных процедурных языков, вышеприведённые утверждения применимы, только если функция объявлена как VOLATILE. Функции объявленные как STABLE или IMMUTABLE в любом случае не будут видеть изменений, сделанных вызывающим оператором.

2823. Триггерные функции на языке C

Далее описываются низкоуровневые детали интерфейса для триггерных функций. Эта информация необходима только при разработке триггерных функций на языке C. При использовании языка более высокого уровня эти детали обрабатываются не видны. В большинстве случаев стоит рассмотреть возможность использования процедурного языка, прежде чем начать разрабатывать триггеры на C. В документации по каждому процедурному языку объясняется, как создавать триггеры на этом языке.

Триггерные функции должны использовать интерфейс функций «версии 1».

Когда функция вызывается диспетчером триггеров, ей не передаются обычные аргументы, но передаётся указатель «context», ссылающийся на структуру TriggerData. Функции на C могут проверить, вызваны ли они диспетчером триггеров или нет, выполнив макрос:

CALLED_AS_TRIGGER(fcinfo)

который разворачивается в:

((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))

Если возвращается истина, то fcinfo->context можно безопасно привести к типу TriggerData

- и использовать указатель на структуру TriggerData. Функция *не* должна изменять структуру

TriggerData или любые данные, которые на неё указывают.

struct TriggerData определяется в commands/trigger.h: typedef struct TriggerData

{

NodeTag type; TriggerEvent tg_event; Relation tg_relation;

HeapTuple tg_trigtuple;

HeapTuple tg_newtuple;

Trigger*tg_trigger; TupleTableSlot *tg_trigslot; TupleTableSlot *tg_newslot; Tuplestorestate *tg_oldtable; Tuplestorestate *tg_newtable; const Bitmapset *tg_updatedcols;

} TriggerData;

где элементы определяются следующим образом:

type – Всегда T_TriggerData.

tg_event – Описывает событие, для которого вызывается функция. Можно использовать следующие макросы для получения информации о tg_event:

TRIGGER_FIRED_BEFORE(tg_event) – Возвращает истину, если триггер сработал до операции.

TRIGGER_FIRED_AFTER(tg_event) – Возвращает истину, если триггер сработал после операции.

TRIGGER_FIRED_INSTEAD(tg_event) – Возвращает истину, если триггер сработал вместо операции.

TRIGGER_FIRED_FOR_ROW(tg_event) – Возвращает истину, если триггер сработал на уровне строки.

TRIGGER_FIRED_FOR_STATEMENT(tg_event) – Возвращает истину, если триггер сработал на уровне оператора.

TRIGGER_FIRED_BY_INSERT(tg_event) – Возвращает истину, если триггер сработал для операции INSERT.

TRIGGER_FIRED_BY_UPDATE(tg_event) – Возвращает истину, если триггер сработал для операции UPDATE.

TRIGGER_FIRED_BY_DELETE(tg_event) – Возвращает истину, если триггер сработал для операции DELETE.

TRIGGER_FIRED_BY_TRUNCATE(tg_event) – Возвращает истину, если триггер сработал для операции TRUNCATE.

tg_relation – Указатель на структуру, описывающую таблицу, для которой сработал триггер. Подробнее об этой структуре описано в файле utils/rel.h.

tg_trigtuple – Указатель на строку, для которой сработал триггер. Это строка, которая вставляется, обновляется или удаляется. При срабатывании триггера для INSERT или DELETE это значение нужно вернуть из функции, только если не планируется изменять строку (в случае INSERT) или пропускать операцию для этой строки.

tg_newtuple – Для триггера на UPDATE это указатель на новую версию строки либо NULL, если триггер на INSERT или DELETE. Это значение нужно вернуть из функции в случае UPDATE, если не планируется изменять строку или пропускать операцию для этой строки.

tg_trigger – Указатель на структуру с типом Trigger, определённую в utils/reltrigger.h: typedef struct Trigger

```
{
Oid     tgoid;
char    *tgname;
Oid     tgfoid;
int16   tgtype;
char    tgenabled;
bool    tgisinternal;
bool    tgisclone;
Oid     tgconstrrelid;
```

```
Oid     tgconstrindid;
Oid     tgconstraint;
bool    tgdeferrable;
bool    tginitdeferred;
int16   tgnargs;
int16   tgnattr;
int16   *tgattr;
char    **tgargs;
char    *tgqual;
char    *tgoldtable;
char    *tgnewtable;
} Trigger;
```

где tgname — имя триггера, tgnargs — количество аргументов в tgargs, и tgargs — массив указателей на аргументы, указанные в команде CREATE TRIGGER. Остальные члены структуры предназначены для внутреннего использования.

tg_trigslot – Слот, содержащий tg_trigtuple, или указатель NULL, если такой строки нет.

tg_newslot – Слот, содержащий tg_newtuple, или указатель NULL, если такой строки нет.

tg_oldtable – Указатель на структуру типа Tuplestorestate, содержащую ноль или несколько строк в формате, определяемом содержимым tg_relation, или указатель NULL, если переходное отношение OLD TABLE отсутствует.

tg_newtable – Указатель на структуру типа Tuplestorestate, содержащую ноль или несколько строк в формате, определяемом содержимым tg_relation, или указатель NULL, если переходное отношение NEW TABLE отсутствует.

tg_updatedcols – Для триггеров UPDATE — битовая карта, в которой отмечается, какие столбцы изменила команда, вызвавшая срабатывание триггера. Используя её, универсальные триггерные функции могут оптимизировать свои действия, не обращая внимания на столбцы, которые не были изменены.

Определить, вошёл ли в битовую карту столбец с атрибутом под номером attnum (считая с 1), можно следующим образом: bms_is_member(attnum – FirstLowInvalidHeapAttributeNumber, trigdata->tg_updatedcols)).

Для всех остальных триггеров содержит NULL.

Чтобы обращаться к переходным таблицам в запросах, выполняемых через SPI, необходимо использовать SPI_register_trigger_data.

Триггерная функция должна возвращать указатель HeapTuple или указатель NULL (но *не* SQL значение null, то есть не нужно устанавливать *isNull* в истину). Если не планируется менять обрабатываемую триггером строку, то нужно вернуть либо tg_trigtuple, либо tg_newtuple.

283. Триггеры событий

В дополнение к триггерам PG360 также предоставляет триггеры событий. В отличие от обычных триггеров, которые подключаются к конкретной таблице и работают только с командами DML, триггеры событий определяются на уровне базы данных и работают с командами DDL.

Как и обычные триггеры, триггеры событий можно создавать на любом процедурном языке, поддерживающим триггеры событий, а также на C, но не на чистом SQL.

2831. Обзор механизма работы триггеров событий

Триггер события срабатывает всякий раз, когда в базе данных, в которой он определён, происходит связанное с ним событие. В настоящий момент поддерживаются следующие события: ddl_command_start, ddl_command_end, table_rewrite и sql_drop. Поддержка дополнительных событий может быть добавлена в будущих выпусках.

Событие ddl_command_start происходит непосредственно перед выполнением команд CREATE, ALTER, DROP, SECURITY LABEL, COMMENT, GRANT и REVOKE. Проверка на существование объекта перед срабатыванием триггера не производится. В качестве исключения, однако, это событие не происходит для команд DDL, обращающихся к общим объектам кластера базы данных — базам данных, табличным пространствам, ролям, а также к самим триггерам событий. Событие ddl_command_start также происходит непосредственно перед выполнением команды SELECT INTO, так как она равнозначна команде CREATE TABLE AS.

Событие ddl_command_end происходит непосредственно после выполнения команд из того же набора. Чтобы получить дополнительную информацию об операциях DDL, повлёкших произошедшее событие, необходимо вызвать функцию pg_event_trigger_ddl_commands(), возвращающую множество, из кода обработчика события ddl_command_end. Этот триггер срабатывает после того, как эти действия имели место (но до фиксации транзакции), так что в системных каталогах можно увидеть уже изменённое состояние.

Событие sql_drop происходит непосредственно перед событием ddl_command_end для команд, которые удаляют объекты базы данных. Для получения списка удалённых объектов необходимо использовать возвращающую набор строк функцию pg_event_trigger_dropped_objects() в триггере события sql_drop. Триггер выполняется после удаления объектов из таблиц системного каталога, поэтому их невозможно больше увидеть.

Событие table_rewrite происходит только после того, как таблица будет перезаписана в результате определённых действий команд ALTER TABLE и ALTER TYPE. Хотя перезапись таблицы может быть вызвана и другими управляющими операторами, в частности CLUSTER и VACUUM, событие table_rewrite для них не вызывается.

Триггеры событий (как и прочие функции) не могут выполняться в прерванной транзакции. Поэтому, если команда DDL завершается ошибкой, соответствующие триггеры ddl_command_end не сработают. И наоборот, если триггер ddl_command_end завершился с ошибкой, последующие триггеры событий не сработают, так же как и сама команда не будет выполняться. Похожим образом, если триггер ddl_command_end завершится ошибкой, действие команды DDL будет отменено, как это происходит при возникновении ошибки внутри транзакции.

Для создания триггера события используется команда CREATE EVENT TRIGGER. Предварительно нужно создать функцию, со специальным возвращаемым типом event_trigger. Данная функция не обязана возвращать значение (и может не возвращать). Возвращаемый тип служит лишь указанием на то, что функция будет вызываться из триггера события.

Если есть несколько триггеров на одно и то же событие, то они будут вызываться в алфавитном порядке по имени триггера.

В определении триггера можно использовать условие WHEN, чтобы, например, триггер ddl_command_start срабатывал только для отдельных команд, которые нужно перехватить. Триггеры событий часто используются для ограничения диапазона DDL-команд, доступных пользователям.

2832. Матрица срабатывания триггеров событий

В Таблице 17 перечислены команды, для которых поддерживаются триггеры событий.

Таблица 17. Поддержка триггеров событий командами DDL

Тег команды	**ddl_ command_ start**	**ddl_ command_ end**	**sql_drop**	**table_ rewrite**	**Замечания**
ALTER AGGREGATE	X	X	-	-	
ALTER COLLATION	X	X	-	-	
ALTER CONVERSION	X	X	-	-	
ALTER DOMAIN	X	X	-	-	
ALTER DEFAULT PRIVILEGES	X	X	-	-	
ALTER EXTENSION	X	X	-	-	
ALTER FOREIGN DATA WRAPPER	X	X	-	-	
ALTER FOREIGN TABLE	X	X	X	-	
ALTER FUNCTION	X	X	-	-	
ALTER LANGUAGE	X	X	-	-	
ALTER LARGE OBJECT	X	X	-	-	
ALTER MATERIALIZED VIEW	X	X	-	-	
ALTER OPERATOR	X	X	-	-	
ALTER OPERATOR CLASS	X	X	-	-	
ALTER OPERATOR FAMILY	X	X	-	-	
ALTER POLICY	X	X	-	-	
ALTER PROCEDURE	X	X	-	-	
ALTER PUBLICATION	X	X	-	-	
ALTER ROUTINE	X	X	-	-	
ALTER SCHEMA	X	X	-	-	
ALTER SEQUENCE	X	X	-	-	
ALTER SERVER	X	X	-	-	
ALTER STATISTICS	X	X	-	-	
ALTER SUBSCRIPTION	X	X	-	-	
ALTER TABLE	X	X	X	X	
ALTER TEXT SEARCH CONFIGURATION	X	X	-	-	
ALTER TEXT SEARCH DICTIONARY	X	X	-	-	
ALTER TEXT SEARCH PARSER	X	X	-	-	

Тег команды	ddl_ command_ start	ddl_ command_ end	sql_drop	table_ rewrite	Замечания
ALTER TEXT SEARCH TEMPLATE	X	X	-	-	
ALTER TRIGGER	X	X	-	-	
ALTER TYPE	X	X	-	X	
ALTER USER MAPPING	X	X	-	-	
ALTER VIEW	X	X	-	-	
COMMENT	X	X	-	-	Только для локальных объектов
CREATE ACCESS METHOD	X	X	-	-	
CREATE AGGREGATE	X	X	-	-	
CREATE CAST	X	X	-	-	
CREATE COLLATION	X	X	-	-	
CREATE CONVERSION	X	X	-	-	
CREATE DOMAIN	X	X	-	-	
CREATE EXTENSION	X	X	-	-	
CREATE FOREIGN DATA WRAPPER	X	X	-	-	
CREATE FOREIGN TABLE	X	X	-	-	
CREATE FUNCTION	X	X	-	-	
CREATE INDEX	X	X	-	-	
CREATE LANGUAGE	X	X	-	-	
CREATE MATERIALIZED VIEW	X	X	-	-	
CREATE OPERATOR	X	X	-	-	
CREATE OPERATOR CLASS	X	X	-	-	
CREATE OPERATOR FAMILY	X	X	-	-	
CREATE POLICY	X	X	-	-	
CREATE PROCEDURE	X	X	-	-	
CREATE PUBLICATION	X	X	-	-	
CREATE RULE	X	X	-	-	
CREATE SCHEMA	X	X	-	-	
CREATE SEQUENCE	X	X	-	-	
CREATE SERVER	X	X	-	-	
CREATE STATISTICS	X	X	-	-	
CREATE SUBSCRIPTION	X	X	-	-	
CREATE TABLE	X	X	-	-	
CREATE TABLE AS	X	X	-	-	
CREATE TEXT SEARCH CONFIGURATION	X	X	-	-	
CREATE TEXT SEARCH DICTIONARY	X	X	-	-	

Тег команды	ddl_ command_ start	ddl_ command_ end	sql_drop	table_ rewrite	Замечания
CREATE TEXT SEARCH PARSER	X	X	-	-	
CREATE TEXT SEARCH TEMPLATE	X	X	-	-	
CREATE TRIGGER	X	X	-	-	
CREATE TYPE	X	X	-	-	
CREATE USER MAPPING	X	X	-	-	
CREATE VIEW	X	X	-	-	
DROP ACCESS METHOD	X	X	X	-	
DROP AGGREGATE	X	X	X	-	
DROP CAST	X	X	X	-	
DROP COLLATION	X	X	X	-	
DROP CONVERSION	X	X	X	-	
DROP DOMAIN	X	X	X	-	
DROP EXTENSION	X	X	X	-	
DROP FOREIGN DATA WRAPPER	X	X	X	-	
DROP FOREIGN TABLE	X	X	X	-	
DROP FUNCTION	X	X	X	-	
DROP INDEX	X	X	X	-	
DROP LANGUAGE	X	X	X	-	
DROP MATERIALIZED VIEW	X	X	X	-	
DROP OPERATOR	X	X	X	-	
DROP OPERATOR CLASS	X	X	X	-	
DROP OPERATOR FAMILY	X	X	X	-	
DROP OWNED	X	X	X	-	
DROP POLICY	X	X	X	-	
DROP PROCEDURE	X	X	X	-	
DROP PUBLICATION	X	X	X	-	
DROP ROUTINE	X	X	X	-	
DROP RULE	X	X	X	-	
DROP SCHEMA	X	X	X	-	
DROP SEQUENCE	X	X	X	-	
DROP SERVER	X	X	X	-	
DROP STATISTICS	X	X	X	-	
DROP SUBSCRIPTION	X	X	X	-	
DROP TABLE	X	X	X	-	
DROP TEXT SEARCH CONFIGURATION	X	X	X	-	
DROP TEXT SEARCH DICTIONARY	X	X	X	-	
DROP TEXT SEARCH PARSER	X	X	X	-	

Тег команды	ddl_ command_ start	ddl_ command_ end	sql_drop	table_ rewrite	Замечания
DROP TEXT SEARCH TEMPLATE	X	X	X	-	
DROP TRIGGER	X	X	X	-	
DROP TYPE	X	X	X	-	
DROP USER MAPPING	X	X	X	-	
DROP VIEW	X	X	X	-	
GRANT	X	X	-	-	Только для локальных объектов
IMPORT FOREIGN SCHEMA	X	X	-	-	
REFRESH MATERIALIZED VIEW	X	X	-	-	
REVOKE	X	X	-	-	Только для локальных объектов
SECURITY LABEL	X	X	-	-	Только для локальных объектов
SELECT INTO	X	X	-	-	

2833. Триггерные функции событий на языке C

Далее описываются низкоуровневые детали интерфейса для событийных триггерных функций. Эта информация необходима только при разработке событийных триггерных функций событий на языке C. При использовании языка более высокого уровня эти детали не видны. В большинстве случаев стоит рассмотреть возможность использования процедурного языка, прежде чем начать разрабатывать событийные триггеры на C. В документации по каждому процедурному языку объясняется, как создавать событийные триггеры на этом языке.

Триггерные функции событий должны использовать «version 1» интерфейса диспетчера функций.

Когда функция вызывается диспетчером триггеров событий, ей не передаются обычные аргументы, но передаётся указатель «context», ссылающийся на структуру EventTriggerData. Функции на C могут проверить вызваны ли они диспетчером триггеров событий или нет выполнив макрос:

CALLED_AS_EVENT_TRIGGER(fcinfo)

который разворачивается в:

EventTriggerData

Если возвращается истина, то fcinfo->context можно безопасно привести к типу EventTriggerData * и использовать указатель на структуру EventTriggerData. Функция *не* должна изменять структуру EventTriggerData или любые данные, которые на неё указывают.

struct EventTriggerData определена в commands/event_trigger.h: typedef struct EventTriggerData

{

NodeTag type;

const char *event; /* имя события */ Node *parsetree; /* дерево разбора */ CommandTag tag; /* тег команды */

} EventTriggerData;

со следующими членами структуры:

type

Всегда T_EventTriggerData.

event – Описывает событие, для которого вызывается функция. Возможные значения: "ddl_command_start", "ddl_command_end", "sql_drop", "table_rewrite".

parsetree – Указатель на дерево разбора команды. Структура дерева разбора может быть изменена без предупреждений.

Tag – Тег команды, для которой сработал триггер события. Например "CREATE FUNCTION".

Функция триггера события должна возвращать указатель NULL (но *не* SQL значение null, то есть не нужно устанавливать *isNull* в истину).

284. Система правил

Далее описана система правил, реализованная в PG360. В некоторых других базах данных определяются активные правила баз данных, которые обычно реализуются в виде процедур и триггеров. Так же их можно реализовать и в PG360.

Система правил (точнее говоря, система правил перезаписи запросов) полностью отличается от механизма хранимых процедур и триггеров. Она изменяет запросы по заданным правилам, а затем передаёт модифицированный запрос планировщику для планирования и выполнения. Это очень мощное средство, подходящее для решения множества задач, например, для определения представлений и процедур на языке запросов или реализации версионности. Теоретические основы и преимущества этой системы правил также описаны в ston90b и ong90 (на английском языке).

284.1. Дерево запроса

Система правил внедрена между анализатором запросов и планировщиком. Она принимает разобранный запрос, одно дерево запроса, и определённые пользователем правила перезаписи, тоже представленные деревьями с некоторой дополнительной информацией, и создаёт некоторое количество деревьев запросов в результате. Таким образом, на входе и выходе этой системы оказывается то, что может сформировать анализатор запросов, и как следствие, всё, с чем работает эта система, представимо в виде операторов SQL.

Дерево запроса – это внутреннее представление оператора SQL, в котором все образующие его части хранятся отдельно. Эти деревья можно увидеть в журнале сервера, если установить параметры конфигурации debug_print_parse, debug_print_rewritten или debug_print_plan. Действия правил также хранятся в виде деревьев запросов, в системном каталоге pg_rewrite. Они не форматируются как при выводе в журнал, но содержат точно такую же информацию.

Дерево запроса состоит из следующих частей:

– тип команды – Это простое значение, говорящее, какая команда (SELECT, INSERT, UPDATE или DELETE) сгенерировала дерево запросов.

– список отношений – Список отношений представляет собой массив отношений, используемых в запросе. В запросе SELECT он включает отношения, указанные после ключевого слова FROM.

Каждый элемент списка отношений представляет таблицу или представление и говорит, с каким именем они упоминаются в других частях запроса. В дереве запросов записываются номера элементов списка отношений, а не их имена, поэтому для него неактуальна проблема дублирования имён, как для оператора SQL. Такая проблема может возникнуть при объединении списков отношений, образованных разными правилами.

– результирующее отношение – Индекс в списке отношений, указывающий на отношение, которое будет получать результаты запроса.

В запросах SELECT результирующее отношение отсутствует. (Особый случай SELECT INTO практически равнозначен CREATE TABLE с последующим INSERT ... SELECT и здесь отдельно не рассматривается.)

Для команд INSERT, UPDATE и DELETE результирующим отношением будет таблица (или представление!), в которой будут происходить изменения.

– выходной список — Это список выражений, определяющих результат запроса. В случае SELECT, это выражения, которые образуют окончательный набор выходных данных. Они соответствуют выражениям, записанным между ключевыми словами SELECT и FROM. (Указание * — это просто краткое обозначение имён всех столбцов отношения. Анализатор разворачивает его в список отдельных столбцов, так что система правил никогда не видит его.)

Командам DELETE не нужен обычный выходной список, так как они не выдают никакие результаты. Вместо этого планировщик добавляет в пустой выходной список специальную запись CTID, чтобы исполнитель мог найти удаляемую строку. (CTID добавляется, когда результирующее отношение — обычная таблица. Если это представление, планировщиком добавляется переменная, содержащая всю строку.)

Для команд INSERT выходной список описывает новые строки, которые должны попасть в результирующее отношение. Он включает выражения в предложении VALUES или предложении SELECT в INSERT ... SELECT. На первом этапе процесс перезаписи добавляет элементы выходного списка для столбцов, которым ничего не присвоила исходная команда, но имеющих значения по умолчанию. Все остальные столбцы (без заданного значения и значения по умолчанию) планировщик заполняет константой NULL.

Для команд UPDATE выходной список описывает новые строки, которые должны заменить старые. В системе правил он содержит только выражения из части SET столбец = выражение. Для пропущенных столбцов планировщик вставляет выражения, копирующие значения из старой строки в новую. Так же, как и с командой DELETE, при этом добавляется CTID или переменная со всей строкой, чтобы исполнитель мог найти изменяемую старую строку.

Каждая запись в выходном списке содержит выражение, которое может быть константой, переменной, указывающей на столбец отношения в таблице отношений, параметром или деревом выражений, образованным из констант, переменных, операторов, вызовов функций и т. д.

– условие фильтра. Условие фильтра запроса — это выражение, во многом похожее на те, что содержатся в выходном списке. Результат этого выражения — логический, он говорит, должна

ли выполняться операция (INSERT, UPDATE, DELETE или SELECT) для данной строки в результате. Оно соответствует предложению WHERE SQL-оператора.

– дерево соединения. Дерево соединения запроса показывает структуру предложения FROM. Для простых запросов вида SELECT ... FROM a, b, c, дерево соединения — это просто список элементов FROM, так как они могут соединяться в любом порядке. Но с выражениями JOIN, особенно с внешними соединениями, приходится соединять отношения именно в заданном порядке. В этом случае дерево соединения отражает структуру выражений JOIN. Ограничения, связанные с конкретными предложениями JOIN (из выражений ON или USING), тоже сохраняются в виде условных выражений, добавленных к соответствующим узлам дерева соединения. Как оказалось, выражение WHERE верхнего уровня тоже удобно хранить как условие, добавленное к элементу верхнего уровня дерева соединения. Поэтому в дереве соединения на самом деле представляются оба предложения оператора SELECT — FROM и WHERE.

– другие. Другие части дерева запроса, например, предложение ORDER BY, в данном контексте не представляют интереса. Система правил выполняет в них некоторые подстановки, применяя правила, но это не имеет непосредственного отношения к основам системы правил.

2842. Система правил и представления

Представления в PG360 реализованы на основе системы правил. Фактически нет никакого отличия

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

от следующих двух команд:

```
CREATE TABLE myview (same column list as mytab);
CREATE RULE "_RETURN" AS ON SELECT TO myview DO INSTEAD
SELECT * FROM mytab;
```

так как именно эти действия CREATE VIEW выполняет внутри. Информация о представлениях в системных каталогах PG360 не отличается от информации о таблицах. Поэтому при анализе запроса нет разницы между таблицами и представлениями. Они представляют собой одно и то же — отношения.

28421. Работа правил SELECT

Правила ON SELECT применяются ко всем запросам на последнем этапе, даже если это команда INSERT, UPDATE или DELETE. Эти правила отличаются от правил других видов тем, что они модифицируют непосредственно дерево запросов, а не создают новое.

В настоящее время возможно только одно действие в правиле ON SELECT и это должно быть безусловное действие SELECT, выполняемое в режиме INSTEAD. Это ограничение было введено, чтобы сделать правила достаточно безопасными для применения обычными пользователями, так что действие правил ON SELECT сводится к реализации представлений.

28422. Правила представлений не для SELECT

До этого в описании правил представлений не затрагивались два компонента дерева запросов — тип команды и результирующее отношение. На самом деле, тип команды не важен для

правил представления, но результирующее отношение может повлиять на работу механизма перезаписи, потому что если это представление, требуются дополнительные операции.

Есть только несколько отличий между деревом запроса для SELECT и деревом для другой команды. Очевидно, у них различные типы команд, и для команды, отличной от SELECT, результирующее отношение указывает на элемент в списке отношений, куда должен попасть результат. Все остальные компоненты в точности те же. Поэтому, например, если взять таблицы t1 и t2 со столбцами a и b, деревья запросов для этих операторов:

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;
UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;
```

будут практически одинаковыми. В частности:

– Списки отношений содержат элементы для таблиц t1 и t2.

– Выходные списки содержат одну переменную, указывающую на столбец b элемента–отношения для таблицы t2.

– Выражения условий сравнивают столбцы a обоих элементов-отношений на равенство.

– Деревья соединений показывают простое соединение между t1 и t2.

Как следствие, для обоих деревьев строятся похожие планы выполнения, с соединением двух таблиц. Для UPDATE планировщик добавляет в выходной список недостающие столбцы из t1 и окончательное дерево становится таким:

```
UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;
```

В результате исполнитель, обрабатывающий соединение, выдаёт тот же результат, что и запрос:

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

Но при применении UPDATE, часть плана исполнителя, в которой выполняется соединение, не представляет, для чего предназначены результаты соединения. Она просто выдаёт результирующий набор строк. Фактически есть одна команда SELECT, а другая, UPDATE, обрабатывается исполнителем выше, где он уже знает, что это команда UPDATE и что результат должен попасть в таблицу t1. Но какие из строк таблицы должны заменяться новыми?

Для решения этой проблемы в выходной список операторов UPDATE (и DELETE) добавляется ещё один элемент: идентификатор текущего кортежа (Current Tuple ID, CTID). Это системный столбец, содержащий номер блока в файле и позицию строки в блоке. Зная таблицу, по CTID можно получить исходную строку в t1, подлежащую изменению. С добавленным в выходной список CTID запрос фактически выглядит следующим образом:

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

Старые строки таблицы не переписываются, поэтому ROLLBACK выполняется быстро. С командой UPDATE в таблицу вставляется новая строка результата (без CTID) и в заголовке старой строки, на которую указывает CTID, в поля cmax и xmax записываются текущий счётчик команд и идентификатор текущей транзакции. Таким образом, старая строка оказывается скрытой и после фиксирования транзакции процесс очистки может окончательно удалить неактуальную версию строки.

28423. Преимущества представлений в PG360

Преимущество реализации представлений через систему правил заключается в том, что планировщик получает в одном дереве запроса всю информацию о таблицах, которые нужно

прочитать, о том, как связаны эти таблицы, об условиях в представлениях, а также об условиях, заданных в исходном запросе. И всё это имеет место, когда сам исходный запрос представляет собой соединение представлений. Планировщик должен выбрать лучший способ выполнения запроса, и чем больше информации он получит, тем лучше может быть его выбор. И то, как в PG360 реализована система правил, гарантирует, что ему поступает вся информация, собранная о запросе на данный момент.

28424. Изменение представления

Если записать имя представления в качестве целевого отношения команды INSERT, UPDATE или DELETE? Если проделать подстановки, описанные выше, будет получено дерево запроса, в котором результирующее отношение указывает на элемент-подзапрос, что не будет работать. Однако PG360 даёт ряд возможностей, чтобы сделать представления изменяемыми.

Если подзапрос выбирает данные из одного базового отношения и он достаточно прост, механизм перезаписи может автоматически заменить его нижележащим базовым отношением, чтобы команды INSERT, UPDATE или DELETE обращались к базовому отношению. Представления, «достаточно простые» для этого, называются *автоматически изменяемыми*. Подробнее виды представлений, которые могут изменяться автоматически, описаны в CREATE VIEW.

Эту задачу также можно решить, создав триггер INSTEAD OF для представления. В этом случае перезапись будет работать немного по-другому. Для INSERT механизм перезаписи не делает с представлением ничего, оставляя его результирующим отношением запроса. Для UPDATE и DELETE ему по-прежнему придётся разворачивать запрос представления, чтобы получить «старые» строки, которые эта команда попытается изменить или удалить. Поэтому представление разворачивается как обычно, но в запрос добавляется ещё один элемент списка отношений, указывающий на представление в роли результирующего отношения.

При этом возникает проблема идентификации строк в представлении, подлежащих изменению. Когда результирующее отношение является таблицей, в выходной список добавляется специальное поле CTID, указывающее на физическое расположение изменяемых строк. Но это не будет работать, когда результирующее отношение — представление, так как в представлениях нет CTID, потому что их строки физически нигде не находятся. Вместо этого, для операций UPDATE или DELETE в выходной список добавляется специальный элемент wholerow (вся строка), который разворачивается в содержимое всех столбцов представления. Используя этот элемент, исполнитель передаёт строку «old» в триггер INSTEAD OF. Какие именно строки должны изменяться фактически, будет решать сам триггер, исходя из полученных значений старых и новых строк.

Кроме того, пользователь может определить правила INSTEAD, в которых задать действия замены для команд INSERT, UPDATE и DELETE с представлением. Эти правила обычно преобразуют команду в другую команду, изменяющую одну или несколько таблиц, а не представление.

Такие правила вычисляются сначала, перезаписывая исходный запрос до того, как он будет планироваться и выполняться. Поэтому, если для представления определены и триггеры INSTEAD

OF, и правила для INSERT, UPDATE или DELETE, сначала вычисляются правила, а в зависимости от их действия, триггеры могут не вызываться вовсе.

Автоматическая перезапись запросов INSERT, UPDATE или DELETE с простыми представлениями всегда производится в последнюю очередь. Таким образом, если у представления есть правила или триггеры, они переопределяют поведение автоматически изменяемых представлений.

Если для представления не определены правила INSTEAD или триггеры INSTEAD OF, и запрос не удаётся автоматически переписать в виде обращения к нижележащему базовому отношению, возникает ошибка, потому что исполнитель не сможет изменить такое представление.

28.4.3. Материализованные представления

Материализованные представления в PG360 основаны на системе правил, как и представления, но их содержимое сохраняется как таблица. Основное отличие между:

```
CREATE MATERIALIZED VIEW mymatview AS SELECT * FROM mytab;
```

и этой командой:

```
CREATE TABLE mymatview AS SELECT * FROM mytab;
```

состоит в том, что материализованное представление впоследствии нельзя будет изменить непосредственно, а запрос, создающий материализованное представление, сохраняется точно так же, как запрос представления, и получить актуальные данные в материализованном представлении можно следующим образом:

```
REFRESH MATERIALIZED VIEW mymatview;
```

Информация о материализованном представлении в системных каталогах PG360 не отличается от информации о таблице или представлении. Поэтому для анализатора запроса материализованное представление является просто отношением, как таблица или представление. Когда запрос обращается к материализованному представлению, данные возвращаются непосредственно из него, как из таблицы; правило применяется, только чтобы его наполнить.

Хотя обращение к данным в материализованном представлении часто выполняется гораздо быстрее, чем обращение к нижележащим таблицам напрямую или через представление, данные в нём не всегда актуальные (но иногда это вполне приемлемо).

Ещё одно применение материализованного представления — предоставить быстрый доступ к данным, получаемым с удалённой системы через обёртку сторонних данных.

28.4.4. Правила для INSERT, UPDATE и DELETE

Правила, определяемые для команд INSERT, UPDATE и DELETE, значительно отличаются от правил представлений, описанных в ранее. Во-первых, команда CREATE RULE позволяет создавать правила со следующими особенностями:

– они могут не определять действия;

– они могут определять несколько действий;

– они могут действовать в режиме INSTEAD или ALSO (по умолчанию);

– становятся полезными псевдоотношения NEW и OLD;

– они могут иметь условия применения.

Во-вторых, они не модифицируют само исходное дерево запроса. Вместо этого они создают несколько новых деревьев запросов и могут заменить исходное.

28441. Работа правил для изменения

Синтаксис:

```
CREATE [ OR REPLACE ] RULE имя AS ON событие
TO таблица [ WHERE условие ]
DO [ ALSO | INSTEAD ] { NOTHING | команда | ( команда ; команда
... ) }
```

В дальнейшем, под *правилами для изменения* подразумеваются правила, определяемые для команд INSERT, UPDATE или DELETE.

Правила для изменения применяются системой правил, когда результирующее отношение и тип команды в дереве запроса совпадает с объектом и событием, заданным в команде CREATE RULE. Для такого правила система правил создаёт список деревьев запросов. Изначально этот список пуст.

С правилом может быть связано ноль (ключевое слово NOTHING), одно или несколько действий. Рассмотрим правило с одним действием. Правило может иметь, а может не иметь условия применения, и действует в режиме INSTEAD или ALSO (по умолчанию).

Что такое условие применения правила? Это условие, которое говорит, когда нужно, а когда не нужно применять действия правила. В этом условии можно обращаться к псевдоотношениям NEW и/или OLD, которые представляют целевое отношение (но с особым значением).

Всего есть три варианта формирования деревьев запросов для правила с одним действием. –

Без условия применения в режиме ALSO или INSTEAD – дерево запроса из действия правила с добавленным условием исходного дерева.

– С условием применения в режиме ALSO – дерево запроса из действия правила с условием применения правила и условием, добавленным из исходного дерева.

– С условием применения в режиме INSTEAD – дерево запроса из действия правила с условием применения правила и условием из исходного дерева; также добавляется исходное дерево запроса с условием, обратным условию применения правила

Для правил ALSO в список добавляется исходное дерево запроса без изменений. Так как исходное дерево запроса также добавляют только правила INSTEAD с условиями применения, в итоге для правила с одним действием можно получить только одно или два дерева запросов.

Для правил ON INSERT исходный запрос (если он не перекрывается режимом INSTEAD) выполняется перед действиями, добавленными правилами. Поэтому эти действия могут видеть вставленные строки. Но для правил ON UPDATE и ON DELETE исходный запрос выполняется после действий, добавленных правилами. При таком порядке эти действия будут видеть строки, подлежащие изменению или удалению; иначе бы действия не работали, не найдя строк, соответствующих их условиям применения (эти строки уже будут изменены или удалены).

Деревья запросов, полученные из действий правил, снова попадают в систему перезаписи, где могут примениться дополнительные правила, добавляющие или убирающие деревья запроса. Поэтому действия правила должны выполнять команды другого типа или работать с другим

результирующим отношением, иначе возникнет бесконечная рекурсия. (Система выявляет подобное рекурсивное разворачивание правил и выдаёт ошибку.)

Деревья запросов, заданные для действий в системном каталоге pg_rewrite, представляют собой только шаблоны. Так как они могут обращаться к элементам NEW и OLD в списке отношений, их можно будет использовать только после некоторых подстановок. В случае ссылки на NEW соответствующий элемент ищется в целевом списке исходного запроса. Если он найден, ссылка заменяется выражением этого элемента. В противном случае NEW означает то же самое, что и OLD (для команды UPDATE) или заменяется значением NULL (для команды INSERT). Любые ссылки на OLD заменяются ссылкой на элемент результирующего отношения в списке отношений.

После того как система применит все правила для изменения, она применяет правила представления к полученному дереву (или деревьям) запроса. Представления не могут добавлять новые действия для изменения, поэтому нет необходимости применять такие правила к результату перезаписи представления.

2845. Правила и права

В результате переписывания запросов системой правил PG360 обращение может происходить не к тем таблицам/представлениям, к которым обращался исходный запрос. С правилами для изменения возможна так же и запись в другие таблицы.

Правила перезаписи не имеют отдельного владельца — владельцем правил перезаписи, определённых для отношения (таблицы или представления), автоматически считается владелец этого отношения. Система правил PG360 меняет поведение стандартного механизма управления доступом. К отношениям, используемым вследствие применения правил, проверяется доступ владельца правила, но не пользователя, выполняющего запрос. Это значит, что пользователь должен иметь права, необходимые только для обращения к таблицам/ представлениям, которые он явно упоминает в своих запросах.

Когда требуется, чтобы представление обеспечивало защиту на уровне строк, к нему нужно применить атрибут security_barrier. Это предотвратит утечку содержимого строк из злонамеренно выбранных функций и операторов до того, как строки будут отфильтрованы представлением.

Представления, созданные с атрибутом security_barrier, могут работать гораздо медленнее, чем обычные.

Планировщик запросов имеет больше свободы, работая с функциями, лишёнными побочных эффектов. Такие функции называются герметичными (LEAKPROOF) и включают только простые часто используемые операторы, например, операторы равенства. Планировщик запросов может безопасно вычислять такие функции в любой момент выполнения запроса, так как при вызове их для строк, невидимых пользователю, не просочится никакая информация об этих строках. Более того, функции, которые не принимают аргументы или которым не передаются аргументы из представления с барьером безопасности, можно не помечать как LEAKPROOF, чтобы они вышли наружу, так как они никогда не получат данные из представления. И напротив, функции, которые могут вызвать ошибку в зависимости от значений аргументов (например, в случае переполнения или деления на ноль), герметичными не являются, и могут выдать существенную информацию о невидимых строках, если будут выполнены перед фильтрами строк.

Даже представление, созданное с атрибутом security_barrier, остаётся безопасным только в том смысле, что содержимое невидимых строк не будет передаваться потенциально небезопасным функциям. Но пользователь может собрать некоторые сведения о невидимых данных и другими способами; например, он может проанализировать план запроса, полученный с EXPLAIN, или замерить время выполнения запросов с этим представлением. Злоумышленник может сделать определённые выводы об объёме невидимых данных или даже получить некоторую информацию о распределении данных или наиболее частых значениях (так как всё это отражается в статистике для оптимизатора и, как следствие, влияет на время выполнения плана или даже на выбор плана). Если возможность атаки через скрытые каналы вызывает опасения, вероятно, будет разумным не предоставлять никакой доступ к этим данным.

2846. Правила и статус команд

Сервер PG360 возвращает строку состояния команды, например, INSERT 149592 1, для каждой получаемой команды. Это довольно прозрачно, когда не задействуются правила, но что произойдёт, если правила перезапишут запрос?

Правила влияют на состояния команды следующим образом:

– Если с запросом не связано безусловное правило INSTEAD, то выполняется заданный исходный запрос и его статус выдаётся как обычно. (Но если определены какие-то условные правила INSTEAD, к исходному запросу добавляется условие, обратное их условиям применения. Это может повлиять на число обрабатываемых строк и выводимый статус команды.)

– Если с запросом связано безусловное правило INSTEAD, исходный запрос не выполняется вовсе. В этом случае сервер возвратит статус команды от последнего запроса, вставленного правилом INSTEAD (условным или безусловным), и тип команды исходного запроса (INSERT, UPDATE или DELETE). Если правила не добавили подходящего запроса, в возвращённом статусе команды показывается исходный тип запроса и нули вместо количества строк и OID.

Статус команды во втором случае должно устанавливать нужное правило INSTEAD, необходимо назначить ему имя, стоящее по алфавиту после других активных правил, чтобы это правило применялось последним.

285. Процедурные языки

PG360 позволяет разрабатывать пользовательские функции не только на SQL и C, но и на других языках. Эти языки в целом называются *процедурными языками* (PL, Procedural Language). Если функция написана на процедурном языке, сервер баз данных сам по себе не знает, как интерпретировать её исходный текст. Вместо этого он передаёт эту задачу специальному обработчику, понимающему данный язык. Обработчик может либо выполнить всю работу по разбору, синтаксическому анализу, выполнению кода и т. д., либо действовать как «прослойка» между PG360 и внешним исполнителем языка программирования. Сам обработчик представляет собой функцию на языке C, скомпилированную в виде разделяемого объекта и загружаемую по требованию, как и любая другая функция на C.

Дистрибутив PG360 включает четыре процедурных языка: PL/pgSQL, PL/Tcl, PL/Perl и PL/Python.

2851. Установка процедурных языков

Прежде всего, процедурный язык должен быть «установлен» в каждую базу данных, где он будет использоваться. Но процедурные языки, устанавливаемые в базу данных template1, автоматически становятся доступными во всех впоследствии создаваемых базах, так как их определения в template1 будут скопированы командой CREATE DATABASE. Таким образом, администратор баз данных может выбрать, какие языки будут доступны в определённых базах данных, и при желании сделать некоторые языки доступными по умолчанию.

Для языков, включённых в дистрибутив, необходимо выполнить команду:

CREATE EXTENSION *имя_языка*,

чтобы установить язык в текущую базу данных.

2.8.6. PL/pgSQL – процедурный язык SQL

2861. Обзор

PL/pgSQL это процедурный язык для СУБД PG360. Целью проектирования PL/pgSQL было создание загружаемого процедурного языка, который:

– используется для создания функций, процедур и триггеров;

– добавляет управляющие структуры к языку SQL;

– может выполнять сложные вычисления;

– наследует все пользовательские типы, функции, процедуры и операторы;

– может быть определён как доверенный язык;

– прост в использовании.

Функции PL/pgSQL могут использоваться везде, где допустимы встроенные функции. Например, можно создать функции со сложными вычислениями и условной логикой, а затем использовать их при определении операторов или в индексных выражениях.

Функции на PL/pgSQL могут принимать в качестве аргументов все поддерживаемые сервером скалярные типы данных или массивы и возвращать в качестве результата любой из этих типов. Они могут принимать и возвращать любой именованный составной тип (тип кортежа). Также есть возможность объявить функцию на PL/pgSQL как принимающую record, то есть ей может быть передан любой составной тип, или как возвращающую record, то есть её результатом будет кортеж, столбцы которого определит спецификация вызывающего запроса.

Использование маркера VARIADIC позволяет объявлять функции на PL/pgSQL с переменным числом аргументов.

Функции на PL/pgSQL могут также принимать и возвращать полиморфные типы, вследствие чего фактические типы данных, обрабатываемые функцией, могут меняться от вызова к вызову.

Функции на PL/pgSQL могут возвращать «множества» (или таблицы) любого типа, которые могут быть возвращены в виде одного объекта. Такие функции генерируют вывод, выполняя команду RETURN NEXT для каждого элемента результирующего набора или RETURN QUERY для вывода результата запроса.

При отсутствии возвращаемого значения функция на PL/pgSQL может возвращать void.

Функции на PL/pgSQL можно объявить с выходными параметрами вместо явного задания типа возвращаемого значения. Это не добавляет никаких фундаментальных возможностей языку, но часто бывает удобно, особенно для возвращения нескольких значений. Нотация RETURNS TABLE может использоваться вместо RETURNS SETOF.

2862. Структура PL/pgSQL

Функции, написанные на PL/pgSQL, определяются на сервере командами CREATE FUNCTION. Например:

```
CREATE FUNCTION somefunc(integer, text) RETURNS integer AS 'тело функции' LANGUAGE plpgsql;
```

PL/pgSQL это блочно-структурированный язык. Текст тела функции должен быть *блоком*. Структура блока:

```
[ <<метка>> ] [ DECLARE
объявления ] BEGIN
операторы
END [ метка ];
```

Каждое объявление и каждый оператор в блоке должны завершаться символом «;» (точка с запятой). Блок, вложенный в другой блок, должен иметь точку с запятой после END, как показано выше. Однако финальный END, завершающий тело функции, не требует точки с запятой.

Метка требуется только тогда, когда нужно идентифицировать блок в операторе EXIT, или дополнить имена переменных, объявленных в этом блоке. Если метка указана после END, то она должна совпадать с меткой в начале блока.

Ключевые слова не чувствительны к регистру символов. Как и в обычных SQL-командах, идентификаторы неявно преобразуются к нижнему регистру, если они не взяты в двойные кавычки.

Комментарии в PL/pgSQL коде работают так же, как и в обычном SQL. Двойное тире (--) начинает комментарий, который завершается в конце строки. Блочный комментарий начинается с /* и завершается */. Блочные комментарии могут быть вложенными.

Любой оператор в выполняемой секции блока может быть *вложенным блоком*. Вложенные блоки используются для логической группировки нескольких операторов или локализации области действия переменных для группы операторов. Во время выполнения вложенного блока переменные, объявленные в нём, скрывают переменные внешних блоков с такими же именами. Чтобы получить доступ к внешним переменным, нужно дополнить их имена меткой блока.

2863. Объявления

Все переменные, используемые в блоке, должны быть определены в секции объявления. (За исключением переменной-счётчика цикла FOR, которая объявляется автоматически. Для цикла по диапазону чисел автоматически объявляется целочисленная переменная, а для цикла по результатам курсора – переменная типа record.)

Переменные PL/pgSQL могут иметь любой тип данных SQL, такой как integer, varchar, char.

Общий синтаксис объявления переменной:

```
имя [ CONSTANT ] тип [ COLLATE имя_правила_сортировки ] [ NOT NULL ] [ { DEFAULT | := |= } выражение ];
```

Предложение DEFAULT, если присутствует, задаёт начальное значение, которое присваивается переменной при входе в блок. Если отсутствует, то переменная инициализируется SQL-значением NULL. Указание CONSTANT предотвращает изменение значения переменной после инициализации, таким образом, значение остаётся постоянным в течение всего блока. Параметр COLLATE определяет правило сортировки, которое будет использоваться для этой переменной. Если указано NOT NULL, то попытка присвоить NULL во время выполнения приведёт к ошибке. Все переменные, объявленные как NOT NULL, должны иметь непустые значения по умолчанию. Можно использовать знак равенства (=) вместо совместимого с PL/SQL :=.

Значение по умолчанию вычисляется и присваивается переменной каждый раз при входе в блок (не только при первом вызове функции).

Переданные в функцию параметры именуются идентификаторами $1, $2 и т. д. Дополнительно, для улучшения читаемости, можно объявить псевдонимы для параметров $*n*. Либо псевдоним, либо цифровой идентификатор используются для обозначения параметра.

Создать псевдоним можно двумя способами. Предпочтительный способ это дать имя параметру в команде CREATE FUNCTION. Другой способ это явное объявление псевдонима при помощи синтаксиса:

```
имя ALIAS FOR $n;
```

Когда функция на PL/pgSQL объявляется с выходными параметрами, им выдаются цифровые идентификаторы $*n* и для них можно создавать псевдонимы точно таким же способом, как и для обычных входных параметров. Выходной параметр это фактически переменная, стартующая с NULL и которой присваивается значение во время выполнения функции. Возвращается последнее присвоенное значение.

Выходные параметры используются для возвращения нескольких значений.

Для функции на PL/pgSQL, возвращающей полиморфный тип, создаётся специальный параметр $0. Его тип данных соответствует типу, фактически возвращаемому функцией, который устанавливается на основании фактических типов входных параметров. Это позволяет функции обращаться к фактически возвращаемому типу данных. Параметр $0 инициализируется в NULL и его можно изменять внутри функции. Таким образом, его можно использовать для хранения возвращаемого значения, хотя это необязательно. Параметру $0 можно дать псевдоним.

2864. Выражения

Все выражения, используемые в операторах PL/pgSQL, обрабатываются основным исполнителем SQL-сервера. Например, для вычисления такого выражения:

```
IF выражение THEN ...
```

PL/pgSQL отправит следующий запрос исполнителю SQL:

```
SELECT выражение
```

При формировании команды SELECT все вхождения имён переменных PL/pgSQL заменяются параметрами. Это позволяет один раз подготовить план выполнения команды SELECT и повторно использовать его в последующих вычислениях с различными значениями переменных. Таким образом, при первом использовании выражения, происходит выполнение команды PREPARE.

2865. Основные операторы

В этом и последующих разделах описаны все типы операторов, которые понимает PL/pgSQL. Все, что не признается в качестве одного из этих типов операторов, считается командой SQL и отправляется для исполнения в основную машину базы данных.

28651. Присваивания

Присваивание значения переменной PL/pgSQL записывается в виде:

```
переменная { := | = } выражение;
```

Выражение в таком операторе вычисляется с помощью SQL-команды SELECT, посылаемой в основную машину базы данных. Выражение должно получить одно значение (возможно, значение строки, если это переменная-кортеж или переменная типа record). Целевая переменная может быть простой переменной (возможно, дополненной именем блока), полем кортежа или записи; или элементом массива, который является простой переменной или полем. Для присваивания можно использовать знак равенства (=) вместо совместимого с PL/SQL :=.

Если тип данных результата выражения не соответствует типу данных переменной, это значение будет преобразовано к нужному типу с использованием приведения присваивания. В случае отсутствия приведения присваивания для этой пары типов, интерпретатор PL/pgSQL попытается преобразовать значение результата через текстовый формат, то есть применив функцию вывода типа результата, а за ней функцию ввода типа переменной. При этом функция ввода может выдавать ошибки времени выполнения, если не воспримет строковое представление значения результата.

28652. Выполнение команды, не возвращающей результат

В функции на PL/pgSQL можно выполнить любую команду SQL, не возвращающую строк, просто написав эту команду (например, INSERT без предложения RETURNING).

Имя любой переменной PL/pgSQL в тексте команды рассматривается как параметр, а затем текущее значение переменной подставляется в качестве значения параметра во время выполнения. Это в точности совпадает с описанной ранее обработкой для выражений.

При выполнении SQL-команды таким образом, PL/pgSQL может кешировать и повторно использовать план выполнения команды.

Иногда необходимо вычислить значение выражения или запроса SELECT, но отказаться от результата. Для этого в PL/pgSQL, используется оператор PERFORM:

```
PERFORM запрос;
```

Эта команда выполняет *запрос* и отбрасывает результат. *Запросы* пишутся таким же образом, как и в команде SQL SELECT, но ключевое слово SELECT заменяется на PERFORM. Для запросов WITH после PERFORM нужно поместить запрос в скобки. (В этом случае запрос может вернуть только одну строку.) Переменные PL/pgSQL будут подставлены в запрос так же, как и в команду, не возвращающую результат, план запроса также кешируется. Кроме того, специальная переменная FOUND устанавливается в истину, если запрос возвращает, по крайней мере, одну строку, или ложь, если не возвращает ни одной строки.

28653. Выполнение запроса, возвращающего одну строку

Результат SQL-команды, возвращающей одну строку (возможно из нескольких столбцов), может быть присвоен переменной типа record, переменной-кортежу или списку скалярных переменных. Для этого нужно к основной команде SQL добавить предложение INTO. Например:

```
SELECT выражения_select INTO [STRICT] цель FROM ...; INSERT ... RETURNING выражения INTO [STRICT] цель; UPDATE ... RETURNING выражения INTO [STRICT] цель; DELETE ... RETURNING выражения INTO [STRICT] цель;
```

где *цель* может быть переменной типа record, переменной-кортежем или разделённым запятыми списком скалярных переменных, полей записи/строки. Переменные PL/pgSQL подставляются в оставшуюся часть запроса, план выполнения кешируется, так же, как было описано выше для команд, не возвращающих строки. Это работает для команд SELECT, INSERT/UPDATE/DELETE с предложением RETURNING и служебных команд, возвращающих результат в виде набора строк (таких как EXPLAIN). За исключением предложения INTO, это те же SQL-команды, как их можно написать вне PL/pgSQL.

Если результат запроса присваивается кортежу или списку переменных, то они должны в точности соответствовать по количеству и типам данных столбцам результата, иначе произойдёт ошибка во время выполнения. Если используется переменная типа record, то она автоматически приводится к типу строки результата запроса.

Предложение INTO может появиться практически в любом месте SQL-команды. Обычно его записывают непосредственно перед или сразу после списка *выражения_select* в SELECT или в конце команды для команд других типов. Рекомендуется следовать этому соглашению на случай, если правила разбора PL/pgSQL ужесточатся в будущих версиях.

Если указание STRICT отсутствует в предложении INTO, то *цели* присваивается первая строка, возвращённая запросом; или NULL, если запрос не вернул строк. (Заметим, что понятие «первая строка» определяется неоднозначно без ORDER BY.) Все остальные строки результата после первой отбрасываются. Можно проверить специальную переменную FOUND, чтобы определить, была ли возвращена запись:

```
SELECT * INTO myrec FROM emp WHERE empname = myname; IF NOT FOUND THEN RAISE EXCEPTION 'Сотрудник % не найден', myname; END IF;
```

Если добавлено указание STRICT, то запрос должен вернуть ровно одну строку или произойдёт ошибка во время выполнения: либо NO_DATA_FOUND (нет строк), либо TOO_MANY_ROWS (более одной строки). Можно использовать секцию исключений в блоке для обработки ошибок, например:

```
BEGIN
SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
EXCEPTION
WHEN NO_DATA_FOUND THEN
RAISE EXCEPTION 'Сотрудник % не найден', myname; WHEN TOO_MANY_ROWS THEN
RAISE EXCEPTION 'Сотрудник % уже существует', myname;
END;
```

После успешного выполнения команды с указанием STRICT, значение переменной FOUND всегда устанавливается в истину.

Для INSERT/UPDATE/DELETE с RETURNING, PL/pgSQL возвращает ошибку, если выбрано более одной строки, даже в том случае, когда указание STRICT отсутствует. Так происходит потому, что у этих команд нет возможности, типа ORDER BY, указать какая из задействованных строк должна быть возвращена.

Если для функции включён режим print_strict_params, то при возникновении ошибки, связанной с нарушением условия STRICT, в детальную (DETAIL) часть сообщения об ошибке будет включена информация о параметрах, переданных запросу. Изменить значение print_strict_params можно установкой параметра plpgsql.print_strict_params. Но это повлияет только на функции, скомпилированные после изменения. Для конкретной функции можно использовать указание компилятора, например:

```
CREATE FUNCTION get_userid(username text) RETURNS int AS $$
#print_strict_params on DECLARE
userid int;
BEGIN
SELECT users.userid INTO STRICT userid
FROM users WHERE users.username = get_userid.username; RETURN
userid;
END;
$$ LANGUAGE plpgsql;
```

В случае сбоя будет сформировано примерно такое сообщение об ошибке:

```
ERROR: query returned no rows
DETAIL: parameters: $1 = 'nosuchuser'
CONTEXT: PL/pgSQL function get_userid(text) line 6 at SQL
statement
```

28.6.5. Выполнение динамически формируемых команд

Часто требуется динамически формировать команды внутри функций на PL/pgSQL, то есть такие команды, в которых при каждом выполнении могут использоваться разные таблицы или типы данных. Обычно PL/pgSQL кеширует планы выполнения, но в случае с динамическими командами это не будет работать. Для исполнения динамических команд предусмотрен оператор EXECUTE:

```
EXECUTE строка-команды [ INTO [STRICT] цель ] [ USING выражение
[, ... ] ];
```

где *строка-команды* это выражение, формирующее строку (типа text) с текстом команды, которую нужно выполнить. Необязательная *цель* – это переменная-запись, переменная-кортеж или разделённый запятыми список простых переменных и полей записи/кортежа, куда будут помещены результаты команды. Необязательные выражения в USING формируют значения, которые будут вставлены в команду.

В сформированном тексте команды замена имён переменных PL/pgSQL на их значения проводиться не будет. Все необходимые значения переменных должны быть вставлены в командную строку при её построении, либо нужно использовать параметры, как описано ниже.

Также, нет никакого плана кеширования для команд, выполняемых с помощью EXECUTE. Вместо этого план создаётся каждый раз при выполнении. Таким образом, строка команды может

динамически создаваться внутри функции для выполнения действий с различными таблицами и столбцами.

Предложение INTO указывает, куда должны быть помещены результаты SQL-команды, возвращающей строки. Если передаётся кортеж или список переменных, то они должны в точности соответствовать структуре результата запроса (когда используется переменная типа record, она автоматически приводится к типу строки результата запроса). Если возвращается несколько строк, то только первая будет присвоена переменной(ым) в INTO. Если не возвращается ни одной строки, то присваивается NULL. Без предложения INTO результаты запроса отбрасываются.

С указанием STRICT запрос должен вернуть ровно одну строку, иначе выдаётся сообщение об ошибке.

В тексте команды можно использовать значения параметров, ссылки на параметры обозначаются как $1, $2 и т. д. Эти символы указывают на значения, находящиеся в предложении USING. Такой метод зачастую предпочтительнее, чем вставка значений в команду в виде текста: он позволяет исключить во время выполнения дополнительные расходы на преобразования значений в текст и обратно, и не открывает возможности для SQL-инъекций, не требуя применять экранирование или кавычки для спецсимволов. Пример:

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2' INTO c
USING checked_user, checked_date;
```

Символы параметров можно использовать только вместо значений данных. Если же требуется динамически формировать имена таблиц или столбцов, их необходимо вставлять в виде текста.

Ещё одно ограничение состоит в том, что символы параметров могут использоваться только в командах SELECT, INSERT, UPDATE и DELETE. В операторы других типов (обычно называемые служебными) значения нужно вставлять в текстовом виде, даже если это просто значения данных.

Команда EXECUTE с неизменяемым текстом и параметрами USING (как в первом примере выше), функционально эквивалентна команде, записанной напрямую в PL/pgSQL, в которой переменные PL/pgSQL автоматически заменяются значениями. Важное отличие в том, что EXECUTE при каждом исполнении заново строит план команды с учётом текущих значений параметров, тогда как PL/ pgSQL строит общий план выполнения и кеширует его при повторном использовании. В тех случаях, когда наилучший план выполнения сильно зависит от значений параметров, можно использовать EXECUTE для гарантии того, что не будет выбран общий план.

В настоящее время команда SELECT INTO не поддерживается в EXECUTE, вместо этого нужно выполнять обычный SELECT и указать INTO для самой команды EXECUTE.

28655. Статус выполнения команды

Определить результат команды можно несколькими способами. Во-первых, можно воспользоваться командой GET DIAGNOSTICS, имеющей форму:

```
GET [ CURRENT ] DIAGNOSTICS переменная { = | := }
элемент [ , ... ];
```

Эта команда позволяет получить системные индикаторы состояния.

Каждый *элемент* представляется ключевым словом, указывающим, какое значение состояния нужно присвоить заданной *переменной* (она должна иметь подходящий тип данных, чтобы принять его). Доступные в настоящее время элементы состояния:

– bigint ROW_COUNT – число строк, обработанных последней командой SQL;

– text PG_CONTEXT – строки текста, описывающие текущий стек вызовов.

Вместо принятого в стандарте SQL присваивания (=) можно применять присваивание с двоеточием (:=). Например:

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

Второй способ определения статуса выполнения команды заключается в проверке значения специальной переменной FOUND, имеющей тип boolean. При вызове функции на PL/pgSQL, переменная FOUND инициализируется в ложь. Далее, значение переменной изменяется следующими операторами:

– SELECT INTO записывает в FOUND true, если строка присвоена, или false, если строки не были получены.

– PERFORM записывает в FOUND true, если строки выбраны (и отброшены) или false, если строки не выбраны.

– UPDATE, INSERT и DELETE записывают в FOUND true, если при их выполнении была задействована хотя бы одна строка, или false, если ни одна строка не была задействована.

– FETCH записывают в FOUND true, если команда вернула строку, или false, если строка не выбрана.

– MOVE записывают в FOUND true при успешном перемещении курсора, в противном случае – false.

– FOR, как и FOREACH, записывает в FOUND true, если была произведена хотя бы одна итерация цикла, в противном случае – false. При этом значение FOUND будет установлено только после выхода из цикла. Пока цикл выполняется, оператор цикла не изменяет значение переменной. Но другие операторы внутри цикла могут менять значение FOUND.

– RETURN QUERY и RETURN QUERY EXECUTE записывают в FOUND true, если запрос вернул хотя бы одну строку, или false, если строки не выбраны.

Другие операторы PL/pgSQL не меняют значение FOUND. EXECUTE изменяет вывод GET DIAGNOSTICS, но не меняет FOUND.

FOUND является локальной переменной в каждой функции PL/pgSQL и любые её изменения, влияют только на текущую функцию.

28656. Не делать ничего

Иногда бывает полезен оператор, который не делает ничего. Например, он может показывать, что одна из ветвей if/then/else сознательно оставлена пустой. Для этих целей используется NULL:

```
NULL;
```

2866. Управляющие структуры

С помощью управляющих структур PL/pgSQL можно гибко и эффективно манипулировать данными PG360.

Команды, возвращающие значения из функции

Две команды позволяют вернуть данные из функции: RETURN и RETURN NEXT.

```
RETURN
RETURN выражение;
```

RETURN с последующим выражением прекращает выполнение функции и возвращает значение выражения в вызывающую программу. Эта форма используется для функций PL/pgSQL, которые не возвращают набор строк.

В функции, возвращающей скалярный тип, результирующее выражение автоматически приводится к типу возвращаемого значения. Однако если возвращаемый тип —составной (строка), возвращаемое выражение должно в точности содержать требуемый набор столбцов. При этом может потребоваться явное приведение типов.

Для функции с выходными параметрами необходимо используйть RETURN без выражения. Будут возвращены текущие значения выходных параметров.

Для функции, возвращающей void, RETURN можно использовать в любом месте, но без выражения после RETURN.

Возвращаемое значение функции не может остаться не определённым. Если достигнут конец блока верхнего уровня, а оператор RETURN так и не встретился, происходит ошибка времени выполнения. Это не касается функций с выходными параметрами и функций, возвращающих void. Для них оператор RETURN выполняется автоматически по окончании блока верхнего уровня.

```
RETURN NEXT и RETURN QUERY
RETURN NEXT выражение; RETURN QUERY запрос;
RETURN QUERY EXECUTE строка-команды [USING выражение [, ...]];
```

Для функций на PL/pgSQL, возвращающих SETOF *некий_тип*, нужно действовать несколько по-иному. Отдельные элементы возвращаемого значения формируются командами RETURN NEXT или RETURN QUERY, а финальная команда RETURN без аргументов завершает выполнение функции. RETURN NEXT используется как со скалярными, так и с составными типами данных. Для составного типа результат функции возвращается в виде таблицы. RETURN QUERY добавляет результат выполнения запроса к результату функции. RETURN NEXT и RETURN QUERY можно свободно смешивать в теле функции, в этом случае их результаты будут объединены.

RETURN NEXT и RETURN QUERY не выполняют возврат из функции. Они просто добавляют строки в результирующее множество. Затем выполнение продолжается со следующего оператора в функции. Успешное выполнение RETURN NEXT и RETURN QUERY формирует множество строк результата. Для выхода из функции используется RETURN, обязательно без аргументов (или можно просто дождаться окончания выполнения функции).

RETURN QUERY имеет разновидность RETURN QUERY EXECUTE, предназначенную для динамического выполнения запроса. В текст запроса можно добавить параметры, используя USING, так же как и с командой EXECUTE.

Для функции с выходными параметрами необходимо использовать RETURN NEXT без аргументов. При каждом исполнении RETURN NEXT текущие значения выходных параметров сохраняются для последующего возврата в качестве строки результата. Если функция с выходными параметрами должна возвращать множество значений, то при объявлении нужно

указывать RETURNS SETOF. При этом если выходных параметров несколько, то используется RETURNS SETOF record, а если только один с типом *некий_тип*, то RETURNS SETOF *некий_тип*.

28661. Завершение процедуры

Процедура не возвращает никакого значения, поэтому она может завершаться без оператора RETURN. Если необходмо досрочно завершить выполнение кода оператором RETURN, необходимо написать RETURN без возвращаемого выражения.

Если у процедуры есть выходные параметры, конечные значения соответствующих им переменных будут выданы вызывающему коду.

28662. Вызов процедуры

Функция, процедура или блок DO в PL/pgSQL может вызвать процедуру, используя оператор CALL. Выходные параметры при этом обрабатываются не так, как это делает CALL в обычном SQL. Каждому параметру INOUT для процедуры должна соответствовать переменная в операторе CALL, и этой переменной по завершении процедуры будет присвоено возвращаемое процедурой значение. Например:

```
CREATE PROCEDURE triple(INOUT x int) LANGUAGE plpgsql
AS $$ BEGIN
x := x * 3;
END;
$$;

DO $$
DECLARE myvar int := 5; BEGIN
CALL triple(myvar);
RAISE NOTICE 'myvar = %', myvar;  -- выводится 15 END;
$$;
```

28663. Условные операторы

Операторы IF и CASE позволяют выполнять команды в зависимости от определённых условий. PL/ pgSQL поддерживает три формы IF:

```
IF ... THEN ... END IF
IF ... THEN ... ELSE ... END IF
IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF
```

и две формы CASE:

```
CASE ... WHEN ... THEN ... ELSE ... END CASE
CASE WHEN ... THEN ... ELSE ... END CASE
```

```
IF-THEN
IF логическое-выражение THEN
операторы
END IF;
```

IF-THEN это простейшая форма IF. Операторы между THEN и END IF выполняются, если условие (*логическое-выражение*) истинно. В противном случае они опускаются.

Пример:

```
IF v_user_id <> 0 THEN
UPDATE users SET email = v_email WHERE user_id = v_user_id; END
IF;
```

```
IF-THEN-ELSE
IF логическое-выражение THEN
операторы
ELSE
операторы
END IF;
```

IF-THEN-ELSE добавляет к IF-THEN возможность указать альтернативный набор операторов, которые будут выполнены, если условие не истинно (в том числе, если условие NULL).

В некоторых случаях двух альтернатив недостаточно. IF-THEN-ELSIF обеспечивает удобный способ проверки нескольких вариантов по очереди. Условия в IF последовательно проверяются до тех пор, пока не будет найдено первое истинное. После этого операторы, относящиеся к этому условию, выполняются, и управление переходит к следующей после END IF команде. (Все последующие условия не проверяются.) Если ни одно из условий IF не является истинным, то выполняется блок ELSE (если присутствует).

Пример:

```
IF number = 0 THEN result := 'zero';
ELSIF number > 0 THEN result := 'positive';
ELSIF number < 0 THEN result := 'negative';
ELSE
```

– остаётся только один вариант: number имеет значение NULL result := 'NULL';

```
END IF;
```

Вместо ключевого слова ELSIF можно использовать ELSEIF.

Другой вариант сделать то же самое, это использование вложенных операторов IF-THEN-ELSE, как в следующем примере:

```
IF demo_row.sex = 'm' THEN pretty_sex := 'man';
ELSE
IF demo_row.sex = 'f' THEN pretty_sex := 'woman';
END IF;
END IF;
```

Однако это требует написания соответствующих END IF для каждого IF, что при наличии нескольких альтернатив делает код более громоздким, чем использование ELSIF.

Оператор CASE:

```
CASE выражение-поиска
WHEN выражение [, выражение [...]] THEN
операторы
[WHEN выражение [, выражение [...]] THEN операторы ...] [ELSE
операторы]
END CASE;
```

Простая форма CASE реализует условное выполнение на основе сравнения операндов. *Выражение– поиска* вычисляется (один раз) и последовательно сравнивается с каждым *выражением* в условиях WHEN. Если совпадение найдено, то выполняются соответствующие *операторы* и управление переходит к следующей после END CASE команде. (Все последующие выражения WHEN не проверяются.) Если совпадение не было найдено, то выполняются *операторы* в ELSE. Но если ELSE нет, то вызывается исключение CASE_NOT_FOUND.

CASE с перебором условий:

```
CASE
WHEN логическое-выражение THEN
операторы
[WHEN логическое-выражение THEN операторы ...] [ELSE операторы]
END CASE;
```

Эта форма CASE реализует условное выполнение, основываясь на истинности логических условий. Каждое *логическое-выражение* в предложении WHEN вычисляется по порядку до тех пор, пока не будет найдено истинное. Затем выполняются соответствующие *операторы* и управление переходит к следующей после END CASE команде. (Все последующие выражения WHEN не проверяются.) Если ни одно из условий не окажется истинным, то выполняются *операторы* в ELSE. Но если ELSE нет, то вызывается исключение CASE_NOT_FOUND.

Эта форма CASE полностью эквивалента IF-THEN-ELSIF, за исключением того, что при невыполнении всех условий и отсутствии ELSE, IF-THEN-ELSIF ничего не делает, а CASE вызывает ошибку.

28664. Простые циклы

Операторы LOOP, EXIT, CONTINUE, WHILE, FOR и FOREACH позволяют повторить серию команд в функции на PL/pgSQL.

```
LOOP
[<<метка>>] LOOP
операторы
END LOOP [ метка ];
```

LOOP организует безусловный цикл, который повторяется до бесконечности, пока не будет прекращён операторами EXIT или RETURN. Для вложенных циклов можно использовать *метку* в операторах EXIT и CONTINUE, чтобы указать, к какому циклу эти операторы относятся.

EXIT:

```
EXIT [ метка ] [WHEN логическое-выражение];
```

Если *метка* не указана, то завершается самый внутренний цикл, далее выполняется оператор, следующий за END LOOP. Если *метка* указана, то она должна относиться к текущему или внешнему циклу, или это может быть метка блока. При этом в именованном цикле/блоке выполнение прекращается, а управление переходит к следующему оператору после соответствующего END.

При наличии WHEN цикл прекращается, только если *логическое-выражение* истинно. В противном случае управление переходит к оператору, следующему за EXIT.

EXIT можно использовать со всеми типами циклов, не только с безусловным.

Когда EXIT используется для выхода из блока, управление переходит к следующему оператору после окончания блока. Для выхода из блока нужно обязательно указывать *метку*. EXIT без *метки* не позволяет прекратить работу блока.

CONTINUE:

```
CONTINUE [ метка ] [WHEN логическое-выражение];
```

Если *метка* не указана, то начинается следующая итерация самого внутреннего цикла. То есть все оставшиеся в цикле операторы пропускаются, и управление переходит к управляющему выражению цикла (если есть) для определения, нужна ли ещё одна итерация цикла. Если *метка* присутствует, то она указывает на метку цикла, выполнение которого будет продолжено.

При наличии WHEN следующая итерация цикла начинается только тогда, когда *логическое–выражение* истинно. В противном случае управление переходит к оператору, следующему за CONTINUE.

CONTINUE можно использовать со всеми типами циклов, не только с безусловным.

WHILE:

```
[<<метка>>]
WHILE логическое-выражение LOOP
операторы
END LOOP [ метка ];
```

WHILE выполняет серию команд до тех пор, пока истинно *логическое-выражение*. Выражение проверяется непосредственно перед каждым входом в тело цикла.

FOR (целочисленный вариант):

```
[<<метка>>]
FOR имя IN [REVERSE] выражение .. выражение [BY выражение] LOOP
операторы
END LOOP [ метка ];
```

В этой форме цикла FOR итерации выполняются по диапазону целых чисел. Переменная *имя* автоматически определяется с типом integer и существует только внутри цикла (если уже существует переменная с таким именем, то внутри цикла она будет игнорироваться). Выражения для нижней и верхней границы диапазона чисел вычисляются один раз при входе в цикл. Если не указано BY, то шаг итерации 1, в противном случае используется значение в BY, которое вычисляется, опять же, один раз при входе в цикл. Если указано REVERSE, то после каждой итерации величина шага вычитается, а не добавляется.

Если нижняя граница цикла больше верхней границы (или меньше, в случае REVERSE), то тело цикла не выполняется вообще. При этом ошибка не возникает.

Если с циклом FOR связана *метка*, к целочисленной переменной цикла можно обращаться по имени, указывая эту *метку*.

28665. Цикл по результатам запроса

Другой вариант FOR позволяет организовать цикл по результатам запроса. Синтаксис:

```
[ <<метка>> ]
FOR цель IN запрос LOOP
операторы
END LOOP [ метка ];
```

Переменная *цель* может быть переменной-кортежем, переменной типа record или разделённым запятыми списком скалярных переменных. Переменной *цель* последовательно присваиваются строки результата запроса, и для каждой строки выполняется тело цикла.

В качестве *запроса* в этом типе оператора FOR может задаваться любая команда SQL, возвращающая строки. Чаще всего это SELECT, но также можно использовать и INSERT, UPDATE или DELETE с предложением RETURNING. Кроме того, возможно применение и некоторых служебных команд, например EXPLAIN.

Для переменных PL/pgSQL в тексте запроса выполняется подстановка значений, план запроса кешируется для возможного повторного использования.

Ещё одна разновидность этого типа цикла FOR-IN-EXECUTE:

```
[ <<метка>> ]
FOR цель IN EXECUTE выражение_проверки [ USING выражение [, ... ] ] LOOP
операторы

END LOOP [ метка ];
```

Она похожа на предыдущую форму, за исключением того, что текст запроса указывается в виде строкового выражения. Текст запроса формируется и для него строится план выполнения при каждом входе в цикл. Это даёт программисту выбор между скоростью предварительно разобранного запроса и гибкостью динамического запроса, так же, как и в случае с обычным оператором EXECUTE. Как и в EXECUTE, значения параметров могут быть добавлены в команду с использованием USING.

Ещё один способ организовать цикл по результатам запроса это объявить курсор.

28666. Цикл по элементам массива

Цикл FOREACH очень похож на FOR. Отличие в том, что вместо перебора строк SQL-запроса происходит перебор элементов массива. (В целом, FOREACH предназначен для перебора выражений составного типа. Варианты реализации цикла для работы с прочими составными выражениями помимо массивов могут быть добавлены в будущем.) Синтаксис цикла FOREACH:

```
[ <<метка>> ]
FOREACH цель [ SLICE число ] IN ARRAY выражение LOOP
операторы
END LOOP [ метка ];
```

Без указания SLICE, или если SLICE равен 0, цикл выполняется по всем элементам массива, полученного из *выражения*. Переменной *цель* последовательно присваивается каждый элемент массива и для него выполняется тело цикла.

Обход элементов проводится в том порядке, в котором они сохранялись, независимо от размерности массива. Как правило, *цель* это одиночная переменная, но может быть и списком

переменных, когда элементы массива имеют составной тип (записи). В этом случае переменным присваиваются значения из последовательных столбцов составного элемента массива.

При положительном значении SLICE FOREACH выполняет итерации по срезам массива, а не по отдельным элементам. Значение SLICE должно быть целым числом, не превышающим размерности массива. Переменная *цель* должна быть массивом, который получает последовательные срезы исходного массива, где размерность каждого среза задаётся значением SLICE.

28667. Обработка ошибок

По умолчанию любая возникающая ошибка прерывает выполнение функции на PL/pgSQL и транзакцию, в которая она выполняется. Использование в блоке секции EXCEPTION позволяет перехватывать и обрабатывать ошибки. Синтаксис секции EXCEPTION расширяет синтаксис обычного блока:

```
[ <<метка>> ] [ DECLARE
объявления ] BEGIN
операторы
EXCEPTION
WHEN условие [ OR условие ... ] THEN
операторы_обработчика
[ WHEN условие [ OR условие ... ] THEN
операторы_обработчика
... ]
END;
```

Если ошибок не было, то выполняются все *операторы* блока и управление переходит к следующему оператору после END. Но если при выполнении *оператора* происходит ошибка, то дальнейшая обработка прекращается и управление переходит к списку исключений в секции EXCEPTION. В этом списке ищется первое исключение, условие которого соответствует ошибке. Если исключение найдено, то выполняются соответствующие *операторы_обработчика* и управление переходит к следующему оператору после END. Если исключение не найдено, то ошибка передаётся наружу, как будто секции EXCEPTION не было. При этом ошибку можно перехватить в секции EXCEPTION внешнего блока. Если ошибка так и не была перехвачена, то обработка функции прекращается.

В качестве *условия* может задаваться одно из имён, перечисленных в Приложении А. Если задаётся имя категории, ему соответствуют все ошибки в данной категории. Специальному имени условия OTHERS (другие) соответствуют все типы ошибок, кроме QUERY_CANCELED и ASSERT_FAILURE. (И эти два типа ошибок можно перехватить по имени, но часто это неразумно.) Имена условий воспринимаются без учёта регистра. Условие ошибки также можно задать кодом SQLSTATE.

Если при выполнении *операторов_обработчика* возникнет новая ошибка, то она не может быть перехвачена в этой секции EXCEPTION. Ошибка передаётся наружу и её можно перехватить в секции EXCEPTION внешнего блока.

При выполнении команд в секции EXCEPTION локальные переменные функции на PL/pgSQL сохраняют те значения, которые были на момент возникновения ошибки. Однако все изменения в базе данных, выполненные в блоке, будут отменены.

28668. Получение информации об ошибке

При обработке исключений часто бывает необходимым получить детальную информацию о произошедшей ошибке. Для этого в PL/pgSQL есть два способа: использование специальных переменных и команда GET STACKED DIAGNOSTICS.

Внутри секции EXCEPTION специальная переменная SQLSTATE содержит код ошибки, для которой было вызвано исключение. Специальная переменная SQLERRM содержит сообщение об ошибке, связанное с исключением. Эти переменные являются неопределёнными вне секции EXCEPTION.

Также в обработчике исключения можно получить информацию о текущем исключении командой:

```
GET STACKED DIAGNOSTICS, которая имеет вид:
GET STACKED DIAGNOSTICS переменная { = | := } элемент [ , ... ];
```

Каждый *элемент* представляется ключевым словом, указывающим, какое значение состояния нужно присвоить заданной *переменной* (она должна иметь подходящий тип данных, чтобы принять его).

Доступные элементы состояния:

text RETURNED_SQLSTATE – код исключения, возвращаемый SQLSTATE;

text COLUMN_NAME – имя столбца, относящегося к исключению;

text CONSTRAINT_NAME – имя ограничения целостности, относящегося к исключению;

text PG_DATATYPE_NAME – имя типа данных, относящегося к исключению;

text MESSAGE_TEXT – текст основного сообщения исключения;

text TABLE_NAME – имя таблицы, относящейся к исключению;

text SCHEMA_NAME – имя схемы, относящейся к исключению;

text PG_EXCEPTION_DETAIL – текст детального сообщения исключения (если есть);

text PG_EXCEPTION_HINT – текст подсказки к исключению (если есть);

text PG_EXCEPTION_CONTEXT – строки текста, описывающие стек вызовов в момент исключения.

Если исключение не устанавливает значение для идентификатора, то возвращается пустая строка.

2867. Курсоры

Вместо того чтобы сразу выполнять весь запрос, есть возможность настроить курсор, инкапсулирующий запрос, и затем получать результат запроса по нескольку строк за раз. Одна из причин так делать заключается в том, чтобы избежать переполнения памяти, когда результат содержит большое количество строк. (Пользователям PL/pgSQL не нужно об этом беспокоиться, так как циклы FOR автоматически используют курсоры, чтобы избежать проблем с памятью.) Более интересным вариантом использования является возврат из функции ссылки на курсор, что

позволяет вызывающему получать строки запроса. Это эффективный способ получать большие наборы строк из функций.

28.6.7.1. Объявление курсорных переменных

Доступ к курсорам в PL/pgSQL осуществляется через курсорные переменные, которые всегда имеют специальный тип данных refcursor. Один из способов создать курсорную переменную, просто объявить её как переменную типа refcursor. Другой способ заключается в использовании синтаксиса объявления курсора, который в общем виде выглядит следующим образом:

имя [[NO] SCROLL] CURSOR [(*аргументы*)] FOR *запрос*;

С указанием SCROLL курсор можно будет прокручивать назад. При NO SCROLL прокрутка назад не разрешается. Если ничего не указано, то возможность прокрутки назад зависит от запроса. Если указаны *аргументы*, то они должны представлять собой пары *имя тип_данных*, разделённые через запятую. Эти пары определяют имена, которые будут заменены значениями параметров в данном запросе. Фактические значения для замены этих имён появятся позже, при открытии курсора.

Реализация SCROLL рассчитывает на то, что в запросе курсора используется FOR UPDATE/SHARE. Кроме того, с запросом, включающим изменчивые функции, лучше всего использовать NO SCROLL. Реализация SCROLL предполагает, что повторное чтение вывода запроса даст согласованные результаты, чего нельзя гарантировать при использовании изменчивой функции.

28.6.7.2. Открытие курсора

Прежде чем получать строки из курсора, его нужно открыть. (Это эквивалентно действию SQL-команды DECLARE CURSOR.) В PL/pgSQL есть три формы оператора OPEN, две из которых используются для несвязанных курсорных переменных, а третья для связанных:

– OPEN FOR *запрос*

```
OPEN несвязанная_переменная_курсора [[NO] SCROLL] FOR запрос;
```

Курсорная переменная открывается и получает конкретный запрос для выполнения. Курсор не может уже быть открытым, а курсорная переменная обязана быть несвязанной (то есть просто переменной типа refcursor). Запрос должен быть командой SELECT или любой другой, которая возвращает строки (к примеру EXPLAIN). Запрос обрабатывается так же, как и другие команды SQL в PL/pgSQL: имена переменных PL/pgSQL заменяются на значения, план запроса кешируется для повторного использования. Подстановка значений переменных PL/pgSQL проводится при открытии курсора командой OPEN, последующие изменения значений переменных не влияют на работу курсора. SCROLL и NO SCROLL имеют тот же смысл, что и для связанного курсора.

– OPEN FOR EXECUTE

```
OPEN несвязанная_переменная_курсора [[NO] SCROLL] FOR EXECUTE
строка_запроса [USING выражение [, ...]];
```

Переменная курсора открывается и получает конкретный запрос для выполнения. Курсор не может быть уже открыт и он должен быть объявлен как несвязанная переменная курсора (то есть, как просто переменная refcursor).

– Открытие связанного курсора

```
OPEN связанная_переменная_курсора [( [имя_аргумента :=] значение_аргумента [, ...] )];
```

Эта форма OPEN используется для открытия курсорной переменной, которая была связана с запросом при объявлении. Курсор не может уже быть открытым. Список фактических значений аргументов должен присутствовать только в том случае, если курсор объявлялся с параметрами. Эти значения будут подставлены в запрос.

План запроса для связанного курсора всегда считается кешируемым. В этом случае нет эквивалента EXECUTE. SCROLL и NO SCROLL не могут быть указаны в этой форме OPEN, возможность прокрутки назад была определена при объявлении курсора.

При передаче значений аргументов можно использовать позиционную или именную нотацию. В позиционной нотации все аргументы указываются по порядку. В именной нотации имя каждого аргумента отделяется от выражения аргумента с помощью :=. Также разрешается смешивать позиционную и именную нотации.

28.6.3. Использование курсоров

После того как курсор будет открыт, с ним можно работать при помощи описанных здесь операторов.

Работать с курсором необязательно в той же функции, где он был открыт. Из функции можно вернуть значение с типом refcursor, что позволит вызывающему продолжить работу с курсором. (Внутри refcursor представляет собой обычное строковое имя так называемого портала, содержащего активный запрос курсора. Это имя можно передавать, присваивать другим переменным с типом refcursor и так далее, при этом портал не нарушается.)

Все порталы неявно закрываются в конце транзакции, поэтому значение refcursor можно использовать для ссылки на открытый курсор только до конца транзакции.

FETCH

```
FETCH [направление { FROM | IN }] курсор INTO цель;
```

FETCH извлекает следующую строку из курсора в *цель*. В качестве *цели* может быть переменная– кортеж, переменная типа record или разделённый запятыми список простых переменных, как и в SELECT INTO. Если следующей строки нет, *цели* присваивается NULL. Как и в SELECT INTO, проверить, была ли получена запись, можно при помощи специальной переменной FOUND.

Здесь *направление* может быть любым допустимым в SQL-команде FETCH вариантом, кроме тех, что извлекают более одной строки. А именно: NEXT, PRIOR, FIRST, LAST, ABSOLUTE *число*, RELATIVE *число*, FORWARD или BACKWARD. Без указания *направления* подразумевается вариант NEXT. Везде, где используется *число*, оно может определяться любым целочисленным выражением (в отличие от SQL-команды FETCH, допускающей только целочисленные константы). Значения *направления*, которые требуют перемещения назад, приведут к ошибке, если курсор не был объявлен или открыт с указанием SCROLL.

курсор это переменная с типом refcursor, которая ссылается на открытый портал курсора.

MOVE

```
MOVE [направление { FROM | IN }] курсор;
```

MOVE перемещает курсор без извлечения данных. MOVE работает точно так же как и FETCH, но при этом только перемещает курсор и не извлекает строку, к которой переместился. Как и в SELECT INTO, проверить успешность перемещения можно с помощью специальной переменной FOUND.

UPDATE/DELETE WHERE CURRENT OF

```
UPDATE таблица SET ... WHERE CURRENT OF курсор; DELETE FROM таблица WHERE CURRENT OF курсор;
```

Когда курсор позиционирован на строку таблицы, эту строку можно изменить или удалить при помощи курсора. Есть ограничения на то, каким может быть запрос курсора (в частности, не должно быть группировок), и крайне желательно использовать указание FOR UPDATE.

CLOSE

```
CLOSE курсор;
```

CLOSE закрывает связанный с курсором портал. Используется для того, чтобы освободить ресурсы раньше, чем закончится транзакция, или чтобы освободить курсорную переменную для повторного открытия.

28.6.7.4. Возврат курсора из функции

Курсоры можно возвращать из функции на PL/pgSQL в случаях, когда нужно вернуть множество строк и столбцов, особенно если выборки очень большие. Для этого, в функции открывается курсор и его имя возвращается вызывающему (или просто открывается курсор, используя указанное имя портала, каким-либо образом известное вызывающему). Вызывающий затем может извлекать строки из курсора. Курсор может быть закрыт вызывающим или он будет автоматически закрыт при завершении транзакции.

Имя портала, используемое для курсора, может быть указано разработчиком или будет генерироваться автоматически. Чтобы указать имя портала, нужно просто присвоить строку в переменную refcursor перед его открытием. Значение строки переменной refcursor будет использоваться командой OPEN как имя портала. Однако если переменная refcursor имеет значение NULL, OPEN автоматически генерирует имя, которое не конфликтует с любым существующим порталом и присваивает его переменной refcursor.

28.6.7.5. Обработка курсора в цикле

Один из вариантов цикла FOR позволяет перебирать строки, возвращённые курсором. Синтаксис:

```
[ <<метка>> ]
FOR переменная-запись IN связанная_переменная_курсора [ ( [ имя_аргумента
:= ] значение_аргумента [, ...] ) ] LOOP
операторы
END LOOP [ метка ];
```

Курсорная переменная должна быть связана с запросом при объявлении. Курсор *не может* быть открытым. Команда FOR автоматически открывает курсор и автоматически закрывает при завершении цикла. Список фактических значений аргументов должен присутствовать только в том случае, если курсор объявлялся с параметрами.

Данная *переменная-запись* автоматически определяется как переменная типа record и существует только внутри цикла (другие объявленные переменные с таким именем игнорируется в цикле). Каждая возвращаемая курсором строка последовательно присваивается этой переменной и выполняется тело цикла.

2868. Управление транзакциями

В процедурах, вызываемых командой CALL, а также в анонимных блоках кода (в команде DO) можно завершать транзакции, выполняя COMMIT и ROLLBACK. После завершения транзакции этими командами новая будет начата автоматически, поэтому отдельной команды START TRANSACTION нет.

Пример:

```
CREATE PROCEDURE transaction_test1() LANGUAGE plpgsql
AS $$ BEGIN
FOR i IN 0..9 LOOP
INSERT INTO test1 (a) VALUES (i); IF i % 2 = 0 THEN
COMMIT;
ELSE
ROLLBACK;
END IF;
END LOOP;
END;
$$;

CALL transaction_test1();
```

Новая транзакция начинается с теми характеристиками, в частности, уровнем изоляции, которые установлены для транзакций по умолчанию. В случаях, когда транзакции фиксируются в цикле, может быть удобнее автоматически начинать следующую транзакцию с теми же характеристиками, что имеет предыдущая. Это позволяют реализовать команды COMMIT AND CHAIN и ROLLBACK AND CHAIN.

Управление транзакциями возможно только в вызовах CALL или DO в коде верхнего уровня или во вложенных CALL или DO без других промежуточных команд. Например, в стеке вызовов CALL proc1() → CALL proc2() → CALL proc3() вторая и третья процедуры могут управлять транзакциями. Но в стеке CALL proc1() → SELECT func2() → CALL proc3() последняя процедура лишена этой возможности из-за промежуточного SELECT.

Циклам с курсорами присущи некоторые особенности. Обычно курсоры автоматически закрываются при фиксировании транзакции. Однако курсор, создаваемый внутри цикла подобным образом, автоматически преобразуется в удерживаемый курсор первой командой COMMIT или ROLLBACK. Это означает, что курсор полностью вычисляется при выполнении первой команды

COMMIT или ROLLBACK, а не для каждой очередной строки. При этом он автоматически удаляется после цикла, так что это происходит практически незаметно для пользователя.

Команды управления транзакциями не допускаются в циклах с курсором, которыми управляют запросы, производящие не только чтение, но и модификацию данных (например, UPDATE ... RETURNING).

Транзакция не может завершаться внутри блока с обработчиками исключений.

28.6.9. Сообщения и ошибки

28.6.9.1. Вывод сообщений и ошибок

Команда RAISE предназначена для вывода сообщений и вызова ошибок.

```
RAISE [ уровень ] 'формат' [, выражение [, ... ]]
[ USING параметр = значение[, ... ] ];
RAISE [ уровень ] имя_условия [ USING параметр = выражение
[, ... ] ];
RAISE [ уровень ] SQLSTATE 'sqlstate'
[ USING параметр = выражение [, ... ] ]; RAISE [ уровень ]
USING параметр = выражение [, ... ];
RAISE ;
```

уровень задаёт уровень важности ошибки. Возможные значения: DEBUG, LOG, INFO, NOTICE, WARNING и EXCEPTION. По умолчанию используется EXCEPTION. EXCEPTION вызывает ошибку (что обычно прерывает текущую транзакцию), остальные значения *уровня* только генерируют сообщения с различными уровнями приоритета. Будут ли сообщения конкретного приоритета переданы клиенту или записаны в журнал сервера, или и то, и другое, зависит от конфигурационных переменных log_min_messages и client_min_messages.

После указания *уровня*, если оно есть, можно задать строку *формата* (это должна быть простая строковая константа, не выражение). Строка формата определяет вид текста об ошибке, который будет выдан. За строкой формата могут следовать необязательные выражения аргументов, которые будут вставлены в сообщение. Внутри строки формата знак % заменяется строковым представлением значения очередного аргумента. Чтобы выдать символ % буквально, необходимо продублировать его (как %%). Число аргументов должно совпадать с числом местозаполнителей % в строке формата, иначе при компиляции функции возникнет ошибка.

При помощи USING и последующих элементов *параметр* = *выражение* можно добавить дополнительную информацию к отчёту об ошибке. Все *выражения* представляют собой строковые выражения. Возможные ключевые слова для *параметра* следующие:

– MESSAGE – Устанавливает текст сообщения об ошибке. Этот параметр не может использоваться, если в команде RAISE присутствует *формат* перед USING.

– DETAIL – Предоставляет детальное сообщение об ошибке.

– HINT – Предоставляет подсказку по вызванной ошибке.

– ERRCODE – Устанавливает код ошибки (SQLSTATE). Код ошибки задаётся либо по имени, как показано в Приложении А, или напрямую, пятисимвольный код SQLSTATE.

– COLUMN CONSTRAINT DATATYPE TABLE SCHEMA – Предоставляет имя соответствующего объекта, связанного с ошибкой.

Ещё один вариант – использовать RAISE USING или RAISE *уровень* USING, а всё остальное записать в списке USING.

И заключительный вариант, в котором RAISE не имеет параметров вообще. Эта форма может использоваться только в секции EXCEPTION блока и предназначена для того, чтобы повторно вызвать ошибку, которая сейчас перехвачена и обрабатывается.

Если в команде RAISE EXCEPTION не задано ни имя условия, ни код SQLSTATE, по умолчанию выдаётся исключение ERRCODE_RAISE_EXCEPTION (P0001). Если не задан текст сообщения, по умолчанию в качестве этого текста передаётся имя условия или код SQLSTATE.

28.6.9.2. Проверка утверждений

Оператор ASSERT представляет удобное средство вставлять отладочные проверки в функции PL/ pgSQL.

```
ASSERT условие [ , сообщение ];
```

Здесь *условие* – это логическое выражение, которое, как ожидается, должно быть всегда истинным; если это так, оператор ASSERT больше ничего не делает. Если же оно возвращает ложь или NULL, этот оператор выдаёт исключение ASSERT_FAILURE. (Если ошибка происходит при вычислении *условия*, она выдаётся как обычная ошибка.)

Если в нём задаётся необязательное *сообщение*, результат этого выражения (если он не NULL) заменяет сообщение об ошибке по умолчанию «assertion failed» (нарушение истинности), в случае, если *условие* не выполняется. В обычном случае, когда условие утверждения выполняется, выражение *сообщения* не вычисляется.

Проверку утверждений можно включить или отключить с помощью конфигурационного параметра plpgsql.check_asserts, принимающего логическое значение; по умолчанию она включена (on). Если этот параметр отключён (off), операторы ASSERT ничего не делают.

Оператор ASSERT предназначен для выявления программных дефектов, а не для вывода обычных ошибок (для этого используется оператор RAISE, описанный выше).

28.6.10. Триггерные функции

В PL/pgSQL можно создавать триггерные функции, которые будут вызываться при изменениях данных или событиях в базе данных. Триггерная функция создаётся командой CREATE FUNCTION, при этом у функции не должно быть аргументов, а типом возвращаемого значения должен быть trigger (для триггеров, срабатывающих при изменениях данных) или event_trigger (для триггеров, срабатывающих при событиях в базе). Для триггеров автоматически определяются специальные локальные переменные с именами вида TG_*имя*, описывающие условие, повлёкшее вызов триггера.

28.6.10.1. Триггеры при изменении данных

Триггер при изменении данных объявляется как функция без аргументов и с типом результата trigger. Эта функция должна объявляться без аргументов, даже если ожидается, что она будет получать аргументы, заданные в команде CREATE TRIGGER – такие аргументы передаются через TG_ARGV, как описано ниже.

Когда функция на PL/pgSQL срабатывает как триггер, в блоке верхнего уровня автоматически создаются несколько специальных переменных:

– NEW – Тип данных RECORD. Переменная содержит новую строку базы данных для команд INSERT/UPDATE в триггерах уровня строки. В триггерах уровня оператора и для команды DELETE эта переменная имеет значение null.

– OLD – Тип данных RECORD. Переменная содержит старую строку базы данных для команд UPDATE/ DELETE в триггерах уровня строки. В триггерах уровня оператора и для команды INSERT эта переменная имеет значение null.

– TG_NAME – Тип данных name. Переменная содержит имя сработавшего триггера.

– TG_WHEN – Тип данных text. Строка, содержащая BEFORE, AFTER или INSTEAD OF, в зависимости от определения триггера.

– TG_LEVEL – Тип данных text. Строка, содержащая ROW или STATEMENT, в зависимости от определения триггера.

– TG_OP – Тип данных text. Строка, содержащая INSERT, UPDATE, DELETE или TRUNCATE, в зависимости от того, для какой операции сработал триггер.

– TG_RELID – Тип данных oid. OID таблицы, для которой сработал триггер.

– TG_RELNAME – Тип данных name. Имя таблицы, для которой сработал триггер.

– TG_TABLE_NAME – Тип данных name. Имя таблицы, для которой сработал триггер.

– TG_TABLE_SCHEMA – Тип данных name. Имя схемы, содержащей таблицу, для которой сработал триггер.

– TG_NARGS – Тип данных integer. Число аргументов в команде CREATE TRIGGER, которые передаются в триггерную функцию.

– TG_ARGV[] – Тип данных массив text. Аргументы от оператора CREATE TRIGGER. Индекс массива начинается с 0. Для недопустимых значений индекса (< 0 или >= tg_nargs) возвращается NULL.

Триггерная функция должна вернуть либо NULL, либо запись/строку, соответствующую структуре таблице, для которой сработал триггер.

Если BEFORE триггер уровня строки возвращает NULL, то все дальнейшие действия с этой строкой прекращаются (т. е. не срабатывают последующие триггеры, команда INSERT/UPDATE/DELETE для этой строки не выполняется). Если возвращается не NULL, то дальнейшая обработка продолжается именно с этой строкой. Возвращение строки отличной от начальной NEW, изменяет строку, которая будет вставлена или изменена. Поэтому, если в триггерной функции нужно выполнить некоторые действия и не менять саму строку, то нужно возвратить переменную NEW (или её эквивалент). Для того чтобы изменить сохраняемую строку, можно поменять отдельные значения в переменной NEW и затем её вернуть. Либо создать и вернуть полностью новую переменную. В случае строчного триггера BEFORE для команды DELETE само возвращаемое значение не имеет прямого эффекта, но оно должно быть отличным от NULL, чтобы не прерывать обработку строки. Переменная NEW всегда NULL в триггерах на DELETE, поэтому возвращать её не имеет смысла. Традиционной идиомой для триггеров DELETE является возврат переменной OLD.

Триггеры INSTEAD OF (это всегда триггеры уровня строк и они могут применяться только с представлениями) могут возвращать NULL, чтобы показать, что они не выполняли никаких

изменений, так что обработку этой строки можно не продолжать (то есть, не вызывать последующие триггеры и не считать строку в числе обработанных строк для окружающих команд INSERT/UPDATE/DELETE). В противном случае должно быть возвращено значение, отличное от NULL, показывающее, что триггер выполнил запрошенную операцию. Для операций INSERT и UPDATE возвращаемым значением должно быть NEW, которое триггерная функция может модифицировать для поддержки предложений INSERT RETURNING и UPDATE RETURNING (это также повлияет на значение строки, передаваемое последующим триггерам, или доступное под специальным псевдонимом EXCLUDED в операторе INSERT с предложением ON CONFLICT DO UPDATE). Для операций DELETE возвращаемым значением должно быть OLD.

Возвращаемое значение для строчного триггера AFTER и триггеров уровня оператора (BEFORE или AFTER) всегда игнорируется. Это может быть и NULL. Однако в этих триггерах по-прежнему можно прервать вызвавшую их команду, для этого нужно явно вызвать ошибку.

28.6.10.2. Триггеры событий

В PL/pgSQL можно создавать событийные триггеры. PG360 требует, чтобы функция, которая вызывается как событийный триггер, объявлялась без аргументов и типом возвращаемого значения был event_trigger. Когда функция на PL/pgSQL вызывается как событийный триггер, в блоке верхнего уровня автоматически создаются несколько специальных переменных: TG_EVENT Тип данных text. Строка, содержащая событие, для которого сработал триггер. TG_TAG Тип данных text. Переменная, содержащая тег команды, для которой сработал триггер.

Пример реализации функции событийного триггера на PL/pgSQL:

```
CREATE OR REPLACE FUNCTION snitch() RETURNS event_trigger AS $$
BEGIN
RAISE NOTICE 'snitch: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;
CREATE EVENT TRIGGER snitch ON ddl_command_start EXECUTE FUNCTION
snitch();
```

Триггер в этом примере выдаёт сообщение NOTICE каждый раз, когда выполняется поддерживаемая команда.

28.7. PL/Tcl – процедурный язык Tcl

PL/Tcl – это загружаемый процедурный язык для СУБД PG360, позволяющий использовать язык Tcl для написания функций и процедур PG360.

28.7.1. Обзор

PL/Tcl предоставляет большинство возможностей, которые имеет разработчик функций на C, с небольшими ограничениями, и позволяет применять мощные библиотеки обработки строк, существующие для Tcl.

Ограничением является то, что весь код выполняется в контексте безопасности интерпретатора Tcl. Помимо ограниченного набора команд безопасного Tcl, разрешены только несколько команд для обращения к базе данных через SPI и вызовы elog() для выдачи сообщений. PL/Tcl не даёт возможности взаимодействовать с внутренним механизмом сервера баз данных или

обращаться к ОС с правами серверного процесса PG360, что возможно в функциях на C. Таким образом, использование этого языка можно доверить непривилегированным пользователям; это не даст им неограниченные полномочия.

Ещё одно существенное ограничение заключается в том, что функции на Tcl нельзя использовать для создания функций ввода/вывода для новых типов данных.

Иногда возникает желание написать функцию на Tcl, которая не будут ограничена безопасным Tcl. Например, может потребоваться функция, которая будет посылать сообщения по почте. Для этих случаев есть вариация PL/Tcl, названная PL/TclU (название подразумевает «untrusted Tcl», недоверенный Tcl). Это тот же язык, за исключением того, что для него используется полноценный интерпретатор Tcl. *Если применяется PL/TclU, он должен быть установлен как недоверенный процедурный язык*, чтобы только суперпользователи могли создавать функции на нём. Автор функции на PL/TclU должен позаботиться о том, чтобы эту функцию нельзя было использовать не по назначению, так как она может делать всё, что может пользователь с правами администратора баз данных.

Разделяемый объектный код для обработчиков вызова PL/Tcl и PL/TclU собирается автоматически и устанавливается в каталог библиотек PG360, если поддержка Tcl включена на этапе конфигурирования процедуры установки. Чтобы установить PL/Tcl и/или PL/TclU в конкретную базу данных, необходимо воспользоваться командой CREATE EXTENSION, например: CREATE EXTENSION pltcl или CREATE EXTENSION pltclu.

2872. Функции на PL/Tcl и их аргументы

Чтобы создать функцию на языке PL/Tcl, необходимо использовать стандартный синтаксис CREATE FUNCTION:

```
CREATE FUNCTION имя_функции (типы_аргументов) RETURNS тип_результата AS $$
# Тело функции на PL/Tcl
$$ LANGUAGE pltcl;
```

С PL/TclU команда та же, но в качестве языка должно быть указано pltclu.

Тело функции содержит скрипт на Tcl. Когда вызывается функция, значения аргументов передаются скрипту Tcl в виде переменных с именами 1 ... *n*. Результат из кода Tcl возвращается как обычно, оператором return. В процедуре значение, возвращаемое из кода Tcl, игнорируется.

Например, функцию, возвращающую большее из двух целых чисел, можно определить следующим образом:

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
if {$1 > $2} {return $1}
return $2
$$ LANGUAGE pltcl STRICT;
```

Необходимо обратить внимание на предложение STRICT, которое избавляет от необходимости думать о входящих значениях NULL: если при вызове передаётся значение NULL, функция не будет выполняться вовсе, будет сразу возвращён результат NULL.

В нестрогой функции, если фактическое значение аргумента – NULL, соответствующей переменной $*n* будет присвоена пустая строка. Чтобы определить, был ли передан NULL в определённом аргументе, необходимо использовать функцию argisnull.

Функции PL/Tcl могут возвращать и результаты составного типа. Для этого код на Tcl должен вернуть список пар имя/значение столбца, соответствующий ожидаемому типу результата. Столбцы, имена которых в этом списке отсутствуют, получат значения NULL, а если в списке указано имя несуществующего столбца, возникнет ошибка.

Функции PL/Tcl могут возвращать наборы результатов. Для этого код на Tcl должен вызывать return_next для каждой возвращаемой строки, передавая ей соответствующее значение, когда возвращается скалярный тип, или список пар имя/значение столбца, когда возвращается составной тип.

2873. Значения данных в PL/Tcl

Значения аргументов, передаваемые в код функции PL/Tcl, представляют собой просто входные аргументы, преобразованные в текстовый вид (так же, как при выводе оператором SELECT). И наоборот, команды return и return_next примут любую строку, соответствующую формату ввода для объявленного типа результата функции или заданного столбца в результате составного типа.

2874. Глобальные данные в PL/Tcl

Иногда необходимо иметь некоторые глобальные данные, сохраняемые между двумя вызовами функции или совместно используемые разными функциями. Это легко сделать в PL/Tcl, но есть некоторые ограничения, которые необходимо понимать.

По соображениям безопасности, PL/Tcl выполняет функции, вызываемые некоторой ролью SQL в отдельном интерпретаторе Tcl, выделенном для этой роли. Это предотвращает случайное или злонамеренное влияние одного пользователя на поведение функций PL/Tcl другого пользователя. В каждом интерпретаторе будут свои значения всех «глобальных» переменных Tcl. Таким образом, в двух функциях PL/Tcl будут общие глобальные переменные, только если они выполняются одной ролью SQL. В приложении, выполняющем код в одном сеансе с разными ролями SQL (вызывающем функции SECURITY DEFINER, использующем команду SET ROLE и т. д.) может понадобиться явно предпринять дополнительные меры, чтобы функции могли разделять свои данные. Для этого сначала необходимо установить для функций, которые должны взаимодействовать, одного владельца, а затем задать для них свойство SECURITY DEFINER.

Все функции PL/TclU, вызываемые в одном сеансе, выполняются одним интерпретатором Tcl, который отличается от интерпретатора(ов), используемого для функций PL/Tcl. Поэтому глобальные данные функций PL/TclU автоматически становятся общими. Это не считается угрозой безопасности, так как все функции PL/TclU выполняются на одном уровне доверия, а именно уровне суперпользователя базы данных.

Чтобы защитить функции PL/Tcl от непреднамеренного влияния друг на друга, каждой из них предоставляется глобальная переменная-массив через команду upvar. Глобальным именем этой переменной является внутреннее имя функции, а в качестве локального выбрано GD. Переменную GD рекомендуется использовать для постоянных внутренних данных функции.

Обычные глобальные переменные Tcl следует использовать только для значений, которые предназначены именно для совместного использования несколькими функциями.

В этом разделе для обозначения необязательных элементов в описании синтаксиса используются знаки вопроса (а не квадратные скобки), как это принято в Tcl. Имеются следующие команды для доступа к базе данных из тела функции PL/Tcl:

`spi_exec ?-count` *n*`? ?-array` *имя*`?` *команда* `?`*тело-цикла*`?`

Выполняет команду SQL, заданную в виде строки. В случае ошибки в этой команде выдаётся ошибка в Tcl. В противном случае spi_exec возвращает число обработанных командой строк (выбранных, добавленных, изменённых или удалённых), либо ноль, если эта команда – служебный оператор. Кроме того, если команда – оператор SELECT, значения выбранных столбцов помещаются в переменные Tcl, как описано ниже.

Необязательное значение -count задаёт для spi_exec максимальное число строк, которое должно быть обработано в команде. Его действие можно представить как выполнение FETCH *n* для курсора, предварительно подготовленного для команды.

Если в качестве команды выполняется оператор SELECT, значения результирующих столбцов помещаются в переменные Tcl, названные по именам столбцов. Если передаётся -array, значения столбцов вместо этого становятся элементами названного ассоциативного массива, индексами в котором становятся имена столбцов. Кроме того, в элементе с именем «.tupno» сохраняется номер текущей строки в результирующем наборе (отсчитывая от нуля), если только это имя не занято одним из столбцов результата.

Если в качестве команды выполняется SELECT без указания скрипта *тело-цикла*, в переменных Tcl или элементах массива сохраняется только первая строка результатов; оставшиеся строки (если они есть), игнорируются. Если запрос не возвращает строки, не сохраняется ничего.

Если передаётся необязательный аргумент *тело-цикла*, заданный в нём блок скрипта Tcl будет выполняться для каждой строки результата запроса. (Аргумент *тело-цикла* игнорируется, если целевая команда – не SELECT.) При этом значения столбцов текущей строки сохраняются в переменных Tcl или элементах массива перед каждой итерацией этого цикла.

Если в столбце результата запроса выдаётся NULL, целевая переменная для неё не устанавливается, и оказывается «неустановленной».

`spi_prepare` *запрос список-типов*

Подготавливает и сохраняет план запроса для последующего выполнения. Сохранённый план будет продолжать существование до завершения текущего сеанса.

Запрос может принимать параметры, то есть местозаполнители для значений, которые будут передаваться, когда план будет собственно выполняться. В строке запроса эти параметры обозначаются как $1 ... $*n*. Если в запросе используются параметры, нужно задать имена типов этих параметров в виде списка Tcl. (Если параметры отсутствуют, необходимо задать пустой *список_типов*.)

Функция spi_prepare возвращает идентификатор запроса, который может использоваться в последующих вызовах spi_execp.

`spi_execp ?-count` *n*`? ?-array` *имя*`? ?-nulls` *строка*`?` *ид-запроса* `?` *список-значений*`? ?`*тело- цикла*`?`

Выполняет запрос, ранее подготовленный функцией spi_prepare. В качестве *ид_запроса* передаётся идентификатор, возвращённый функцией spi_prepare. Если в запросе задействуются

параметры, необходимо указать *список-значений*. Это должен быть принятый в Tcl список параметров. Он должен иметь ту же длину, что и список типов параметров, ранее переданный spi_prepare. Необходимо опустить *список-значений*, если у запроса нет параметров.

Необязательный аргумент -nulls принимает строку из пробелов и символов 'n', которые отмечают, в каких параметрах spi_execp передаются значения NULL. Если присутствует, эта строка должна иметь ту же длину, что и *список-значений*. В случае её отсутствия значения всех параметров считаются отличными от NULL.

Не считая отличий в способе передачи запроса и параметров, spi_execp работает так же, как spi_exec. Параметры -count, -array и *тело-цикла* задаются так же, и так же передаётся возвращаемое значение.

`subtransaction` *`команда`*

Скрипт Tcl, который содержит *команда*, выполняется в подтранзакции SQL. Если этот скрипт возвращает ошибку, вся подтранзакция откатывается назад, а затем в окружающий код Tcl возвращается ошибка.

`quote строка`

Дублирует все вхождения апострофа и обратной косой черты в заданной строке. Это можно использовать для защиты строк, которые будут вставляться в команды SQL, передаваемые в spi_exec или spi_prepare.

`elog уровень сообщение`

Выдаёт служебное сообщение или сообщение об ошибке. Возможные уровни сообщений: DEBUG (ОТЛАДКА), LOG (СООБЩЕНИЕ), INFO (ИНФОРМАЦИЯ), NOTICE (ЗАМЕЧАНИЕ), WARNING (ПРЕДУПРЕЖДЕНИЕ), ERROR (ОШИБКА) и FATAL (ВАЖНО). С уровнем ERROR выдаётся ошибка; если она не перехватывается окружающим кодом Tcl, она распространяется в вызывающий запрос, что приводит к прерыванию текущей транзакции или подтранзакции. По сути то же самое делает команда error языка Tcl. Сообщение уровня FATAL прерывает транзакцию и приводит к завершению текущего сеанса. (Вероятно, нет обоснованной причины использовать этот уровень ошибок в функциях PL/Tcl, но он поддерживается для полноты.) При использовании других уровней происходит просто вывод сообщения с заданным уровнем важности. Будут ли сообщения определённого уровня передаваться клиенту и/или записываться в журнал, определяется конфигурационными переменными log_min_messages и client_min_messages.

2876. Триггерные функции на PL/Tcl

На PL/Tcl можно написать триггерные функции. PG360 требует, чтобы функция, которая будет вызываться как триггерная, была объявлена как функция без аргументов и возвращала тип trigger.

Информация от менеджера триггеров передаётся в тело функции в следующих переменных:

– $TG_name – Имя триггера из оператора CREATE TRIGGER.

– $TG_relid – Идентификатор объекта таблицы, для которой будет вызываться триггерная функция.

– $TG_table_name – Имя таблицы, для которой будет вызываться триггерная функция.

– $TG_table_schema – Схема таблицы, для которой будет вызываться триггерная функция.

– $TG_relatts – Список языка Tcl, содержащий имена столбцов таблицы. В начало списка добавлен пустой элемент, поэтому при поиске в этом списке имени столбца с помощью стандартной в Tcl команды lsearch будет возвращён номер элемента, начиная с 1, так же, как нумеруются столбцы в PG360. (В позициях удалённых столбцов также содержатся пустые элементы, так что нумерация следующих за ними атрибутов не нарушается.)

– $TG_when – Строка BEFORE, AFTER или INSTEAD OF, в зависимости от типа события триггера.

– $TG_level – Строка ROW или STATEMENT, в зависимости от уровня события триггера.

– $TG_op – Строка INSERT, UPDATE, DELETE или TRUNCATE, в зависимости от действия события триггера.

– $NEW – Ассоциативный массив, содержащий значения новой строки таблицы для действий INSERT или UPDATE, либо пустой массив для DELETE. Индексами в массиве являются имена столбцов. Столбцы со значениями NULL в нём отсутствуют. Для триггеров уровня оператора этот массив не определяется.

– $OLD – Ассоциативный массив, содержащий значения старой строки таблицы для действий UPDATE или DELETE, либо пустой массив для INSERT. Индексами в массиве являются имена столбцов. Столбцы со значениями NULL в нём отсутствуют. Для триггеров уровня оператора этот массив не определяется.

– $args – Список на языке Tcl аргументов функции, заданных в операторе CREATE TRIGGER. Эти аргументы также доступны под обозначениями $1 ... $*n* в теле функции.

Возвращаемым значением триггерной функции может быть строка OK или SKIP либо список пар имя столбца/значение. Если возвращается значение OK, операция (INSERT/UPDATE/DELETE), которая привела к срабатыванию триггера, выполняется нормально. Значение SKIP указывает менеджеру триггеров просто пропустить эту операцию с текущей строкой данных. Если возвращается список, через него PL/Tcl передаёт менеджеру триггеров изменённую строку; содержимое изменённой строки задаётся именами и значениями столбцов в списке. Все столбцы, не перечисленные в этом списке, получают значения NULL. Возвращать изменённую строку имеет смысл только для триггеров уровня строки с порядком BEFORE команд INSERT и UPDATE, в которых вместо заданной в $NEW будет записываться изменённая строка; либо с порядком INSTEAD OF команд INSERT и UPDATE, в которых возвращаемая строка служит исходными данными для предложений INSERT RETURNING или UPDATE RETURNING. В триггерах уровня строки с порядком BEFORE или INSTEAD OF команды DELETE возврат изменённой строки воспринимается так же, как и возврат значения OK, то есть операция выполняется. Для всех остальных типов триггеров возвращаемое значение игнорируется.

28.7.7. Функции событийных триггеров в PL/Tcl

На PL/Tcl можно написать функции событийных триггеров. PG360 требует, чтобы функция, которая будет вызываться как событийный триггер, была объявлена как функция без аргументов и возвращала тип event_trigger.

Информация от менеджера триггеров передаётся в тело функции в следующих переменных:

– $TG_event – Имя события, при котором срабатывает этот триггер.

– $TG_tag – Тег команды, для которой срабатывает этот триггер.

Возвращаемое значение триггерной функции игнорируется.

28.7.8. Обработка ошибок в PL/Tcl

Tcl-код, содержащийся или вызываемый из функции PL/Tcl, может выдавать ошибку либо выполняя недопустимую операцию, либо генерируя ошибку с помощью команды error языка Tcl или команды elog языка PL/Tcl. Такие ошибки могут быть перехвачены в среде Tcl с помощью команды Tcl catch. Если ошибка не перехватывается, а распространяется выше уровня выполнения функций PL/Tcl, она передаётся в запрос, вызвавший функцию, как ошибка SQL.

И напротив, ошибки СУБД, возникающие внутри команд spi_exec, spi_prepare и spi_execp в среде PL/Tcl, выдаются как ошибки Tcl, так что их можно перехватить командой Tcl catch. (Каждая из этих команд PL/Tcl выполняет SQL-операцию в подтранзакции, которая откатывается в случае ошибки, так что для частично завершённых операций производится автоматическая очистка.) Опять же, если ошибка не перехватывается и распространяется выше верхнего уровня, она становится ошибкой SQL.

В Tcl имеется переменная errorCode, представляющая дополнительную информацию об ошибке в виде, удобном для обработки в программах на Tcl. Эта информация передаётся в формате списка Tcl, первое слово в котором указывает на подсистему или библиотеку, выдающую ошибку; последующее содержимое определяется в зависимости от подсистемы или библиотеки. Для ошибок СУБД, возникающих в командах PL/Tcl, первым словом будет POSTGRES, вторым – номер версии PG360, а дополнительные слова представляют пары имя/значения, передающие подробную информацию об ошибке. В этих парах всегда передаются поля SQLSTATE, condition и message. Также могут передаваться поля detail, hint, context, schema, table, column, datatype, constraint, statement, cursor_position, filename, lineno и funcname.

С информацией в переменной errorCode среды PL/Tcl удобно работать, загрузив переменную в массив, чтобы имена полей стали индексами в массиве.

28.7.9. Управление транзакциями

В процедуре, которая вызывается в коде верхнего уровня или в анонимном блоке кода (в команде DO), можно управлять транзакциями. Чтобы зафиксировать текущую транзакцию, необходимо выполнить команду commit, а чтобы откатить – rollback. (Выполнить SQL-команды COMMIT или ROLLBACK через spi_exec или подобную функцию нельзя. Соответствующие операции могут выполняться только данными функциями.) После завершения одной транзакции следующая начинается автоматически, отдельной функции для этого нет.

28.7.10. Конфигурация PL/Tcl

Параметры конфигурации, влияющие на работу PL/Tcl:

`pltcl.start_proc (string)`

В этом параметре, если он не пуст, задаётся имя (возможно, дополненное схемой) функции на языке PL/Tcl без параметров, которая будет выполняться, когда для PL/Tcl будет создаваться новый экземпляр Tcl. Такая функция может выполнять инициализацию в рамках сеанса, например, загружать дополнительный код Tcl. Новый интерпретатор Tcl создаётся при первом

выполнении какой-либо функции PL/Tcl в сеансе базы данных или когда требуется дополнительный интерпретатор из-за того, что функция PL/Tcl была вызвана новой ролью SQL.

Указанная функция должна быть написана на языке pltcl и не должна иметь свойство SECURITY DEFINER. (Благодаря этим ограничениям эта функция будет запускаться в интерпретаторе, который она должна инициализировать.) Текущий пользователь должен иметь право и на её выполнение тоже.

Если эта функция завершится ошибкой, эта ошибка прервёт вызов функции, которой потребовался новый интерпретатор, и распространится в вызывающий запрос, приводя к прерыванию текущей транзакции или подтранзакции. Любые действия, уже произведённые в среде Tcl, отменены не будут; однако этот интерпретатор более не будет использоваться. При следующей попытке использования этого языка последует повторная попытка инициализации со свежим интерпретатором Tcl.

Изменять этот параметр разрешено только суперпользователям. Хотя изменить его можно в рамках сеанса, такие изменения не повлияют на работу интерпретаторов Tcl, созданных ранее.

`pltclu.start_proc (string)`

Это параметр полностью аналогичен pltcl.start_proc, но применяется к PL/TclU. Указанная функция должна быть написана на языке pltclu.

28.7.11. Имена процедур Tcl

В PG360 одно имя функции может использоваться разными определениями функций, если они имеют разное число и типы аргументов. Tcl, однако, требует, чтобы имена всех процедур различались. PL/Tcl решает эту проблему, устанавливая такие внутренние имена процедур Tcl, чтобы они включали в свой состав OID функции из системной таблицы pg_proc. Таким образом, функциям PG360 с одним именем и разными типами аргументов так же будут соответствовать различные процедуры Tcl.

28.8. PL/Perl – процедурный язык Perl

PL/Perl – это загружаемый процедурный язык, позволяющий реализовывать функции и процедуры PG360 на языке программирования Perl.

Основным преимуществом PL/Perl является то, что он позволяет применять в сохранённых функциях и процедурах множество функций и операторов «перемалывания строк», имеющихся в Perl. Разобрать сложные строки на языке Perl может быть гораздо проще, чем используя строковые функции и управляющие структуры в PL/pgSQL.

Чтобы установить PL/Perl в определённую базу данных, необходимо выполнить команду:

`CREATE EXTENSION plperl.`

28.8.1. Функции на PL/Perl и их аргументы

Чтобы создать функцию на языке PL/Perl, необходимо использовать стандартный синтаксис CREATE FUNCTION:

```
CREATE FUNCTION имя_функции (типы-аргументов) RETURNS тип-результата
-- здесь описываются атрибуты функции AS $$
# Тело функции на PL/Perl
```

```
$$ LANGUAGE plperl;
```

Тело функции содержит код Perl. Фактически, код обвязки PL/Perl помещает этот код в подпрограмму Perl. Функция PL/Perl вызывается в скалярном контексте, так что она не может вернуть список. Не скалярные значения (массивы, записи и множества) можно вернуть по ссылке, как описывается ниже.

В процедуре PL/Perl возвращаемое из кода Perl значение игнорируется.

PL/Perl также поддерживает анонимные блоки кода, которые выполняются оператором DO:

```
DO $$
# Код PL/Perl
$$ LANGUAGE plperl;
```

Анонимный блок кода не принимает аргументы, а любое значение, которое он мог бы вернуть, отбрасывается. В остальном он работает подобно коду функции.

Синтаксис команды CREATE FUNCTION требует, чтобы тело функции было записано как строковая константа.

Аргументы и результат обрабатываются как и в любой другой подпрограмме на Perl: аргументы передаются в @_, а результирующим значением будет указанное в return или полученное в последнем выражении, вычисленном в функции.

Если функции передаётся NULL-значение SQL, значением аргумента в Perl станет «undefined».

Всё в аргументах функции, что не является ссылкой, является строкой, то есть стандартным для PG360 внешним текстовым представлением соответствующего типа данных. В случае с обычными числовыми или текстовыми типами, Perl просто воспринимает их должным образом, и программист, как правило, может об этом не думать. Однако в более сложных случаях может потребоваться преобразовать аргумент в форму, подходящую для использования в Perl. Например, для преобразования типа bytea в двоичное значение можно использовать функцию decode_bytea.

Аналогично, значения, передаваемые в PG360, должны быть в формате внешнего текстового представления. Например, для подготовки двоичных данных к возврату в значении bytea можно воспользоваться функцией encode_bytea.

2882. Значения в PL/Perl

Значения аргументов, передаваемые в код функции PL/Perl, представляют собой просто входные аргументы, преобразованные в текстовый вид (так же, как при выводе оператором SELECT). И наоборот, команды return и return_next могут принять любую строку, соответствующую формату ввода для объявленного типа результата функции.

2883. Встроенные функции

28831. Обращение к базе данных из PL/Perl

Обращаться к самой базе данных из кода Perl можно, используя следующие функции:

```
spi_exec_query(запрос [, макс-строк])
```

spi_exec_query выполняет команду SQL и возвращает весь набор строк в виде ссылки на массив хешей.

```
spi_query(команда)
```

```
spi_fetchrow(cursor)
spi_cursor_close(cursor)
```

Функции spi_query и spi_fetchrow применяются в паре, когда набор строк может быть очень большим или когда нужно возвращать строки по мере их поступления. Функция spi_fetchrow работает *только* с spi_query.

Обычно вызов spi_fetchrow нужно повторять, пока не будет получен результат undef, показывающий, что все строки уже прочитаны. Курсор, возвращаемый функцией spi_query, автоматически освобождается, когда spi_fetchrow возвращает undef. Если не нужно читать все строки, необходимо освободить курсор, выполнив spi_cursor_close, чтобы не допустить утечки памяти.

```
spi_prepare(команда, типы аргументов)
spi_query_prepared(план, аргументы)
spi_exec_prepared(план [, атрибуты], аргументы)
spi_freeplan(план)
```

Функции spi_prepare, spi_query_prepared, spi_exec_prepared и spi_freeplan реализуют ту же функциональность, но для подготовленных запросов. Функция spi_prepare принимает строку запроса с нумерованными местозаполнителями аргументов ($1, $2 и т. д.) и список строк с типами аргументов:

```
$plan = spi_prepare('SELECT * FROM test WHERE id > $1 AND name =
$2', 'INTEGER', 'TEXT');
```

План запроса, подготовленный вызовом spi_prepare, можно использовать вместо строки запроса либо в spi_exec_prepared, возвращающей тот же результат, что и spi_exec_query, либо в spi_query_prepared, возвращающей курсор так же, как spi_query, который затем можно передать в spi_fetchrow. В необязательном втором параметре spi_exec_prepared можно передать хеш с атрибутами; в настоящее время поддерживается только атрибут limit, задающий максимальное число строк, которое может вернуть запрос.

Подготовленные запросы хороши тем, что позволяют использовать единожды подготовленный план для неоднократного выполнения запроса. Когда план оказывается не нужен, его можно освободить, вызвав spi_freeplan.

```
spi_commit()
spi_rollback()
```

Эти функции фиксируют или откатывают текущую транзакцию. Они могут вызываться только в процедурах или в анонимных блоках кода (в команде DO), вызываемых из кода верхнего уровня. (Выполнить SQL-команды COMMIT или ROLLBACK через spi_exec_query или подобную функцию нельзя. Соответствующие операции могут выполняться только данными функциями.) После завершения одной транзакции следующая начинается автоматически, отдельной функции для этого нет.

28832. Вспомогательные функции в PL/Perl

```
elog(уровень, сообщение)
```

Выдаёт служебное сообщение или сообщение об ошибке. Возможные уровни сообщений: DEBUG (ОТЛАДКА), LOG (СООБЩЕНИЕ), INFO (ИНФОРМАЦИЯ), NOTICE (ЗАМЕЧАНИЕ), WARNING (ПРЕДУПРЕЖДЕНИЕ) и ERROR (ОШИБКА). С уровнем ERROR выдаётся ошибка;

если она не перехватывается окружающим кодом Perl, она распространяется в вызывающий запрос, что приводит к прерыванию текущей транзакции или подтранзакции. По сути то же самое делает команда die языка Perl. При использовании других уровней происходит просто вывод сообщения с заданным уровнем важности. Будут ли сообщения определённого уровня передаваться клиенту и/или записываться в журнал, определяется конфигурационными параметрами log_min_messages и client_min_messages.

`quote_literal(`*`строка`*`)`

Оформляет переданную строку для использования в качестве текстовой строки в SQL–операторе. Включённые в неё апострофы и обратная косая черта при этом дублируются. quote_literal возвращает undef, когда получает аргумент undef; если такие аргументы возможны, часто лучше использовать quote_nullable.

`quote_nullable(`*`строка`*`)`

Оформляет переданную строку для использования в качестве текстовой строки в SQL–операторе; либо, если поступает аргумент undef, возвращает строку «NULL» (без кавычек). Символы апостроф и обратная косая черта дублируются должным образом.

`quote_ident(`*`строка`*`)`

Оформляет переданную строку для использования в качестве идентификатора в SQL–операторе. При необходимости идентификатор заключается в кавычки (например, если он содержит символы, недопустимые в открытом виде, или буквы в разном регистре). Если переданная строка содержит кавычки, они дублируются.

`decode_bytea(`*`строка`*`)`

Возвращает неформатированные двоичные данные, представленные содержимым заданной строки, которая должна быть закодирована как bytea.

`encode_bytea(`*`строка`*`)`

Возвращает закодированные в виде bytea двоичные данные, содержащиеся в переданной строке.

`encode_array_literal(`*`массив`*`)` `encode_array_literal(`*`массив`*`, `*`разделитель`*`)`

Возвращает содержимое указанного массива в виде строки в формате массива. Возвращает значение аргумента неизменённым, если это не ссылка не массив. Разделитель элементов в строке массива по умолчанию – «, » (если разделитель не определён или undef).

`encode_typed_literal(`*`значение, имя_типа`*`)`

Преобразует переменную Perl в значение типа данных, указанного во втором аргументе, и возвращает строковое представление этого значения. Корректно обрабатывает вложенные массивы и значения составных типов.

`encode_array_constructor(`*`массив`*`)`

Возвращает содержимое переданного массива в виде строки в формате конструктора массива. Отдельные значения заключаются в кавычки функцией quote_nullable. Возвращает значение аргумента, заключённое в кавычки функцией quote_nullable, если аргумент – не ссылка на массив.

`looks_like_number(`*`строка`*`)`

Возвращает значение true, если содержимое переданной строки похоже на число, по правилам Perl, и false в обратном случае. Возвращает undef для аргумента undef. Ведущие и

замыкающие пробелы игнорируются. Строки Inf и Infinity считаются представляющими число (бесконечность).

`is_array_ref(`*аргумент*`)`

Возвращает значение true, если переданный аргумент можно воспринять как ссылку на массив, то есть это ссылка на ARRAY или PostgreSQL::InServer::ARRAY. В противном случае возвращает false.

2884. Глобальные значения в PL/Perl

Можно использовать для хранения данных, включая ссылки на код, глобальный хеш %_SHARED. Эти данные будут сохраняться между вызовами функции на протяжении всего текущего сеанса.

По соображениям безопасности, PL/Perl выполняет функции, вызываемые некоторой ролью SQL, в отдельном интерпретаторе Perl, выделенном для этой роли. Это предотвращает случайное или злонамеренное влияние одного пользователя на поведение функций PL/Perl другого пользователя. В каждом интерпретаторе будет своё значение переменной %_SHARED и собственное глобальное состояние. Таким образом, две функции PL/Perl будут разделять одно значение %_SHARED, только если они выполняются одной ролью SQL. В приложении, выполняющем код в одном сеансе с разными ролями SQL (вызывающем функции SECURITY DEFINER, использующем команду SET ROLE и т. д.) может понадобиться явно предпринять дополнительные меры, чтобы функции на PL/Perl могли разделять данные через %_SHARED. Для этого сначала необходимо установить для функций, которые должны взаимодействовать, одного владельца, а затем задать для них свойство SECURITY DEFINER.

2885. Доверенный и недоверенный PL/Perl

Обычно PL/Perl устанавливается в базу данных как «доверенный» язык программирования с именем plperl. При этом в целях безопасности определённые операции в Perl запрещаются. Вообще говоря, запрещаются все операции, взаимодействующие с окружением. В том числе, это операции с файлами, require и use (для внешних модулей). Поэтому функции на PL/Perl, в отличие от функций на C, никаким образом не могут взаимодействовать с внутренними механизмами сервера баз данных или обращаться к ОС с правами серверного процесса. Вследствие этого, использовать этот язык можно разрешить любому непривилегированному пользователю баз данных.

В следующем примере показана функция, которая не будет работать, потому что операции с файловой системы запрещены по соображениям безопасности:

```
CREATE FUNCTION badfunc() RETURNS integer AS $$ my $tmpfile =
"/tmp/badfile";
open my $fh, '>', $tmpfile
or elog(ERROR, qq{could not open the file "$tmpfile": $!}); print
$fh "Testing writing to a file\n";
close $fh or elog(ERROR, qq{could not close the file "$tmpfile":
$!}); return 1;
$$ LANGUAGE plperl;
```

Создать эту функцию не удастся, так как при проверке её правильности будет обнаружено использование запрещённого оператора.

Для потребностей написать на Perl код, функциональность которого не будет ограничиваться, PL/Perl также можно установить как «недоверенный» язык (обычно его называют PL/PerlU). В этом случае будут доступны все возможности языка Perl. Устанавливая язык, необходимо указать имя plperlu, чтобы выбрать недоверенную вариацию PL/Perl.

Автор функции на PL/PerlU должен позаботиться о том, чтобы эту функцию нельзя было использовать не по назначению, так как она может делать всё, что может пользователь с правами администратора баз данных. СУБД позволяет создавать функции на недоверенных языках только суперпользователям базы данных.

Если показанная выше функция будет создана суперпользователем, и при этом будет выбран язык plperlu, она выполнится успешно.

Таким же образом, в анонимном блоке кода на Perl разрешены абсолютно любые операции, если в качестве языка вместо plperl выбирается plperlu, но выполнять этот код должен суперпользователь.

2886. Триггеры на PL/Perl

PL/Perl можно использовать для написания триггерных функций. В триггерной функции хеш– массив $_TD содержит информацию о произошедшем событии триггера. $_TD – глобальная переменная, которая получает нужное локальное значение при каждом вызове триггера. Хеш–массив $_TD содержит следующие поля:

– $_TD->{new}{foo} – Новое значение столбца foo.

– $_TD->{old}{foo} – Старое значение столбца foo.

– $_TD->{name} – Имя вызываемого триггера.

– $_TD->{event} – Событие триггера: INSERT, UPDATE, DELETE, TRUNCATE или UNKNOWN.

– $_TD->{when} – Когда вызывается триггер: BEFORE (ДО), AFTER (ПОСЛЕ), INSTEAD OF (ВМЕСТО) или UNKNOWN (НЕИЗВЕСТНО).

– $_TD->{level} – Уровень триггера: ROW (СТРОКА), STATEMENT (ОПЕРАТОР) или UNKNOWN (НЕИЗВЕСТНЫЙ).

– $_TD->{relid} – OID таблицы, для которой сработал триггер.

– $_TD->{table_name} – Имя таблицы, для которой сработал триггер.

– $_TD->{relname} – Имя таблицы, для которой сработал триггер.

– $_TD->{table_schema} – Имя схемы, содержащей таблицу, для которой сработал триггер.

– $_TD->{argc} – Число аргументов в триггерной функции.

– @{$_TD->{args}} – Аргументы триггерной функции. Не определено, если $_TD->{argc} равно 0.

В триггерах уровня строки возможны следующие варианты возврата:

– return – Выполнить операцию

– "SKIP" – Не выполнять операцию

– "MODIFY" – Указывает, что строка NEW была изменена триггерной функцией

2887. Событийные триггеры на PL/Perl

PL/Perl можно использовать для написания функций событийных триггеров. В функции событийного триггера хеш-массив $_TD содержит информацию о произошедшем событии триггера.

$_TD – глобальная переменная, которая получает нужное локальное значение при каждом вызове триггера.

Хеш-массив $_TD содержит следующие поля:

– $_TD->{event} – Имя события, при котором срабатывает этот триггер.

– $_TD->{tag} – Тег команды, для которой срабатывает этот триггер.

Возвращаемое значение триггерной функции игнорируется.

2888. Конфигурирование PL/Perl

Параметры конфигурации, влияющие на работу PL/Perl:

```
plperl.on_init (string)
```

Задаёт код Perl, который будет выполняться при первой инициализации интерпретатора Perl, до того, как он получает специализацию plperl или plperlu. Когда этот код выполняется, функции SPI ещё не доступны. Если выполнение кода завершается ошибкой, инициализация интерпретатора прерывается и ошибка распространяется в вызывающий запрос, в результате чего текущая транзакция или подтранзакция прерывается.

Размер этого кода ограничивается одной строкой. Более объёмный код можно поместить в модуль и загрузить этот модуль в строке on_init.

Любые модули, загруженные в plperl.on_init, явно или неявно, будут доступны для использования в коде на языке plperl. Это может создать угрозу безопасности. Чтобы определить, какие модули были загружены, можно выполнить:

```
DO 'elog(WARNING, join ", ", sort keys %INC)' LANGUAGE plperl;
```

Если библиотека plperl включена в shared_preload_libraries, инициализация произойдёт в главном процессе (postmaster) и в этом случае необходимо очень серьёзно оценить риск нарушения работоспособности этого процесса. Основной смысл использовать эту возможность в том, чтобы модули Perl, подключаемые в plperl.on_init, загружались только при запуске главного процесса, и это исключало бы издержки загрузки для отдельных сеансов. Однако эти издержки исключаются только при загрузке в сеансе первого интерпретатора Perl – будь то PL/PerlU или PL/Perl для первой SQL-роли, вызывающей функцию на PL/Perl. Любые дополнительные интерпретаторы Perl, создаваемые в сеансе базы данных, должны будут выполнять plperl.on_init заново.

Задать этот параметр можно только в postgresql.conf или в командной строке при запуске сервера.

```
plperl.on_plperl_init (string)
plperl.on_plperlu_init (string)
```

В этих параметрах задаётся код Perl, который будет выполняться в момент, когда интерпретатор Perl получает специализацию plperl или plperlu, соответственно. Это произойдёт, когда в рамках сеанса будет первый раз вызвана функция на PL/Perl или PL/PerlU, либо когда потребуется дополнительный интерпретатор при использовании другого языка или при вызове

функции PL/Perl новой SQL-ролью. Этот код выполняется после инициализации, произведённой в plperl.on_init. Однако функции SPI в момент исполнения этого кода ещё не доступны. Код в plperl.on_plperl_init запускается после того, как интерпретатор «помещается под замок», так что в нём разрешаются только доверенные операции.

Если этот код завершается ошибкой, инициализация прерывается и ошибка распространяется в вызывающий запрос, что приводит к прерыванию текущей транзакции или подтранзакции. При этом любые действия, уже произведённые в Perl, не будут отменены; однако использоваться этот интерпретатор больше не будет. При следующей попытке использовать этот язык система попытается заново инициализировать свежий интерпретатор Perl.

Изменять эти параметры разрешено только суперпользователям. Хотя изменить их можно в рамках сеанса, такие изменения не повлияют на работу интерпретаторов Perl, задействованных для выполнения функций ранее.

`plperl.use_strict (boolean)`

При значении, равном true, последующая компиляция функций PL/Perl будет выполняться с включённым указанием strict. Этот параметр не влияет на функции, уже скомпилированные в текущем сеансе.

289. PL/Python – процедурный язык Python

Процедурный язык PL/Python позволяет писать функции и процедуры PG360 на языке Python.

Чтобы установить PL/Python в определённую базу данных, необходимо выполнить команду CREATE EXTENSION plpythonu .

PL/Python представлен только в виде «недоверенного» языка, что означает, что он никаким способом не ограничивает действия пользователей, и поэтому он называется plpythonu. Автор функции на недоверенном языке PL/Python должен позаботиться о том, чтобы эту функцию нельзя было использовать не по назначению, так как она может делать всё, что может пользователь с правами администратора баз данных. Создавать функции на недоверенных языках, таких как plpythonu, разрешено только суперпользователям.

289.1. Python 2 и Python 3

PL/Python поддерживает две вариации языка: Python 2 и Python 3. Так как языки Python 2 и Python 3 несовместимы в некоторых важных аспектах, во избежание смешения их в PL/Python применяется следующая схема именования:

Язык PG360 с именем plpython2u представляет реализацию PL/Python, основанную на вариации языка Python 2.

Язык PG360 с именем plpython3u представляет реализацию PL/Python, основанную на вариации языка Python 3.

Язык с именем plpythonu представляет реализацию PL/Python, основанную на версии Python по умолчанию, в данный момент это Python 2. (Этот выбор по умолчанию не зависит от того, какая версия считается локальной версией «по умолчанию», например, на какую версию указывает /usr/bin/python.)

Использовать PL/Python на базе Python 2 и PL/Python на базе Python 3 в одном сеансе нельзя, так как это приведёт к конфликту символов в динамических модулях, что может повлечь сбой серверного процесса PG360. В системе есть проверка, предотвращающая смешение основных версий Python в одном сеансе, которая прервёт сеанс при выявлении расхождения. Однако использовать обе вариации в одной базе данных всё же возможно, обращаясь к ним в разных сеансах.

2892. Функции на PL/Python

Функции на PL/Python объявляются стандартным образом с помощью команды CREATE FUNCTION:

```
CREATE FUNCTION имя_функции (аргументы) RETURNS возвращаемый_тип
AS $$
# Тело функции на PL/Python
$$ LANGUAGE plpythonu;
```

Тело функции содержит просто скрипт на языке Python. Когда вызывается функция, её аргументы передаются в виде элементов списка args; именованные аргументы также передаются скрипту Python как обычные переменные. С именованными аргументами скрипт обычно лучше читается. Результат из кода Python возвращается обычным способом: командой return или yield (в случае функции, возвращающей множество). Если возвращаемое значение не определено, Python возвращает None. Исполнитель PL/Python преобразует None языка Python в значение NULL языка SQL. В процедуре код Python должен возвращать None (обычно для этого процедура завершается без оператора return или используется оператор return без аргумента); в противном случае выдаётся ошибка.

Значения аргументов задаются в глобальных переменных. Согласно правилам видимости в Python, тонким следствием этого является то, что переменной аргумента нельзя присвоить внутри функции выражение, включающее имя самой этой переменной, если только эта переменная не объявлена глобальной в текущем блоке.

2893. Сопоставление типов данных

Когда вызывается функция PL/Python, её аргументы преобразуются из типа PG360 в соответствующий тип Python по таким правилам:

– Тип PG360 boolean преобразуется в тип bool языка Python.

– Типы PG360 smallint и int преобразуются в тип int языка Python. Типы PG360 bigint и oid становятся типами long в Python 2 и int в Python 3.

– Типы PG360 real и double преобразуются в тип float языка Python.

– Тип PG360 numeric преобразуется в тип Decimal среды Python. Этот тип импортируется из пакета cdecimal, при его наличии. В противном случае используется decimal.Decimal из стандартной библиотеки. Тип cdecimal работает значительно быстрее, чем decimal. Однако в Python версии 3.3 и выше тип cdecimal включается в стандартную библиотеку под именем decimal, так что теперь этого различия нет.

– Тип PG360 bytea становится типом str в Python 2 и bytes в Python 3. В Python 2 такую строку следует воспринимать как последовательность байт без какой-либо определённой кодировки символов.

– Все другие типы данных, включая типы символьных строк PG360, преобразуются в тип str языка Python. В Python 2 эта строка будет передаваться в кодировке сервера PG360; в Python 3 это будет строка в Unicode, как и все строки.

– Информация о нескалярных типах данных приведена ниже.

При завершении функции PL/Python её значение результата преобразуется в тип данных, объявленный как тип результата в PG360, следующим образом:

– Когда тип результата функции в PG360 – boolean, возвращаемое значение приводится к логическому типу по правилам, принятым в *Python*. То есть false будет возвращено для 0 и пустой строки, но, для 'f' будет возвращено true.

– Когда тип результата функции PG360 – bytea, возвращаемое значение будет преобразовано в строку (Python 2) или набор байт (Python 3), используя встроенные средства Python, а затем будет приведено к типу bytea.

– Для всех других типов результата PG360 возвращаемое значение преобразуется в строку с помощью встроенной в Python функции str, и полученная строка передаётся функции ввода типа данных PG360. (Если значение в Python имеет тип float, оно преобразуется встроенной функцией repr, а не str, для недопущения потери точности.)

Из кода Python 2 строки должны передаваться в PG360 в кодировке сервера PG360. При передаче строки, неприемлемой для текущей кодировки сервера, возникает ошибка, но не все несоответствия кодировки могут быть выявлены, так что с некорректной кодировкой всё же могут быть получены нечитаемые строки. Строки Unicode переводятся в нужную кодировку автоматически, так что использовать их может быть безопаснее и удобнее. В Python 3 все строки имеют кодировку Unicode.

– Информация о нескалярных типах данных приведена ниже.

Логические несоответствия между объявленным в PG360 типом результата и типом фактически возвращаемого объекта Python игнорируются – значение преобразуется в любом случае.

2894. Массивы, списки

Значения массивов SQL передаются в PL/Python в виде списка Python.

Многомерные массивы передаются в PL/Python в виде вложенных списков Python. Например, двухмерный массив представляется как список списков. При передаче многомерного массива SQL из функции PL/Python необходимо, чтобы все внутренние списки на каждом уровне имели одинаковый размер.

2895. Триггерные функции

Когда функция используется как триггер, словарь TD содержит значения, связанные с работой триггера:

– TD["event"] – содержит название события в виде строки: INSERT, UPDATE, DELETE или TRUNCATE.

– TD["when"] – содержит одну из строк: BEFORE, AFTER или INSTEAD OF.

– TD["level"] – содержит ROW или STATEMENT.

– TD["new"],

– TD["old"] – Для триггера уровня строки одно или оба этих поля содержат соответствующие строки триггера, в зависимости от события триггера.

– TD["name"] – содержит имя триггера.

– TD["table_name"] – содержит имя таблицы, для которой сработал триггер.

– TD["table_schema"] – содержит схему таблицы, для которой сработал триггер.

– TD["relid"] – содержит OID таблицы, для которой сработал триггер.

– TD["args"] – Если в команде CREATE TRIGGER задавались аргументы, их можно получить как элементы массива с TD["args"][0] по TD["args"][*n*-1].

2896. Обращение к базе данных

Исполнитель языка PL/Python автоматически импортирует модуль Python с именем plpy. Можно использовать функции и константы, объявленные в этом модуле, обращаясь к ним по именам вида plpy.*имя*.

28961. Функции обращения к базе данных

Модуль plpy содержит различные функции для выполнения команд в базе данных:

`plpy.execute(`*`запрос`* `[,` *`макс-строк`*`])`

При вызове plpy.execute со строкой запроса и необязательным аргументом, ограничивающим число строк, выполняется заданный запрос, а то, что он выдаёт, возвращается в виде объекта результата.

Объект результата имитирует список или словарь. Получить из него данные можно по номеру строки и имени столбца.

Число возвращённых в этом объекте строк можно получить, воспользовавшись встроенной функцией len.

Для объекта результата определены следующие дополнительные методы:

- `nrows()` – Возвращает число строк, обработанных командой. Это число не обязательно будет равно числу возвращённых строк. Например, команда UPDATE устанавливает это значение, но не возвращает строк (без указания RETURNING).
- `status()` – Значение состояния, возвращённое SPI_execute().
- `colnames()`,
- `coltypes()`,
- `coltypmods()` – Возвращают список имён столбцов, список OID типов столбцов и список модификаторов типа этих столбцов, соответственно.

Эти методы вызывают исключение, когда им передаётся объект, полученный от команды, не возвращающей результирующий набор, например, UPDATE без RETURNING, либо DROP TABLE.

- `__str__()` Стандартный метод __str__ определён так, чтобы можно было, например, вывести отладочное сообщение с результатами запроса, вызвав plpy.debug(rv).

Объект результата может быть изменён.

При вызове plpy.execute весь набор результатов будет прочитан в память. Эту функцию следует использовать, только если набор будет относительно небольшим. Чтобы исключить риск переполнения памяти при выборке результатов большого объёма, необходимо используйть plpy.cursor вместо plpy.execute.

```
plpy.prepare(запрос [, типы_аргументов])
plpy.execute(план [, аргументы [, макс-строк]])
```

Функция plpy.prepare подготавливает план выполнения для запроса. Она вызывается со строкой запроса и списком типов параметров (если в запросе есть параметры).

Чтобы запустить подготовленный оператор на выполнение, необходимо использовать вариацию функции plpy.execute:

```
rv = plpy.execute(plan, ["name"], 5)
plpy.cursor(запрос)
plpy.cursor(план [, аргументы])
```

Функция plpy.cursor принимает те же аргументы, что и plpy.execute (кроме ограничения строк) и возвращает объект курсора, который позволяет обрабатывать объёмные наборы результатов небольшими порциями. Как и plpy.execute, этой функции можно передать строку запроса или объект плана со списком аргументов, а можно вызывать функцию cursor как метод объекта плана.

Объект курсора реализует метод fetch, который принимает целочисленный параметр и возвращает объект результата. При каждом следующем вызове fetch возвращаемый объект будет содержать следующий набор строк, в количестве, не превышающем значение параметра. Когда строки закончатся, fetch начнёт возвращать пустой объект результата. Объекты курсора также предоставляют *интерфейс итератора*, выдающий по строке за один раз, пока не будут выданы все строки. Данные, выбираемые таким образом, возвращаются не как объекты результата, а как словари (одной строке результата соответствует один словарь).

Курсоры ликвидируются автоматически. Но если нужно явно освободить все ресурсы, занятые курсором, то необходимо вызвать метод close. Продолжать получать данные через курсор, который был закрыт, нельзя.

28.9.6.2. Обработка ошибок

Функции, обращающиеся к базе данных, могут сталкиваться с ошибками, в результате которых они будут прерываться и вызывать исключение. Обе функции plpy.execute и plpy.prepare могут вызывать экземпляр подкласса исключения plpy.SPIError, которое по умолчание прекращает выполнение функции. Эту ошибку можно обработать, как и любое другое исключение в Python, применив конструкцию try/except.

28.9.7. Управление транзакциями

В процедуре, которая вызывается в коде верхнего уровня или в анонимном блоке кода (в команде DO), можно управлять транзакциями. Чтобы зафиксировать текущую транзакцию, необходимо вызвать plpy.commit(), а чтобы откатить – plpy.rollback(). (Выполнить SQL-команды COMMIT или ROLLBACK через plpy.execute или подобную функцию нельзя. Соответствующие

операции могут выполняться только данными функциями.) После завершения одной транзакции следующая начинается автоматически, отдельной функции для этого нет.

Транзакцию нельзя завершить в случае существования открытой явной подтранзакции.

2898. Вспомогательные функции

Модуль plpy также предоставляет функции:

```
plpy.debug( msg, **kwargs )
plpy.log( msg, **kwargs )
plpy.info( msg, **kwargs )
plpy.notice( msg, **kwargs )
plpy.warning( msg, **kwargs )
plpy.error( msg, **kwargs ) plpy.fatal( msg, **kwargs )
```

Функции plpy.error и plpy.fatal выдают исключение Python, которое, если его не перехватить, распространяется в вызывающий запрос, что приводит к прерыванию текущей транзакции или подтранзакции. Команды raise plpy.Error(*msg*) и raise plpy.Fatal(*msg*) равнозначны вызовам plpy.error(*msg*) и plpy.fatal(*msg*), соответственно, но форма raise не позволяет передавать аргументы с ключами. Другие функции просто выдают сообщения разных уровней важности. Будут ли сообщения определённого уровня передаваться клиентам и/или записываться в журнал сервера, определяется конфигурационными переменными log_min_messages и client_min_messages.

Аргумент *msg* задаётся как позиционный.

Дополнительно только по ключам принимаются следующие аргументы:

– detail;

– hint;

– sqlstate;

– schema_name;

– table_name;

– column_name;

– datatype_name;

– constraint_name.

Ещё один набор вспомогательных функций:

```
plpy.quote_literal(строка),
plpy.quote_nullable(строка),
plpy.quote_ident(строка)
```

– Они равнозначны встроенным функциям заключения в кавычки. Они используются при конструировании свободно составляемых запросов.

2899. Переменные окружения

Некоторые переменные окружения, воспринимаемые интерпретатором Python, тоже могут влиять на поведение PL/Python. При необходимости их нужно установить в среде основного серверного процесса PG360, например, в скрипте запуска.

Переменные окружения, влияющие на PL/Python:

– PYTHONHOME;

– PYTHONPATH;

– PYTHONY2K;

– PYTHONOPTIMIZE;
– PYTHONDEBUG;
– PYTHONVERBOSE;
– PYTHONCASEOK;
– PYTHONDONTWRITEBYTECODE;
– PYTHONIOENCODING;
– PYTHONUSERBASE;
– PYTHONHASHSEED.

28.10. Интерфейс программирования сервера

Интерфейс программирования сервера (SPI, Server Programming Interface) даёт разработчикам пользовательских функций на C возможность запускать команды SQL из своих функций или процедур. SPI представляет собой набор интерфейсных функций, упрощающих доступ к анализатору, планировщику и исполнителю запросов. В SPI есть также функции для управления памятью.

Необходимо учесть, что если команда, вызванная через SPI, прерывается ошибкой, управление не возвращается в функцию на C. Вместо этого происходит откат транзакции или подтранзакции, из которой вызывалась функция. (Это может показаться удивительным, с учётом того, что для большинства функций SPI описаны соглашения по возврату ошибок. Однако эти соглашения применимы только к ошибкам, выявляемым в самих функциях SPI.) Получить управление после ошибки можно, только организовав собственную подтранзакцию, окружающую вызовы SPI, в которых возможна ошибка.

Функции SPI выдают неотрицательный результат в случае успеха (либо через возвращаемое целочисленное значение, либо в глобальной переменной SPI_result, как описано ниже). В случае ошибки выдаётся отрицательный результат или NULL.

Файлы исходного кода, использующие SPI, должны включать заголовочный файл executor/spi.h.

28.10.1. Интерфейсные функции:

– SPI_connect, SPI_connect_ext — подключить функцию на C к менеджеру SPI.

Синтаксис:

```
int SPI_connect(void)
int SPI_connect_ext(int options)
```

Описание:

SPI_connect устанавливает подключение вызова функции на C к менеджеру SPI. Данную функцию необходимо использовать, если необходимо выполнять команды через SPI. Некоторые вспомогательные функции SPI могут вызываться из неподключённых функций.

SPI_connect_ext делает то же самое, но принимает один аргумент, через который можно передать дополнительные флаги. Поддерживаются следующие флаги:

-SPI_OPT_NONATOMIC – Переводит подключение SPI в *неатомарный* режим, в котором разрешаются вызовы функций управления транзакциями (SPI_commit, SPI_rollback). В обычном режиме вызов этих функций приводит к немедленной ошибке.

Вызов SPI_connect() равнозначен SPI_connect_ext(0).

Возвращаемое значение:

SPI_OK_CONNECT – при успехе.

SPI_ERROR_CONNECT – при ошибке.

– SPI_finish – отключить функцию на C от менеджера SPI.

Синтаксис:

```
int SPI_finish(void)
```

Описание:

SPI_finish закрывает текущее соединение с менеджером SPI. Эту функцию необходимо вызывать после завершения операций SPI, которые должны выполняться в текущем вызове функции на C. Однако при прерывании транзакции, выполняя elog(ERROR), о закрытии соединения можно не беспокоиться. В этом случае SPI произведёт очистку автоматически.

Возвращаемое значение:

SPI_OK_FINISH – если отключение выполнено корректно.

SPI_ERROR_UNCONNECTED – если вызывается из неподключённой функции на C

– SPI_execute — выполнить команду.

Синтаксис:

```
int SPI_execute(const char * command, bool read_only, long count)
```

Описание:

SPI_execute выполняет заданную команду SQL для получения строк в количестве, ограниченном *count*. С параметром *read_only*, равным true, команда должна только читать данные; это несколько сокращает издержки на её выполнение.

Эту функцию можно вызывать только из подключённой функции на C.

Если *count* равен 0, команда выполняется для всех строк, к которым она применима. Если *count* больше нуля, будет получено не более чем *count* строк; выполнение команды остановится при достижении этого предела, практически так же, как и с предложением LIMIT в запросе.

В одной строке можно передать несколько команд; SPI_execute возвращает результат команды, выполненной последней. Параметр *count* при этом будет применяться к каждой команде по отдельности (несмотря даже на то, что возвращён будет только последний результат). Это ограничение не будет распространяться на скрытые команды, генерируемые правилами.

Когда параметр *read_only* равен false, SPI_execute увеличивает счётчик команд и получает новый *снимок* перед выполнением каждой очередной команды в строке. Этот снимок фактически не меняется при текущем уровне изоляции транзакций SERIALIZABLE или REPEATABLE READ, но в режиме READ COMMITTED после обновления снимка очередная команда может видеть результаты только что зафиксированных транзакций из других сеансов. Это важно для согласованного поведения, когда команды модифицируют базу данных.

Когда параметр *read_only* равен true, SPI_execute не обновляет снимок и не увеличивает счётчик команд, и допускает в строке команд только SELECT. Заданные команды выполняются со снимком, ранее полученным для окружающего запроса. Этот режим выполнения несколько быстрее режима чтения/записи вследствие исключения издержек, связанных с отдельными

командами. Он также позволяет создавать подлинно *стабильные* функции: так как последующие вызовы в транзакции будут использовать один снимок, результаты команд не изменятся.

Смешивать команды, только читающие, с командами, читающими и пишущими, в одной процедуре, использующей SPI, обычно неразумно; запросы только на чтение не увидят результатов изменений в базе данных, произведённых пишущими запросами.

Число строк, которые были фактически обработаны командой (последней), возвращается в глобальной переменной SPI_processed. Если эта функция возвращает значение SPI_OK_SELECT, SPI_OK_INSERT_RETURNING, SPI_OK_DELETE_RETURNING или SPI_OK_UPDATE_RETURNING, можно обратиться по глобальному указателю SPITupleTable *SPI_tuptable и прочитать строки результата. Некоторые служебные команды (например, EXPLAIN) также возвращают наборы строк, и SPI_tuptable будет содержать их результаты и в этих случаях. Другие вспомогательные команды (COPY, CREATE TABLE AS) не возвращают набор строк, так что указатель SPI_tuptable равен NULL, но они так же возвращают число обработанных строк в SPI_processed.

Структура SPITupleTable определена следующим образом:

```
typedef struct SPITupleTable
{
/* Открытые члены */
  TupleDesc   tupdesc;      /*    дескриптор кортежа */
  HeapTuple   *vals;        /*    массив кортежей */
  uint64      numvals;      /*    число фактически представленных кортежей */

/* Закрытые члены, не предназначенные для внешнего использования */
uint64 alloced;         /* зарезервированное в памяти число элементов vals */
MemoryContext tuptabcxt;    /* контекст таблицы результатов в памяти */
slist_node next;        /* ссылка для внутреннего обслуживания */
SubTransactionId subid;     /* подтранзакция, создавшая структуру tuptable */
} SPITupleTable;
```

Поля tupdesc, vals и numvals могут использоваться кодом, вызывающим SPI, остальные поля являются внутренними. vals представляет собой массив указателей на кортежи. Число записей в нём указывается в numvals (по некоторым историческим причинам это число также возвращается в SPI_processed). Поле tupdesc содержит дескриптор кортежа, который можно передать функциям SPI, работающими с кортежами.

SPI_finish освобождает все структуры SPITupleTable, размещённые в памяти для текущей функции на C. Можно освободить структуру конкретной результирующей таблицы, вызвав SPI_freetuptable.

Аргументы:

– const char * *command* – строка с командой, которая должна быть выполнена.

– bool *read_only* – true для режима выполнения «только чтение».

– long *count* – максимальное число строк, которое должно быть возвращено; с 0 ограничения нет.

Возвращаемое значение:

Если команда была выполнена успешно, возвращается одно из следующих (неотрицательных) значений:

SPI_OK_SELECT – если выполнялась команда SELECT (но не SELECT INTO).

SPI_OK_SELINTO – если выполнялась команда SELECT INTO.

SPI_OK_INSERT – если выполнялась команда INSERT.

SPI_OK_DELETE – если выполнялась команда DELETE.

SPI_OK_UPDATE – если выполнялась команда UPDATE.

SPI_OK_INSERT_RETURNING – если выполнялась команда INSERT RETURNING.

SPI_OK_DELETE_RETURNING – если выполнялась команда DELETE RETURNING.

SPI_OK_UPDATE_RETURNING – если выполнялась команда UPDATE RETURNING.

SPI_OK_UTILITY – если выполнялась служебная команда (например, CREATE TABLE).

SPI_OK_REWRITTEN – если команда была преобразована правилом в команду другого вида (например, UPDATE стал командой INSERT).

В случае ошибки возвращается одно из следующих отрицательных значений:

SPI_ERROR_ARGUMENT – если в качестве *command* передан NULL или *count* меньше 0.

SPI_ERROR_COPY – при попытке выполнить COPY TO stdout или COPY FROM stdin.

SPI_ERROR_TRANSACTION – при попытке выполнить команду управления транзакциями (BEGIN, COMMIT, ROLLBACK, SAVEPOINT, PREPARE TRANSACTION, COMMIT PREPARED, ROLLBACK PREPARED или любую их вариацию).

SPI_ERROR_OPUNKNOWN – если тип команды неизвестен (такого быть не должно).

SPI_ERROR_UNCONNECTED – если вызывается из неподключённой функции на C.

– SPI_exec — выполнить команду чтения/записи.

Синтаксис:

```
int SPI_exec(const char * command, long count)
```

Описание:

SPI_exec действует подобно SPI_execute, но ей не передаётся параметр *read_only* (всегда подразумевается false).

Аргументы:

– const char * *command* – строка с командой, которая должна быть выполнена.

– long *count* – максимальное число строк, которое должно быть возвращено; с 0 ограничения нет.

Возвращаемое значение:

Возвращаемые значения те же, что и у SPI_execute.

– SPI_execute_with_args — выполнить команду с выделенными параметрами.

Синтаксис:

```
int SPI_execute_with_args(const char *command,
int nargs, Oid *argtypes,
Datum *values, const char *nulls, bool read_only, long count)
```

Описание:

SPI_execute_with_args выполняет команду, которая может включать ссылки на параметры, передаваемые извне. В тексте команды параметры обозначаются символами $*n*, а в вызове указываются типы данных и значения для каждого такого символа. Параметры *read_only* и *count* имеют тот же смысл, что и в SPI_execute.

Основное преимущество этой функции по сравнению с SPI_execute в том, что она позволяет передавать в команду значения данных, не требуя кропотливой подготовки строк, и таким образом сокращает риск атак с SQL-инъекцией.

Подобного результата можно достичь, вызвав SPI_prepare и затем SPI_execute_plan; однако с данной функцией план запроса всегда подстраивается под переданные конкретные значения параметров. Поэтому для разового выполнения запроса рекомендуется применять эту функцию. Если же одна и та же команда должна выполняться с самыми разными параметрами, какой вариант окажется быстрее, будет зависеть от стоимости повторного планирования и выигрыша от выбора специализированных планов.

Аргументы:

– const char * *command* – строка команды.

– int *nargs* – число входных параметров ($1, $2 и т. д.).

– Oid * *argtypes* – массив размера *nargs*, содержащий OID типов параметров.

– Datum * *values* – массив размера *nargs*, содержащий фактические значения параметров.

– const char * *nulls* – массив размера *nargs*, описывающий, в каких параметрах передаётся NULL.

Если в *nulls* передаётся NULL, SPI_execute_with_args считает, что ни один из параметров не равен NULL. В противном случае элемент массива *nulls* должен содержать ' ', если значение соответствующего параметра не NULL, либо 'n', если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *values*, не учитывается.) *nulls* — это не текстовая строка, а просто массив: ноль ('\0') в конце не нужен.

bool *read_only* – true для режима выполнения «только чтение».

long *count* – максимальное число строк, которое должно быть возвращено; с 0 ограничения нет.

Возвращаемое значение:

Возвращаемые значения те же, что и у SPI_execute.

Переменные SPI_processed и SPI_tuptable устанавливаются как в SPI_execute, если вызов был успешным.

– SPI_prepare — подготовить оператор, но пока не выполнять его.

Синтаксис:

`SPIPlanPtr SPI_prepare(const char * command, int nargs, Oid * argtypes)`

Описание:

SPI_prepare создаёт и возвращает подготовленный оператор для заданной команды. Подготовленный оператор может быть затем неоднократно выполнен функцией SPI_execute_plan.

Когда одна и та же или похожие команды выполняются неоднократно, обычно выгоднее произвести анализ запроса только раз, а ещё выгоднее может быть повторно использовать план

выполнения команды. SPI_prepare преобразует строку команды в подготовленный оператор, включающий в себя результаты анализа запроса. Подготовленный оператор также оставляет место для кеширования плана выполнения, если выбор специализированного плана для каждого выполнения не принесёт пользы.

Подготавливаемую команду можно сделать более общей, записав параметры ($1, $2, etc.) вместо значений, задаваемыми константами в обычной команде. Фактические значения параметров в этом случае будут задаваться при вызове SPI_execute_plan. Это позволяет применять подготовленную команду в более широком круге ситуаций, чем это возможно без параметров.

Оператор, возвращаемый функцией SPI_prepare, может использоваться только в текущем вызове функции на C, так как SPI_finish освобождает память, выделенную для такого оператора. Но этот оператор может быть сохранён на будущее с помощью функций SPI_keepplan или SPI_saveplan.

Аргументы:

– const char * *command* – строка команды.

– int *nargs* – число входных параметров ($1, $2 и т. д.).

– Oid * *argtypes* – указатель на массив, содержащий OID типов параметров.

Возвращаемое значение:

SPI_prepare возвращает ненулевой указатель на SPIPlan, скрытую структуру, представляющую подготовленный оператор. В случае ошибки возвращается NULL, а в SPI_result устанавливается один из кодов ошибок, определённых для SPI_execute, за исключением того, что код SPI_ERROR_ARGUMENT устанавливается, когда *command* — NULL, когда *nargs* меньше 0 или когда *nargs* больше 0, а *argtypes* — NULL.

Эту функцию следует вызывать только из подключённой функции на C.

– SPI_prepare_cursor — подготовить оператор, но пока не выполнять его.

Синтаксис:

```
SPIPlanPtr SPI_prepare_cursor(const char * command, int nargs,
Oid * argtypes, int cursorOptions)
```

Описание:

Функция SPI_prepare_cursor равнозначна SPI_prepare, за исключением того, что ей можно передать «параметры курсора». Эти параметры задаются битовой маской со значениями, определёнными в nodes/parsenodes.h для поля options структуры DeclareCursorStmt. SPI_prepare подразумевает, что эти параметры всегда нулевые.

Аргументы:

– const char * *command* – строка команды.

– int *nargs* – число входных параметров ($1, $2 и т. д.).

– Oid * *argtypes* – указатель на массив, содержащий OID типов параметров.

– int *cursorOptions* – битовая маска параметров курсора; 0 выбирает поведение по умолчанию.

Возвращаемое значение:

SPI_prepare_cursor возвращает результат по тем же соглашениям, что и SPI_prepare.

– SPI_prepare_params — подготовить оператор, но пока не выполнять его.

Синтаксис:

```
SPIPlanPtr SPI_prepare_params(const char * command,
ParserSetupHook parserSetup, void * parserSetupArg,
int cursorOptions)
```

Описание:

SPI_prepare_params создаёт и возвращает подготовленный оператор для заданной команды, но не выполняет саму команду. Эта функция равнозначна SPI_prepare_cursor, но позволяет вызывающему дополнительно установить функции-обработчики для управления разбором ссылок на внешние параметры.

Аргументы:

– const char * *command* – строка команды.

– ParserSetupHook *parserSetup* – Функция настройки обработчиков разбора.

– void * *parserSetupArg* – аргумент для сквозной передачи в *parserSetup.*

– int *cursorOptions* – битовая маска параметров курсора; 0 выбирает поведение по умолчанию.

Возвращаемое значение:

SPI_prepare_params возвращает результат по тем же соглашениям, что и SPI_prepare.

– SPI_getargcount — получить число аргументов, требующихся оператору, подготовленному функцией SPI_prepare.

Синтаксис:

```
int SPI_getargcount(SPIPlanPtr plan)
```

Описание:

SPI_getargcount возвращает число аргументов, требующихся для выполнения оператора, подготовленного функцией SPI_prepare.

Аргументы:

– SPIPlanPtr *plan* – подготовленный оператор (возвращаемый функцией SPI_prepare).

Возвращаемое значение:

Число аргументов, которое ожидает план, заданный параметром *plan*. Если значение *plan* неверное или NULL, в SPI_result устанавливается код SPI_ERROR_ARGUMENT, а функция возвращает -1.

– SPI_getargtypeid — получить OID типа аргумента для оператора, подготовленного функцией SPI_prepare.

Синтаксис:

```
Oid SPI_getargtypeid(SPIPlanPtr plan, int argIndex)
```

Описание:

SPI_getargtypeid возвращает OID, представляющий тип аргумента под номером *argIndex* оператора, подготовленного функцией SPI_prepare. Первый аргумент идёт под номером ноль.

Аргументы:

– SPIPlanPtr *plan* – подготовленный оператор (возвращаемый функцией SPI_prepare).

– int *argIndex* – индекс аргумента, начиная с нуля.

Возвращаемое значение:

OID типа аргумента с заданным индексом. Если значение *plan* неверное или NULL, либо *argIndex* меньше 0 или не меньше числа аргументов, объявленных при подготовке плана (передаваемого в *plan*), в SPI_result устанавливается SPI_ERROR_ARGUMENT и возвращается InvalidOid.

– SPI_is_cursor_plan — выдать true, если оператор, подготовленный функцией SPI_prepare, можно использовать с SPI_cursor_open.

Синтаксис:

```
bool SPI_is_cursor_plan(SPIPlanPtr plan)
```

Описание:

SPI_is_cursor_plan возвращает true, если оператор, подготовленный функцией SPI_prepare, можно передать в качестве аргумента SPI_cursor_open, или false в противном случае. Для положительного ответа в *plan* должна быть представлена одна команда, и эта команда должна возвращать кортежи; например, SELECT может быть подходящей командой, если он не содержит предложения INTO, а UPDATE подходит, только если он содержит предложение RETURNING.

Аргументы:

– SPIPlanPtr *plan* – подготовленный оператор (возвращаемый функцией SPI_prepare).

Возвращаемое значение:

Значение true или false, показывающее, можно ли для подготовленного оператора, заданного параметром *plan*, получить курсор, при SPI_result равном нулю. Если дать ответ невозможно (например, если значение *plan* неверное или NULL, либо вызывающий не подключён к SPI), в SPI_result устанавливается соответствующий код ошибки и возвращается false.

SPI_execute_plan — выполнить оператор, подготовленный функцией SPI_prepare.

Синтаксис:

```
int SPI_execute_plan(SPIPlanPtr plan, Datum * values, const char * nulls, bool read_only, long count)
```

Описание:

SPI_execute_plan выполняет оператор, подготовленный функцией SPI_prepare или родственными ей. Параметры *read_only* и *count* имеют тот же смысл, что и в SPI_execute.

Аргументы:

– SPIPlanPtr *plan* – подготовленный оператор (возвращаемый функцией SPI_prepare).

– Datum * *values* – Массив фактических значений параметров. Его размер должен равняться числу аргументов оператора.

– const char * *nulls* – Массив, описывающий, в каких параметрах передаётся NULL. Должен иметь размер, равный числу аргументов оператора.

Если в *nulls* передаётся NULL, SPI_execute_plan считает, что ни один из параметров не равен NULL. В противном случае элемент массива *nulls* должен содержать ' ', если значение соответствующего параметра не NULL, либо 'n', если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *values*, не учитывается.) *nulls* — это не текстовая строка, а просто массив: ноль ('\0') в конце не нужен.

– bool *read_only* – true для режима выполнения «только чтение»

– long *count* – максимальное число строк, которое должно быть возвращено; с 0 ограничения нет

Возвращаемое значение:

Возвращаемые значения те же, что и у SPI_execute, со следующими дополнительными вариантами ошибок (отрицательных результатов):

SPI_ERROR_ARGUMENT – Если *plan* неверный или NULL, либо *count* меньше 0.

SPI_ERROR_PARAM – Если в *values* передан NULL и *plan* был подготовлен с другими параметрами.

Переменные SPI_processed и SPI_tuptable устанавливаются как в SPI_execute, если вызов был успешным.

– SPI_execute_plan_with_paramlist — выполнить оператор, подготовленный функцией SPI_prepare.

Синтаксис:

```
int SPI_execute_plan_with_paramlist(SPIPlanPtr plan,
ParamListInfo params, bool read_only,
long count)
```

Описание:

SPI_execute_plan_with_paramlist выполняет оператор, подготовленный функцией SPI_prepare. Данная функция равнозначна SPI_execute_plan, не считая того, что информация о значениях параметров, передаваемых запросу, представляется по-другому. Представление ParamListInfo может быть удобным для передачи значений, уже имеющих нужный формат. Эта функция также поддерживает динамические наборы параметров, которые реализуются через функции– обработчики, устанавливаемые в ParamListInfo.

Аргументы:

– SPIPlanPtr *plan* – подготовленный оператор (возвращаемый функцией SPI_prepare).

– ParamListInfo *params* – структура данных, содержащая типы и значения параметров; NULL, если их нет.

– bool *read_only* – true для режима выполнения «только чтение».

– long *count* – максимальное число строк, которое должно быть возвращено; с 0 ограничения нет;

Возвращаемое значение:

Возвращаемые значения те же, что и у SPI_execute_plan.

Переменные SPI_processed и SPI_tuptable устанавливаются как в SPI_execute_plan, если вызов был успешным.

– SPI_execp — выполнить оператор в режиме чтения/записи.

Синтаксис:

```
int SPI_execp(SPIPlanPtr plan, Datum * values, const char *
nulls, long count)
```

Описание:

SPI_execp действует подобно SPI_execute_plan, но ей не передаётся параметр *read_only* (всегда подразумевается false).

Аргументы:

– SPIPlanPtr *plan* – подготовленный оператор (возвращаемый функцией SPI_prepare).

– Datum * *values* – Массив фактических значений параметров. Его размер должен равняться числу аргументов оператора.

– const char * *nulls* – Массив, описывающий, в каких параметрах передаётся NULL. Должен иметь размер, равный числу аргументов оператора.

Если в *nulls* передаётся NULL, SPI_execp считает, что ни один из параметров не равен NULL. В противном случае элемент массива *nulls* должен содержать ' ', если значение соответствующего параметра не NULL, либо 'n', если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *values*, не учитывается.) *nulls* — это не текстовая строка, а просто массив: ноль ('\0') в конце не нужен.

– long *count* – максимальное число строк, которое должно быть возвращено; с 0 ограничения нет.

Возвращаемое значение:

Возвращаемые значения те же, что и у SPI_execute_plan.

Переменные SPI_processed и SPI_tuptable устанавливаются как в SPI_execute, если вызов был успешным.

– SPI_cursor_open — открыть курсор для оператора, созданного функцией SPI_prepare.

Синтаксис:

```
Portal SPI_cursor_open(const char * name, SPIPlanPtr plan,
Datum * values, const char * nulls, bool read_only)
```

Описание:

SPI_cursor_open открывает курсор (внутри называемый порталом), через который будет выполняться оператор, подготовленный функцией SPI_prepare. Параметры этой функции имеют тот же смысл, что и соответствующие параметры SPI_execute_plan.

Применение курсора по сравнению с непосредственным выполнением оператора даёт двойную выгоду. Во-первых, строки результата можно получать в небольших количествах, без риска исчерпать всю память при выполнении запросов, возвращающих много строк. Во-вторых, портал может существовать и после завершения текущей функции на C (на самом деле он может просуществовать до конца текущей транзакции). Возвратив имя портала в код, вызывающий функцию на C, можно организовать выдачу результата в виде набора строк.

Переданные значения параметров копируются в портал курсора, так что их можно освободить и во время существования курсора.

Аргументы:

– const char * *name* – имя портала, либо NULL, чтобы имя выбрала система.

– SPIPlanPtr *plan* – подготовленный оператор (возвращаемый функцией SPI_prepare).

– Datum * *values* – Массив фактических значений параметров. Его размер должен равняться числу аргументов оператора.

– const char * *nulls* – Массив, описывающий, в каких параметрах передаётся NULL. Должен иметь размер, равный числу аргументов оператора.

Если в *nulls* передаётся NULL, SPI_cursor_open считает, что ни один из параметров не равен NULL. В противном случае элемент массива *nulls* должен содержать ' ', если значение соответствующего параметра не NULL, либо 'n', если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *values*, не учитывается.) *nulls* — это не текстовая строка, а просто массив: ноль ('\0') в конце не нужен.

– bool *read_only* – true для режима выполнения «только чтение».

Возвращаемое значение:

Указатель на портал, содержащий курсор. Соглашение о возврате ошибок отсутствует; все ошибки выдаются через elog.

– SPI_cursor_open_with_args — открывает курсор для запроса с параметрами.

Синтаксис:

```
Portal SPI_cursor_open_with_args(const char *name,
const char *command,
int nargs, Oid *argtypes,
Datum *values, const char *nulls, bool read_only, int cursorOptions)
```

Описание:

SPI_cursor_open_with_args открывает курсор (внутри называемый порталом) для выполнения заданного запроса. Большинство параметров имеют тот же смысл, что и соответствующие параметры функций SPI_prepare_cursor и SPI_cursor_open.

Для разового выполнения запроса эту функцию следует предпочесть SPI_prepare_cursor с последующей SPI_cursor_open. Если же одна и та же команда должна выполняться с самыми разными параметрами, какой вариант окажется быстрее, будет зависеть от стоимости повторного планирования и выигрыша от выбора специализированных планов.

Переданные значения параметров копируются в портал курсора, так что их можно освободить и во время существования курсора.

Аргументы:

– const char * *name* – имя портала, либо NULL, чтобы имя выбрала система.

– const char * *command* – строка команды.

– int *nargs* – число входных параметров ($1, $2 и т. д.).

– Oid * *argtypes* – массив размера *nargs*, содержащий OID типов параметров.

– Datum * *values* – массив размера *nargs*, содержащий фактические значения параметров.

– const char * *nulls* – массив размера *nargs*, описывающий, в каких параметрах передаётся NULL.

Если в *nulls* передаётся NULL, SPI_cursor_open_with_args считает, что ни один из параметров не равен NULL. В противном случае элемент массива *nulls* должен содержать ' ', если значение соответствующего параметра не NULL, либо 'n', если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *values*, не учитывается.) *nulls* — это не текстовая строка, а просто массив: ноль ('\0') в конце не нужен.

– bool *read_only* – true для режима выполнения «только чтение».

– int *cursorOptions* – битовая маска параметров курсора; 0 выбирает поведение по умолчанию.

Возвращаемое значение:

Указатель на портал, содержащий курсор. Соглашение о возврате ошибок отсутствует; все ошибки выдаются через elog.

– SPI_cursor_open_with_paramlist — открыть курсор с параметрами.

Синтаксис:

```
Portal SPI_cursor_open_with_paramlist(const char *name,
SPIPlanPtr plan, ParamListInfo params, bool read_only)
```

Описание:

SPI_cursor_open_with_paramlist открывает курсор (внутри называемый порталом) для выполнения оператора, подготовленного функцией SPI_prepare. Эта функция равнозначна SPI_cursor_open, не считая того, что информация о значениях параметров, передаваемых запросу, представляется по-другому. Представление ParamListInfo может быть удобным для передачи значений, уже имеющих нужный формат. Эта функция также поддерживает динамические наборы параметров через функции-обработчики, устанавливаемые в ParamListInfo.

Переданные значения параметров копируются в портал курсора, так что их можно освободить и во время существования курсора.

Аргументы:

– const char * *name* – имя портала, либо NULL, чтобы имя выбрала система.

– SPIPlanPtr *plan* – подготовленный оператор (возвращаемый функцией SPI_prepare).

– ParamListInfo *params* – структура данных, содержащая типы и значения параметров; NULL, если их нет.

– bool *read_only* – true для режима выполнения «только чтение».

Возвращаемое значение:

Указатель на портал, содержащий курсор. Соглашение о возврате ошибок отсутствует; все ошибки выдаются через elog.

– SPI_cursor_find — найти существующий курсор по имени.

Синтаксис:

```
Portal SPI_cursor_find(const char * name)
```

Описание:

SPI_cursor_find находит существующий портал по имени. В основном это используется для разрешения имени курсора, возвращённого в текстовом виде какой-то другой функцией.

Аргументы:

– const char * *name* – имя портала.

Возвращаемое значение:

указатель на портал с заданным именем или NULL, если такой портал не найден.

– SPI_cursor_fetch — выбрать строки через курсор.

Синтаксис:

```
void SPI_cursor_fetch(Portal portal, bool forward, long count)
```

Описание:

SPI_cursor_fetch выбирает некоторое количество строк через курсор. Эта функция реализует подмножество возможностей SQL-команды FETCH (расширенную функциональность предоставляет SPI_scroll_cursor_fetch).

Аргументы:

– Portal *portal* – портал, содержащий курсор.

– bool *forward* – true для выборки с перемещением вперёд, false — назад.

– long *count* – максимальное число строк, которое нужно выбрать.

Возвращаемое значение:

Переменные SPI_processed и SPI_tuptable устанавливаются как в SPI_execute, если вызов был успешным.

Замечания:

Выборка назад может не поддерживаться, если план курсора был создан без параметра CURSOR_OPT_SCROLL.

– SPI_cursor_move — переместить курсор.

Синтаксис:

```
void SPI_cursor_move(Portal portal, bool forward, long count)
```

Описание:

SPI_cursor_move перемещает курсор на несколько строк. Эта функция реализует подмножество возможностей SQL-команды MOVE (расширенную функциональность предоставляет SPI_scroll_cursor_move).

Аргументы:

– Portal *portal* – портал, содержащий курсор.

– bool *forward* – true для перемещения вперёд, false — назад.

– long *count* – максимальное число строк, на какое возможно перемещение.

Замечания:

Перемещение назад может не поддерживаться, если план курсора был создан без параметра CURSOR_OPT_SCROLL.

– SPI_scroll_cursor_fetch — выбрать строки через курсор.

Синтаксис:

```
void SPI_scroll_cursor_fetch(Portal portal, FetchDirection direction, long count)
```

Описание:

SPI_scroll_cursor_fetch выбирает некоторое количество строк через курсор. Её функциональность равнозначна FETCH в SQL.

Аргументы:

– Portal *portal* – портал, содержащий курсор.

– FetchDirection *direction* – один из вариантов: FETCH_FORWARD, FETCH_BACKWARD, FETCH_ABSOLUTE или FETCH_RELATIVE.

– long *count* – число строк, выбираемых с направлением FETCH_FORWARD или FETCH_BACKWARD; абсолютный номер выбираемой строки с вариантом FETCH_ABSOLUTE; либо относительный номер выбираемой строки с вариантом FETCH_RELATIVE

Возвращаемое значение:

Переменные SPI_processed и SPI_tuptable устанавливаются как в SPI_execute, если вызов был успешным.

Замечания:

Подробнее о параметрах *direction* и *count* рассказывается в описании SQL-команды FETCH.

Варианты направления, отличные от FETCH_FORWARD, могут не поддерживаться, если план курсора был создан без параметра CURSOR_OPT_SCROLL.

– SPI_scroll_cursor_move — переместить курсор.

Синтаксис:

```
void SPI_scroll_cursor_move(Portal portal, FetchDirection direction, long count)
```

Описание:

SPI_scroll_cursor_move перемещает курсор на несколько строк. Её функциональность равнозначна MOVE в SQL.

Аргументы:

– Portal *portal* – портал, содержащий курсор.

– FetchDirection *direction* – один из вариантов: FETCH_FORWARD, FETCH_BACKWARD, FETCH_ABSOLUTE или FETCH_RELATIVE.

– long *count* – число строк, на которое сдвигается курсор, с направлением FETCH_FORWARD или FETCH_BACKWARD; абсолютный номер строки, к которой переходит курсор, с направлением FETCH_ABSOLUTE; либо относительный номер строки, к которой переходит курсор, с направлением FETCH_RELATIVE

Возвращаемое значение:

В случае успеха переменная SPI_processed устанавливается как в SPI_execute. В SPI_tuptable оказывается NULL, так как эта функция не возвращает никакие строки.

Замечания:

Подробнее о параметрах *direction* и *count* рассказывается в описании SQL-команды FETCH.

Варианты направления, отличные от FETCH_FORWARD, могут не поддерживаться, если план курсора был создан без параметра CURSOR_OPT_SCROLL.

– SPI_cursor_close — закрыть курсор.

Синтаксис:

```
void SPI_cursor_close(Portal portal)
```

Описание:

SPI_cursor_close закрывает ранее созданный курсор и освобождает память, занятую его порталом.

Все открытые курсоры закрываются автоматически в конце транзакции. Вызывать SPI_cursor_close может потребоваться, только если возникает желание освободить ресурсы скорее.

Аргументы:

– Portal *portal* – портал, содержащий курсор.

– SPI_keepplan — сохранить подготовленный оператор.

Синтаксис:

```
int SPI_keepplan(SPIPlanPtr plan)
```

Описание:

SPI_keepplan закрепляет переданный оператор (подготовленный функцией SPI_prepare), чтобы он не был ликвидирован функцией SPI_finish или диспетчером транзакций. Это даёт возможность повторно использовать подготовленные операторы при последующих вызовах функции на C в текущем сеансе.

Аргументы:

– SPIPlanPtr *plan* – подготовленный оператор, который нужно сохранить.

Возвращаемое значение:

0 в случае успеха; SPI_ERROR_ARGUMENT, если *plan* неверный или NULL

Замечания

Переданный оператор перемещается в постоянное хранилище путём смены указателя (копировать данные не требуется). Если нужно удалить его, необходимо выполнить для него функцию SPI_freeplan.

– SPI_saveplan — сохранить подготовленный оператор.

Синтаксис:

```
SPIPlanPtr SPI_saveplan(SPIPlanPtr plan)
```

Описание:

SPI_saveplan копирует переданный оператор (подготовленный функцией SPI_prepare) в память, чтобы он не был ликвидирован функцией SPI_finish или менеджером транзакций, и возвращает указатель на скопированный оператор. Это даёт возможность повторно использовать подготовленные операторы при последующих вызовах функции на C в текущем сеансе.

Аргументы:

– SPIPlanPtr *plan* – подготовленный оператор, который нужно сохранить.

Возвращаемое значение:

Указатель на скопированный оператор, либо NULL в случае ошибки. При ошибке SPI_result принимает одно из этих значений:

SPI_ERROR_ARGUMENT – если *plan* неверный или

NULL SPI_ERROR_UNCONNECTED – если вызывается из неподключённой функции на C

Замечания:

Изначально переданный оператор не освобождается, поэтому необходимо выполнить SPI_freeplan для него, чтобы высвободить память до SPI_finish.

В большинстве случаев SPI_keepplan предпочтительнее данной функции, так как она даёт примерно тот же результат, но обходится без физического копирования структур данных подготовленного оператора.

– SPI_register_relation — сделать эфемерное именованное отношение доступным по имени в запросах SPI.

Синтаксис:

```
int SPI_register_relation(EphemeralNamedRelation enr)
```

Описание:

SPI_register_relation делает эфемерное именованное отношение (со связанной информацией) доступным в запросах, планируемых и выполняемых через текущее подключение SPI.

Аргументы:

– EphemeralNamedRelation *enr* – запись эфемерного именованного отношения в реестре.

Возвращаемое значение:

Если команда была выполнена успешно, возвращается следующее (неотрицательное) значение:

SPI_OK_REL_REGISTER – если отношение было успешно зарегистрировано по имени.

В случае ошибки возвращается одно из следующих отрицательных значений:

SPI_ERROR_ARGUMENT – если NULL передан в *enr* или в поле name/

SPI_ERROR_UNCONNECTED – если вызывается из неподключённой функции на C.

SPI_ERROR_REL_DUPLICATE – если имя, заданное в поле name структуры *enr*, уже зарегистрировано для этого отношения.

– SPI_unregister_relation — удалить эфемерное именованное отношение из реестра.

Синтаксис:

```
int SPI_unregister_relation(const char * name)
```

Описание:

SPI_unregister_relation удаляет эфемерное именованное отношение из реестра для текущего подключения.

Аргументы:

– const char * *name* – имя записи отношения в реестре.

Возвращаемое значение:

Если команда была выполнена успешно, возвращается следующее (неотрицательное) значение:

SPI_OK_REL_UNREGISTER – если совокупность кортежей была успешно удалена из реестра.

В случае ошибки возвращается одно из следующих отрицательных значений:

SPI_ERROR_ARGUMENT – если в *name* передан NULL.

SPI_ERROR_UNCONNECTED – если вызывается из неподключённой функции на C.

SPI_ERROR_REL_NOT_FOUND – если *name* не находится в реестре для текущего подключения.

– SPI_register_trigger_data — сделать эфемерные данные триггера доступными в запросах SPI.

Синтаксис:

```
int SPI_register_trigger_data(TriggerData *tdata)
```

Описание:

SPI_register_trigger_data делает эфемерные отношения, которые перехватывает триггер, доступными для запросов, планируемых и выполняемых через текущее подключение SPI. В настоящее время это переходные таблицы, перехватываемые триггером AFTER, определённым с предложением REFERENCING OLD/NEW TABLE AS. Эта функция должна вызываться функцией, реализующей триггер на языке программирования, после подключения.

Аргументы:

– TriggerData **tdata* – объект TriggerData, передаваемый функцией, реализующей триггер, через fcinfo->context.

Возвращаемое значение:

Если команда была выполнена успешно, возвращается следующее (неотрицательное) значение:

SPI_OK_TD_REGISTER – если перехваченные данные триггера (при наличии) были успешно зарегистрированы.

В случае ошибки возвращается одно из следующих отрицательных значений:

SPI_ERROR_ARGUMENT – если в *tdata* передан NULL.

SPI_ERROR_UNCONNECTED – если вызывается из неподключённой функции на C.

SPI_ERROR_REL_DUPLICATE – если имя в любом из переходных отношений в данных триггера уже зарегистрировано для этого подключения.

Вспомогательные интерфейсные функции, предоставляющие возможности для извлечения информации из наборов результатов, возвращаемых SPI_execute и другими функциями SPI:

– SPI_fname — определить имя столбца с заданным номером.

Синтаксис:

```
char * SPI_fname(TupleDesc rowdesc, int colnumber)
```

Описание:

SPI_fname возвращает копию имени столбца с заданным номером. (Когда эта копия имени будет не нужна, её можно освободить с помощью pfree.).

Аргументы:

– TupleDesc *rowdesc* – описание строк.

– int *colnumber* – номер столбца (начиная с 1).

Возвращаемое значение:

Имя столбца; NULL, если *colnumber* вне допустимого диапазона.

В случае ошибки в SPI_result устанавливается SPI_ERROR_NOATTRIBUTE.

– SPI_fnumber — определить номер столбца с заданным именем.

Синтаксис:

```
int SPI_fnumber(TupleDesc rowdesc, const char * colname)
```

Описание:

SPI_fnumber возвращает номер столбца, имеющего заданное имя.

Если *colname* ссылается на системный столбец (например, ctid), возвращается соответствующий отрицательный номер столбца. Вызывающий должен проверять, не была ли

возвращена ошибка, сравнивая значение результата именно с SPI_ERROR_NOATTRIBUTE; проверка результата по условию меньше или равно нулю не будет корректной, если только системные столбцы не должны исключаться.

Аргументы:

– TupleDesc *rowdesc* – описание строк.

– const char * *colname* – имя столбца.

Возвращаемое значение:

Номер столбца (начиная с 1 для столбцов, создаваемых пользователем), либо SPI_ERROR_NOATTRIBUTE, если столбец с заданным именем не найден.

– SPI_getvalue — получить строковое значение указанного столбца.

Синтаксис:

```
char * SPI_getvalue(HeapTuple row, TupleDesc rowdesc, int colnumber)
```

Описание:

SPI_getvalue возвращает строковое представление значения указанного столбца.

Результат возвращается в памяти, размещённой функцией palloc. (Когда он будет не нужен, эту память можно освободить с помощью pfree.)

Аргументы:

– HeapTuple *row* – строка с нужными данными.

– TupleDesc *rowdesc* – описание строк.

– int *colnumber* – номер столбца (начиная с 1).

Возвращаемое значение:

Значение столбца, либо NULL, если столбец содержит NULL, *colnumber* вне допустимого диапазона (в SPI_result при этом устанавливается SPI_ERROR_NOATTRIBUTE) или если отсутствует функция вывода (в SPI_result устанавливается SPI_ERROR_NOOUTFUNC).

– SPI_getbinval — получить двоичное значение указанного столбца.

Синтаксис:

```
Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc, int colnumber, bool * isnull)
```

Описание:

SPI_getbinval возвращает значение указанного столбца во внутренней форме (в структуре Datum).

Эта функция не выделяет новый блок памяти для данных. В случае с типом, передаваемым по ссылке, возвращаемым значением будет указатель на переданную строку данных.

Аргументы:

– HeapTuple *row* – строка с нужными данными.

– TupleDesc *rowdesc* – описание строк.

– int *colnumber* – номер столбца (начиная с 1).

– bool * *isnull* – признак того, что столбец содержит NULL.

Возвращаемое значение:

Возвращается двоичное значение столбца. Если этот столбец содержит NULL, переменной, на которую указывает *isnull*, присваивается true; в противном случае — false.

При ошибке в SPI_result устанавливается SPI_ERROR_NOATTRIBUTE.

– SPI_gettype — получить имя типа данных указанного столбца

Синтаксис:

```
char * SPI_gettype(TupleDesc rowdesc, int colnumber)
```

Описание:

SPI_gettype возвращает копию имени типа данных указанного столбца. (Когда эта копия имени будет не нужна, её можно освободить с помощью pfree.)

Аргументы:

– TupleDesc *rowdesc* – описание строк.

– int *colnumber* – номер столбца (начиная с 1).

Возвращаемое значение:

Имя типа данных указанного столбца, либо NULL в случае ошибки.

При ошибке в SPI_result устанавливается SPI_ERROR_NOATTRIBUTE.

– SPI_gettypeid — получить OID типа данных указанного столбца.

Синтаксис:

```
Oid SPI_gettypeid(TupleDesc rowdesc, int colnumber)
```

Описание:

SPI_gettypeid возвращает OID типа данных указанного столбца.

Аргументы:

– TupleDesc *rowdesc* – описание строк.

– int *colnumber* – номер столбца (начиная с 1).

Возвращаемое значение:

OID типа данных указанного столбца, либо InvalidOid в случае ошибки. При ошибке в SPI_result устанавливается SPI_ERROR_NOATTRIBUTE.

– SPI_getrelname — возвращает имя указанного отношения.

Синтаксис:

```
char * SPI_getrelname(Relation rel)
```

Описание:

SPI_getrelname возвращает копию имени указанного отношения. (Когда эта копия имени будет не нужна, её можно освободить с помощью pfree.)

Аргументы:

– Relation *rel* – целевое отношение.

Возвращаемое значение:

Имя указанного отношения.

– SPI_getnspname — возвращает пространство имён указанного отношения.

Синтаксис:

```
char * SPI_getnspname(Relation rel)
```

Описание:

SPI_getnspname возвращает копию имени пространства имён, к которому принадлежит указанное отношение (Relation). Пространство имён по-другому называется схемой отношения.

Когда значение, возвращённое этой функцией, будет не нужно, необходимо освободить его с помощью pfree.

Аргументы:

– Relation *rel* – целевое отношение.

Возвращаемое значение:

Имя пространства имён указанного отношения.

– SPI_result_code_string — возвращает код ошибки в виде строки.

Синтаксис:

```
const char * SPI_result_code_string(int code);
```

Описание:

SPI_result_code_string выдаёт строковое представление для кода результата, который возвращается различными функциями SPI или находится в SPI_result.

Аргументы:

– int *code* – код результата.

Возвращаемое значение

Строковое представление кода результата.

28.10.3. Управление памятью

PG360 выделяет память в *контекстах памяти* и тем самым реализует удобный способ управления выделением памяти в различных местах, с разными сроками жизни выделенной памяти. При уничтожении контекста освобождается вся выделенная в нём память. Таким образом, нет необходимости контролировать каждый отдельный объект во избежание утечек памяти; вместо этого достаточно управлять только небольшим числом контекстов. Функция palloc и родственные ей освобождают память из «текущего» контекста.

SPI_connect создаёт новый контекст памяти и делает его текущим. SPI_finish восстанавливает контекст, который был текущим до этого, и уничтожает контекст, созданный функцией SPI_connect. Эти действия обеспечивают при выходе из функции на C освобождение временной памяти, выделенной внутри этой функции, во избежание утечки памяти.

Когда вызывается SPI_connect, текущим контекстом становится частный контекст функции на C, создаваемый в SPI_connect. Все операции выделения памяти, выполняемые функциями palloc, repalloc или служебными функциями SPI (кроме описанных в этом разделе исключений), производятся в этом контексте. Когда функция на C отключается от менеджера SPI (выполняя SPI_finish), текущим контекстом снова становится верхний контекст исполнителя, а вся память, выделенная в контексте этой функции, освобождается, так что использовать её дальше нельзя.

– SPI_palloc — выделить память в верхнем контексте исполнителя.

Синтаксис:

```
void * SPI_palloc(Size size)
```

Описание:

SPI_palloc выделяет память в верхнем контексте исполнителя.

Эту функцию можно использовать только когда установлено подключение к SPI. В противном случае она выдаёт ошибку.

Аргументы:

– Size *size* – размер выделяемой памяти, в байтах.

Возвращаемое значение:

указатель на выделенный блок памяти заданного размера.

– SPI_repalloc — поменять блок памяти в верхнем контексте исполнителя.

Синтаксис:

```
void * SPI_repalloc(void * pointer, Size size)
```

Описание:

SPI_repalloc изменяет размер блока памяти, ранее выделенного функцией SPI_palloc.

Эта функция теперь не отличается от простой repalloc. Она сохранена только для обратной совместимости с существующим кодом.

Аргументы:

– void * *pointer* – указатель на существующий блок памяти, подлежащий изменению.

– Size *size* – размер выделяемой памяти, в байтах.

Возвращаемое значение:

указатель на новый блок памяти указанного размера, в который скопировано содержимое прежнего блока

– SPI_pfree — освободить память в верхнем контексте исполнителя.

Синтаксис:

```
void SPI_pfree(void * pointer)
```

Описание:

SPI_pfree освобождает память, ранее выделенную функцией SPI_palloc или SPI_repalloc.

Эта функция теперь не отличается от простой pfree. Она сохранена только для обратной совместимости с существующим кодом.

Аргументы:

– void * *pointer* – указатель на существующий блок памяти, подлежащий освобождению.

– SPI_copytuple — скопировать строку в верхнем контексте исполнителя.

Синтаксис:

```
HeapTuple SPI_copytuple(HeapTuple row)
```

Описание:

SPI_copytuple делает копию строки в верхнем контексте исполнителя. Обычно это применяется, когда нужно вернуть изменённую строку из триггера. В функции, которая должна возвращать составной тип, нужно использовать SPI_returntuple.

Эту функцию можно использовать только когда установлено подключение к SPI. В противном случае она возвращает NULL и устанавливает в SPI_result значение SPI_ERROR_UNCONNECTED.

Аргументы:

– HeapTuple *row* – строка, подлежащая копированию.

Возвращаемое значение:

скопированная строка либо NULL в случае ошибки (SPI_result содержит код ошибки).

– SPI_returntuple — подготовить строку для возврата в виде Datum.

Синтаксис:

```
HeapTupleHeader SPI_returntuple(HeapTuple row, TupleDesc rowdesc)
```

Описание:

SPI_returntuple делает копию строки в верхнем контексте исполнителя и возвращает её в форме типа Datum. Чтобы выдать результат, полученный указатель остаётся только преобразовать в Datum функцией PointerGetDatum.

Эту функцию можно использовать только когда установлено подключение к SPI. В противном случае она возвращает NULL и устанавливает в SPI_result значение SPI_ERROR_UNCONNECTED.

Эту операцию следует применять в функциях, объявленных как возвращающие составные типы. В триггерах она не применяется; чтобы вернуть изменённую строку из триггера, используется SPI_copytuple.

Аргументы:

– HeapTuple *row* – строка, подлежащая копированию.

– TupleDesc *rowdesc* – дескриптор строки (передается каждый раз один дескриптор для более эффективного кеширования).

Возвращаемое значение:

HeapTupleHeader, указывающий на скопированную строку, или NULL в случае ошибки (SPI_result содержит код ошибки).

– SPI_modifytuple — создать строку, заменяя отдельные поля в данной.

Синтаксис:

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple row, int ncols,
int * colnum, Datum * values, const char * nulls)
```

Описание:

SPI_modifytuple создаёт новую строку, подставляя новые значения для указанных столбцов и копируя исходное содержимое остальных столбцов. Исходная строка не изменяется. Новая строка возвращается в верхнем контексте исполнителя.

Эту функцию можно использовать только когда установлено подключение к SPI. В противном случае она возвращает NULL и устанавливает в SPI_result значение SPI_ERROR_UNCONNECTED.

Аргументы:

– Relation *rel* – Используется только в качестве дескриптора строки. (Передача отношения вместо собственно дескриптора строки — нехорошая особенность.)

– HeapTuple *row* – строка, подлежащая изменению.

– int *ncols* – число изменяемых столбцов.

– int * *colnum* – массив длины *ncols*, содержащий номера изменяемых столбцов (начиная с 1).

– Datum * *values* – массив длины *ncols*, содержащий новые значения указанных столбцов.

– const char * *nulls* – массив длины *ncols*, описывающий, в каких столбцах передаётся NULL.

Если в *nulls* передаётся NULL, SPI_modifytuple считает, что ни один из параметров не равен NULL. В противном случае элемент массива *nulls* должен содержать ' ', если значение соответствующего параметра не NULL, либо 'n', если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *values*, не учитывается.) *nulls* — это не текстовая строка, а просто массив: ноль '\0' в конце не нужен.

Возвращаемое значение:

новая строка с изменениями, размещённая в верхнем контексте исполнителя, или NULL при ошибке (SPI_result содержит код ошибки).

В случае ошибки в SPI_result устанавливается:

SPI_ERROR_ARGUMENT – если *rel* — NULL, либо *row* — NULL, либо *ncols* меньше или равно 0, либо *colnum* — NULL, либо *values* — NULL.

SPI_ERROR_NOATTRIBUTE – если *colnum* содержит недопустимый номер столбца (меньше или равен 0, либо больше числа столбцов в строке *row*).

SPI_ERROR_UNCONNECTED – если SPI неактивен.

– SPI_freetuple — освободить строку, размещённую в верхнем контексте исполнителя.

Синтаксис:

```
void SPI_freetuple(HeapTuple row)
```

Описание:

SPI_freetuple освобождает строку, ранее размещённую в верхнем контексте исполнителя.

Эта функция теперь не отличается от простой heap_freetuple. Она сохранена только для обратной совместимости с существующим кодом.

Аргументы:

– HeapTuple *row* – строка, подлежащая освобождению.

– SPI_freetuptable — освободить набор строк, созданный SPI_execute или подобной функцией.

Синтаксис:

```
void SPI_freetuptable(SPITupleTable * tuptable)
```

Описание:

SPI_freetuptable освобождает набор строк, созданных предыдущей функцией SPI выполнения команд, например SPI_execute. Таким образом, при вызове этой функции в качестве аргумента часто передаётся глобальная переменная SPI_tuptable.

Эта функция используется, когда функция на C, использующая SPI, должна выполнить несколько команд, но не хочет сохранять результаты предыдущих команд до завершения. Любые не освобождённые таким образом наборы строк будут всё равно освобождены при выполнении SPI_finish. Кроме того, если была запущена подтранзакция, а затем она прервалась в ходе выполнения использующей SPI функции, все наборы строк, созданные в рамках подтранзакции, будут автоматически освобождены.

Аргументы:

– SPITupleTable * *tuptable* – указатель на набор строк, который нужно освободить (если NULL, ничего не происходит).

– SPI_freeplan — освободить ранее сохранённый подготовленный оператор.

Синтаксис:

```
int SPI_freeplan(SPIPlanPtr plan)
```

Описание:

SPI_freeplan освобождает подготовленный оператор, до этого выданный функцией SPI_prepare или сохранённый функциями SPI_keepplan и SPI_saveplan.

Аргументы:

– SPIPlanPtr *plan* – указатель на оператор, подлежащий освобождению.

Возвращаемое значение

0 в случае успеха; SPI_ERROR_ARGUMENT, если *plan* неверный или NULL

28.10.4. Управление транзакциями

Выполнять команды управления транзакциями (в частности, COMMIT и ROLLBACK) через функции SPI, такие как SPI_execute, нельзя. Однако имеются отдельные интерфейсные функции, которые предназначены для управления транзакциями через SPI.

Не всегда безопасно и разумно начинать и заканчивать транзакции в произвольных определяемых пользователями функциях, вызываемых из SQL, не принимая во внимание контекст их вызова. Например, завершение транзакции в середине функции, вызванной в сложном SQL–выражении внутри некоторой SQL-команды, скорее всего приведёт к странным внутренним ошибкам или сбоям. Представленные здесь интерфейсные функции прежде всего предназначены для использования реализациями процедурных языков с целью управления транзакциями в процедурах уровня SQL, вызываемых командой CALL (при этом учитывается её контекст):

– SPI_commit, SPI_commit_and_chain — зафиксировать текущую транзакцию.

Синтаксис:

```
void SPI_commit(void)
void SPI_commit_and_chain(void)
```

Описание:

SPI_commit фиксирует текущую транзакцию. Это примерно равносильно выполнению SQL–команды COMMIT. После того как транзакция зафиксирована, автоматически начинается новая транзакция с характеристиками по умолчанию, что позволяет сразу продолжить использование функций SPI. Если во время фиксации происходит ошибка, текущая транзакция откатывается, начинается новая, а затем уже выдаётся ошибка.

SPI_commit_and_chain делает то же самое, но новая транзакция получает те же характеристики, что завершённая. Эта функция подобна SQL-команде COMMIT AND CHAIN.

Эти функции можно выполнить, только если SPI-подключение переведено в неатомарный режим в результате вызова SPI_connect_ext.

– SPI_rollback, SPI_rollback_and_chain — прервать текущую транзакцию.

Синтаксис:

```
void SPI_rollback(void)
void SPI_rollback_and_chain(void)
```

Описание:

SPI_rollback откатывает текущую транзакцию. Это примерно равносильно выполнению SQL– команды ROLLBACK. После того как транзакция отменена, автоматически начинается новая транзакция с характеристиками по умолчанию, что позволяет сразу продолжить использование функций SPI.

SPI_rollback_and_chain делает то же самое, но новая транзакция получает те же характеристики, что завершённая. Эта функция подобна SQL-команде ROLLBACK AND CHAIN.

Эти функции можно выполнить, только если SPI-подключение переведено в неатомарный режим в результате вызова SPI_connect_ext.

– SPI_start_transaction — устаревшая функция.

Синтаксис:

```
void SPI_start_transaction(void)
```

Описание:

Функция SPI_start_transaction не делает ничего, она сохранена только для совместимости с более ранними выпусками PG360. Ранее её надо было вызывать после SPI_commit или SPI_rollback, но сейчас эти функции начинают новую транзакцию автоматически.

Видимость изменений в данных. Видимость изменений в данных, которые производятся функциями, использующими SPI, (или любыми другими функциями на C), описывается следующими правилами:

– В процессе выполнения SQL-команды любые произведённые ей изменения не видны для неё самой. Например, в команде:

INSERT INTO a SELECT * FROM a;

вставляемые строки не видны в части SELECT.

– Изменения, произведённые командой К, видны во всех командах, запущенных после К, независимо от того, были ли эти команды запущены из К (во время выполнения К) или после завершения К.

– Команды, выполняемые через SPI внутри функции, вызванной SQL-командой (будь то обычная функция или триггер), следуют одному или другому из вышеприведённых правил в зависимости флага чтения/записи, переданного SPI. Команды, выполняемые в режиме «только чтение», следует первому правилу: они не видят изменений, произведённых вызывающей командой. Команды, выполняемые в режиме «чтение-запись», следуют второму правилу: они могут видеть все произведённые к этому времени изменения.

– Все стандартные процедурные языки устанавливают режим чтения-записи в SPI в зависимости от атрибута изменчивости функции. Команды функций STABLE и IMMUTABLE выполняются в режиме «только чтение», тогда как команды функций VOLATILE — в режиме «чтение-запись». Хотя авторы функций на C могут нарушить это соглашение, вряд ли это будет хорошей идеей.

2.9. Порядок действий в случае сбоев, отказа компьютера, при уничтожении или модификации программ и служебных данных СУБД PG360

2.9.1. При аппаратном отказе сервера необходимо выполнить следующие действия:

– заменить сервер;

– установить необходимое общесистемное программное обеспечение;

– установить СУБД PG360, руководствуясь п. 2.5.;

– восстановить данные из резервной копии.

2.9.2. При уничтожении или модификации программ и служебных данных СУБД PG360 оператор - администратор СУБД PG360 должен выполнить следующие действия:

– установить СУБД PG360, руководствуясь п. 2.5.;

– восстановить данные из резервной копии.

3. ОБРАЩЕНИЕ К ПРОГРАММЕ

3.1. Вызов СУБД PG360 осуществляется следующим способом:

– запуском программы pg_ctl для инициализации, запуска, остановки или управления сервером PG360 в ОС Linux из папки /usr/local/pgsql/bin/:

```
pg_ctl init[db] [-D каталог_данных] [-s] [-o параметры-initdb]
pg_ctl start [-D каталог_данных] [-l имя_файла] [-W] [-t секунды] [-s] [-o параметры] [-p путь] [-c]
pg_ctl stop [-D каталог_данных] [-m s[mart] | f[ast] | i[mmediate] ] [-W] [-t секунды] [-s]
pg_ctl restart [-D каталог_данных] [-m s[mart] | f[ast] | i[mmediate] ] [-W] [-t секунды] [-s] [- o параметры] [-c]
pg_ctl reload [-D каталог_данных] [-s]
pg_ctl status [-D каталог_данных]
pg_ctl promote [-D каталог_данных] [-W] [-t секунды] [-s]
pg_ctl logrotate [-D каталог_данных] [-s]
pg_ctl kill имя_сигнала ид_процесса
```

3.2. Вызов серверных приложений осуществляется следующим способом:

– запуском программы initdb для создания кластера баз данных PG360:

```
initdb [параметр...] [ --pgdata | -D ]каталог
```

– запуском программы pg_archivecleanup для вычищения файлов архивов WAL PG360:

```
pg_archivecleanup [параметр...] расположение_архива старейший_сохраняемый_файл
```

– запуском программы pg_checksums для включения, отключения или проверки контрольных сумм данных в кластере PG360:

```
pg_checksums [параметр...] [[ -D | --pgdata ]каталог_данных]
```

– запуском программы pg_controldata для вывода управляющей информации кластера баз данных PG360:

```
pg_controldata [параметр] [[ -D | --pgdata ]datadir]
```

– запуском программы pg_resetwal для очистки журнала упреждающей записи и другой управляющей информации кластера PG360:

```
pg_resetwal [ -f | --force ] [ -n | --dry-run ] [параметр...] [ -D | --pgdata ]каталог_данных
```

– запуском программы pg_rewind для синхронизации каталога данных PG360 с другим каталогом, ответвлённым от него:

```
pg_rewind [параметр...] { -D | --target-pgdata }каталог { --source-pgdata=каталог | --source- server=строка_подключения }
```

– запуском программы pg_test_fsync для подбора наилучшего варианта wal_sync_method для PG360:

```
pg_test_fsync [параметр...]
```

– запуском программы pg_test_timing для определения издержки замера времени:

```
pg_test_timing [параметр...]
```

– запуском программы pg_waldump для вывода журнала упреждающей записи кластера БД PG360:

```
pg_waldump [параметр...] [начальный_сегмент [конечный_сегмент]]
```

– запуском программы postgres для управления сервером баз данных PG360:

```
postgres [параметр...]
```

3.2. Вызов клиентских приложений осуществляется следующим способом:

– запуском программы clusterdb для повторной кластеризации таблиц базы данных PG360:

```
clusterdb [параметр-подключения...] [ --verbose | -v ] [ --table | -t таблица ] ... [имя_бд]
clusterdb [параметр-подключения...] [ --verbose | -v ] --all | -a
```

– запуском программы createdb для создания базы данных PG360:

```
createdb [параметр-подключения...] [параметр...] [имя_бд [описание]]
```

– запуском программы createuser для создания новой учётной записи PG360:

```
createuser [параметр-подключения...] [параметр...] [имя_пользователя]
```

– запуском программы dropdb для удаления базы данных PG360:

```
dropdb [параметр-подключения...] [параметр...] имя_бд
```

– запуском программы dropuser для удаления учётной записи пользователя PG360:

```
dropuser [параметр-подключения...] [параметр...] [имя_пользователя]
```

– запуском программы ecpg для запуска встроенного C-препроцессора SQL:

```
ecpg [параметр...] файл...
```

– запуском программы pg_basebackup для создания резервной копии кластера PG360:

```
pg_basebackup [параметр...]
```

– запуском программы pgbench для запуска теста производительности PG360:

```
pgbench -i [параметр...] [имя_бд]
pgbench [параметр...] [имя_бд]
```

– запуском программы pg_config для вывода информации об установленной версии PG360:

```
pg_config [параметр...]
```

– запуском программы pg_dump для выгрузки базы данных PG360 в виде скрипта или в архивном формате:

```
pg_dump [параметр-подключения...] [параметр...] [имя_бд]
```

– запуском программы pg_dumpall для выгрузки кластера баз данных PG360 в формате скрипта:

```
pg_dumpall [параметр-подключения...] [параметр...]
```

– запуском программы pg_isready для проверки соединения с сервером PG360:

```
pg_isready [параметр-подключения...] [параметр...]
```

– запуском программы pg_receivewal для приема журнала упреждающей записи с сервера PG360:

```
pg_receivewal [параметр...]
```

– запуском программы pg_ recvlogical для управления потоками логического декодирования PG360:

```
pg_recvlogical [параметр...]
```

– запуском программы pg_restore для восстановления базы данных PG360 из файла архива, созданного командой pg_dump:

```
pg_restore [параметр-подключения...] [параметр...] [имя_файла]
```

– запуском программы pg_verifybackup для проверки целостности базовой копии кластера PG360:

```
pg_verifybackup [параметр...]
```

– запуском программы psql – интерактивного терминала PG360:

```
psql [параметр...] [имя_бд [имя_пользователя]]
```

– запуском программы reindexdb для перестроения индексов в базе данных PG360:

```
reindexdb [параметр-подключения...] [параметр...] [ -S | --schema схема ] ... [ -t | --table таблица] ... [ -i | --index индекс ] ... [имя_бд]
reindexdb [параметр-подключения...] [параметр...] -a | --all
reindexdb [параметр-подключения...] [параметр...] -s | --system [имя_бд]
```

– запуском программы vacuumdb для выполнения очистки и анализа базы данных PG360:

```
vacuumdb [параметр-подключения...] [параметр...] [ -t | --table таблица [( столбец [,...] )] ] ... [имя_бд]
vacuumdb [параметр-подключения...] [параметр...] -a | --all
```

4. ВХОДНЫЕ И ВЫХОДНЫЕ ДАННЫЕ

4.1. Входные и выходные данные программы pg_ctl для инициализации, запуска, остановки или управления сервером PG360:

Команда start запускает сервер. Процесс запускается в фоне, а стандартный ввод связывается с /dev/null. По умолчанию в Unix-подобных системах вывод и ошибки сервера пишутся в устройство стандартного вывода (не ошибок) pg_ctl. Вывод pg_ctl следует перенаправить в файл или процесс, например, приложение ротации журналов rotatelogs; иначе postgres будет писать вывод в управляющий терминал (в фоновом режиме) и останется в группе процессов оболочки.Это поведение по умолчанию можно изменить и направить вывод сервера в файл, добавив ключ -l. Предпочтительными вариантами является использование -l или перенаправление вывода.

Команда stop останавливает сервер, работающий с указанным каталогом данных. Параметр -m позволяет выбрать один из трёх режимов остановки. Режим «Smart» запрещает новые подключения, а затем ожидает отключения всех существующих клиентов и завершения всех текущих процессов резервного копирования. Если сервер работает в режиме горячего резерва, восстановление и потоковая репликация будут прерваны, как только отключатся все клиенты. Режим «Fast» (выбираемый по умолчанию) не ожидает отключения клиентов и завершает все текущие процессы резервного копирования. Все активные транзакции откатываются, а клиенты принудительно отключаются, после чего сервер останавливается. Режим «Immediate» незамедлительно прерывает все серверные процессы, не выполняя процедуру штатной остановки. Этот вариант влечёт необходимость выполнить восстановление после сбоя при следующем запуске сервера.

Команда restart производит остановку и последующий запуск сервера. Это позволяет изменить параметры командной строки postgres либо применить изменения в файле конфигурации, не вступающие в силу без перезапуска сервера. Если в командной строке при запуске сервера указывались относительные пути, команда restart может не выполниться, если вызвать pg_ctl не в том каталоге, где производился предыдущий запуск.

Команда reload посылает процессу сервера postgres сигнал SIGHUP, получив который он перечитывает свои файлы конфигурации (postgresql.conf, pg_hba.conf и т. д.). Это позволяет применить изменения параметров в файле конфигурации, не требующие полного перезапуска сервера.

Команда status проверяет, работает ли сервер в указанном каталоге данных. Если да, она выдаёт PID сервера и параметры командной строки, с которыми он был запущен. Если сервер не работает, pg_ctl возвращает код завершения 3. Если в параметрах не указан доступный каталог данных, pg_ctl возвращает код завершения 4.

Команда promote указывает серверу, работающему в режиме резерва с указанным каталогом данных, выйти из этого режима и начать операции чтения/записи.

Команда logrotate прокручивает файл журнала сервера.

Команда kill передаёт сигнал заданному процессу. Для получения списка имён поддерживаемых сигналов используется параметр --help.

Параметры:

-c, --core-files – Способствует сбросу дампа памяти процесса при крахе сервера на платформах, где это возможно, поднимая мягкие ограничения, задаваемые для файлов дампа. Это используется при отладке и диагностике проблем, так как позволяет получить трассировку стека отказавшего процесса сервера.

-D *каталог_данных,*

--pgdata=*каталог_данных* – Указывает размещение конфигурационных файлов кластера. Если этот ключ опущен, используется значение переменной окружения PGDATA.

-l *имя_файла,*

--log=*имя_файла* – Направляет вывод сообщений сервера в файл *имя_файла*. Файл создаётся, если он ещё не существует. При этом устанавливается umask 077, что предотвращает доступ других пользователей к этому файлу.

-m *режим,* --mode=*режим* – Задаёт режим остановки кластера. Значением *режим* может быть smart, fast или immediate, либо первая буква этих вариантов. Если этот ключ опущен, по умолчанию выбирается режим fast.

-o *параметры,* --options=*параметры* – Указывает параметры, которые будут передаваться непосредственно программе postgres. Ключ -o можно указывать несколько раз, при этом ей будут переданы параметры из всех ключей.

Задаваемые *параметры* обычно следует обрамлять одинарными или двойными кавычками, чтобы они передавались одной группой.

-o *параметры-initdb,*

--options=*параметры-initdb* – Указывает параметры, которые будут передаваться непосредственно программе initdb. Ключ -o можно указывать несколько раз, при этом ей будут переданы параметры из всех ключей.

Задаваемые *параметры-initdb* обычно следует обрамлять одинарными или двойными кавычками, чтобы они передавались вместе одной группой.

-p *путь* – Указывает размещение исполняемого файла postgres. По умолчанию задействуется исполняемый файл postgres из того же каталога, из которого запускался pg_ctl, а если это невозможно, из жёстко заданного каталога инсталляции.

-s, --silent – Выводить лишь ошибки, без сообщений информационного характера.

-t *секунды,* --timeout=*секунды* – Задаёт максимальное время (в секундах) ожидания завершения операции. По умолчанию действует значение переменной среды PGCTLTIMEOUT или, если оно не задано, 60 секунд.

-V, --version – Выводит версию pg_ctl и прерывает выполнение.

-W, --no-wait – Не ждать завершения операции. Этот режим противоположен режиму -w.

Если ожидание отключено, запрошенное действие вызывается, но о его результате ничего не известно. В этом случае для проверки текущего состояния и результата операции потребуется обратиться к файлу журнала сервера или воспользоваться внешней системой мониторинга.

-?, --help – Вывести справку по команде pg_ctl и прервать выполнение.

-w, --wait – Ждать завершения операции. Этот режим поддерживается (и действует по умолчанию) для команд start, stop, restart, promote и register.

Файлы:

postmaster.pid – Проверяя этот файл в каталоге данных, pg_ctl определяет, работает ли сервер в настоящий момент.

postmaster.opts – Если файл существует в каталоге хранения данных, то pg_ctl (при restart) передаст его содержимое в качестве аргументов postgres, если не указаны иные значения в -o. Содержимое файла также отображается при вызове в режиме status.

В процессе ожидания pg_ctl постоянно проверяет PID-файл сервера, приостанавливаясь на короткое время между проверками. Запуск считается завершённым, когда PID-файл указывает на то, что сервер готов принимать подключения. Остановка считается завершённой, когда сервер удаляет свой PID-файл.

Программа pg_ctl возвращает код завершения в зависимости от успеха запуска или остановки. Если операция не заканчивается за отведённое время, программа pg_ctl завершается с ненулевым кодом выхода.

4.2. Входные и выходные данные программы initdb для создания кластера баз данных PG360:

Параметры:

-A *authmethod,* --auth=*authmethod* – Параметр определяет метод аутентификации по умолчанию для локальных пользователей, используемый в файле pg_hba.conf (строки host и local). Программа initdb предварительно внесёт указанный метод аутентификации в pg_hba.conf в записи как обычных соединений, так и соединений репликации.

--auth-host=*authmethod* – Параметр указывает метод аутентификации для локальных пользователей, подключающихся по TCP/IP, используемый в pg_hba.conf (строки host).

--auth-local=*authmethod* – Параметр выбирает метод аутентификации локальных пользователей, подключающихся через Unix-сокет, используемый в pg_hba.conf (строки local).

-D *каталог,* --pgdata=*каталог* – Параметр указывает каталог хранения данных кластера. Это единственный обязательный параметр для команды initdb. При этом его можно указать в переменной окружения PGDATA, что будет удобным при дальнейшем использовании (postgres обращается к этой же переменной).

-E *кодировка,* --encoding=*кодировка* – Устанавливает кодировку шаблона и новых баз данных по умолчанию, если не указать иное при их создании. По умолчанию устанавливается исходя из указанной локали, и далее, если не удалось определить, выбирается SQL_ASCII.

-g, --allow-group-access – Позволяет пользователям, входящим в группу владельца кластера, читать все файлы кластера, создаваемые программой initdb. В Windows этот ключ не работает, так как там не поддерживаются разрешения для группы в стиле POSIX.

-k, --data-checksums – Применять контрольные суммы на страницах данных для выявления сбоев при вводе/выводе, которые иначе останутся незамеченными. Расчёт контрольных сумм может повлечь заметное снижение производительности. Когда контрольные суммы включены, они рассчитываются для всех объектов и во всех базах данных. Все ошибки контрольных сумм будут видны в представлении pg_stat_database.

--locale=*локаль* – Устанавливает локаль кластера по умолчанию. Если флаг не указан, локаль устанавливается согласно окружению, в котором исполняется команда initdb.

-N, --no-sync – По умолчанию initdb ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром initdb завершается быстрее, без ожидания, но в случае неожиданного сбоя ОС каталог данных может оказаться испорченным. Этот параметр может быть полезен при тестировании; в производственной среде применять его не следует.

--pwfile=*имя_файла* – Принуждает initdb читать пароль суперпользователя базы данных из файла, первая строка которого используется в качестве пароля.

-S, --sync-only – Безопасно записывает все файлы базы на диск и останавливается. Другие операции initdb при этом не выполняются.

-T *конфигурация,* --text-search-config=*конфигурация* – Устанавливает конфигурацию текстового поиска по умолчанию.

-U *имя_пользователя,* --username=*имя_пользователя* – Устанавливает имя суперпользователя базы данных. По умолчанию используется имя пользователя ОС, запустившего initdb. По факту, само по себе имя суперпользователя базы данных не важно, но этот параметр позволяет оставить привычное postgres, если имя пользователя ОС другое.

-W, --pwprompt – Указывает initdb запросить пароль, который будет назначен суперпользователю базы данных. Это не важно, если не планируется использовать аутентификацию по паролю. В ином случае этот режим аутентификации оказывается неприменимым, пока пароль не задан.

-X *каталог,* --waldir=*каталог* – Этот параметр указывает каталог для хранения журнала упреждающей записи.

--wal-segsize=*размер* – Задаёт *размер сегмента WAL*, в мегабайтах. Такой размер будет иметь каждый отдельный файл в журнале WAL. По умолчанию размер равен 16 мегабайтам. Значение должно задаваться степенью 2 от 1 до 1024 (в мегабайтах). Этот параметр можно установить только во время инициализации и нельзя изменить позже.

-V, --version – Выводит версию initdb и останавливается.

-?, --help – Показывает помощь по аргументам команды initdb и останавливается.

4.3. Входные и выходные данные программы pg_archivecleanup для вычищения файлов архивов WAL PG360:

Параметры:

-d – Выводить подробные отладочные сообщения в stderr.

-n – Вывести имена файлов, которые должны быть удалены, в stdout (не выполняя удаление).

-V, --version – Вывести версию pg_archivecleanup и завершиться.

-x *расширение* – Установить расширение, которое будет убрано из имён файлов до принятия решения об удалении определённых файлов.

-?, --help – Вывести справку об аргументах командной строки pg_archivecleanup и завершиться.

4.3. Входные и выходные данные программы pg_checksums для включения, отключения или проверки контрольных сумм данных в кластере PG360:

Параметры:

-D *каталог,* --pgdata=*каталог* – Указывает каталог, в котором располагается кластер баз данных.

-c, --check – Запускает проверку контрольных сумм. Это режим по умолчанию, который выбирается, когда не указан никакой другой.

-d, --disable – Отключает контрольные суммы.

-e, --enable – Включает контрольные суммы.

-f *файловый_узел,* --filenode=*файловый_узел* – Проверять контрольные суммы только в отношении, которому соответствует указанный *файловый_узел*.

-N, --no-sync – По умолчанию pg_checksums ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром pg_checksums завершается быстрее, без ожидания, но в случае неожиданного сбоя ОС каталог с изменёнными файлами может повредиться.

-p, --progress – Включает вывод сообщений о прогрессе. Эти сообщения будут выводиться при проверке или включении контрольных сумм.

-v, --verbose – Выводить подробные сообщения, в частности список всех проверенных файлов.

-V, --version – Выводит версию pg_checksums и завершает работу.

-?, --help – Показывает справку по аргументам командной строки pg_checksums и завершает работу.

4.3. Входные и выходные данные программы pg_controldata для вывода управляющей информации кластера баз данных PG360:

– datadir – каталог кластера баз данных.

Параметры:

-V, -– version – вывод версии и прервать выполнение.

-?, --help – отобразить помощь по поддерживаемым командой аргументам.

4.3. Входные и выходные данные программы pg_resetwal для очистки журнала упреждающей записи и другой управляющей информации кластера PG360:

– каталог_данных – каталог кластера баз данных;

-f, --force – Принудительно выполнять pg_resetwal, даже если не удаётся получить приемлемые данные из pg_control.

-n, --dry-run – С ключом -n/--dry-run команда pg_resetwal отображает значения, извлечённые из pg_control, а также значения, которые планируется изменить, и завершается, не внося никаких изменений.

-V, --version – Показать версию, а затем завершиться.

-?, --help – Показать справку, а затем завершиться.

Следующие параметры необходимы, только когда pg_resetwal не может определить подходящие значения, прочитав pg_control. Безопасные значения можно определить, как описано ниже. Для значений, принимающих числовые аргументы, можно задать шестнадцатеричные значения, добавив префикс 0x.

-c *xid,xid*

--commit-timestamp-ids=*xid,xid* – Вручную задать идентификаторы старейшей и новейшей транзакций, для которых можно получить время фиксации.

-e *эпоха_xid,* --epoch=*эпоха_xid* – Вручную задать эпоху в ID следующей транзакции.

-l *walfile,* --next-wal-file=*walfile* – Вручную задать начальную позицию WAL, указав имя файла следующего сегмента WAL.

-m *mxid,mxid,* --multixact-ids=*mxid,mxid* – Вручную задать ID следующей и старейшей мультитранзакции.

-o *oid,* --next-oid=*oid* – Вручную задать следующий OID.

-O *mxoff,* --multixact-offset=*mxoff* – Вручную задать смещение следующей мультитранзакции.

--wal-segsize=*размер_сегмента_wal* – Задать другой размер сегмента WAL, в мегабайтах. Значение параметра должно быть степенью 2 от 1 до 1024 (в мегабайтах).

-u *xid,* --oldest-transaction-id=*xid* – Вручную задать ID старейшей незамороженной транзакции.

-x *xid,* --next-transaction-id=*xid* – Вручную задать ID следующей транзакции.

4.3. Входные и выходные данные программы pg_rewind для синхронизации каталога данных PG360 с другим каталогом, ответвлённым от него:

-D *каталог,* --target-pgdata=*каталог* – Этот параметр задаёт целевой каталог данных, который будет синхронизирован с источником. Целевой сервер должен быть отключён штатным образом до запуска pg_rewind

--source-pgdata=*каталог* – Задаёт путь в файловой системе к каталогу данных исходного сервера, с которым будет синхронизироваться целевой. Для применения этого ключа исходный сервер должен быть остановлен штатным образом.

--source-server=*строка_подключения* – Задаёт строку подключения libpq для подключения к исходному серверу PG360, с которым будет синхронизирован целевой. Подключение должно устанавливаться как обычное (не реплицирующее) от имени роли, имеющей необходимые права для выполнения функций pg_rewind на исходном сервере (подробнее об этом говорится в Замечаниях), или от имени суперпользователя. Для применения этого параметра исходный сервер должен быть запущен и работать не в режиме восстановления.

-R, --write-recovery-conf – Создать standby.signal и добавить параметры подключения в postgresql.auto.conf в выходном каталоге. Этот ключ требует указания --source-server.

-n, --dry-run – Делать всё, кроме внесения изменений в целевой каталог.

-N, --no-sync – По умолчанию pg_rewind ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром pg_rewind завершается быстрее, без ожидания, но в случае неожиданного сбоя ОС целевой каталог данных может оказаться испорченным.

-P, --progress – Включает вывод сообщений о прогрессе. При этом в процессе копирования данных из исходного кластера будет выдаваться приблизительный процент выполнения.

-c, --restore-target-wal – Использовать команду restore_command, определённую в конфигурации целевого кластера, для получения файлов WAL из архива в случае отсутствия их в каталоге pg_wal.

--debug – Выводить подробные отладочные сообщения.

--no-ensure-shutdown – Для pg_rewind необходимо, чтобы целевой сервер был остановлен штатным образом до синхронизации. По умолчанию, если целевой сервер не был остановлен штатно, pg_rewind запускает его в однопользовательском режиме, чтобы он выполнил процедуру восстановления после сбоя, а затем останавливает его. Когда передаётся данный ключ, pg_rewind не делает этого, а завершается с ошибкой немедленно, если сервер не был остановлен корректно. Ожидается, что в этом случае пользователи сами исправят сложившуюся ситуацию.

-V, --version – Показать версию, а затем завершиться.

-?, --help – Показать справку, а затем завершиться.

4.3. Входные и выходные данные программы pg_test_fsync для подбора наилучшего варианта wal_sync_method для PG360:

Параметры:

-f, --filename – Задаёт имя файла для записи данных тестов. Этот файл должен находиться в той же файловой системе, где размещается или будет размещаться каталог pg_wal. (В каталоге pg_wal содержатся файлы WAL.) По умолчанию выбирается файл pg_test_fsync.out в текущем каталоге.

-s, --secs-per-test – Задаёт продолжительность каждого теста (в секундах). Чем больше длится тест, тем точнее результат, но тем дольше работает программа. Со значением по умолчанию (5 секунд) программа должна завершиться примерно за 2 минуты.

-V, --version – Вывести версию pg_test_fsync и завершиться.

-?, --help – Вывести справку об аргументах командной строки pg_test_fsync и завершиться.

4.3. Входные и выходные данные программы pg_test_timing для определения издержки замера времени:

Параметры:

pg_test_timing принимает следующие аргументы командной строки:

-d *длительность,* --duration=*длительность* – Задаёт продолжительность теста (в секундах). Чем больше эта продолжительность, тем выше точность и больше вероятность обнаружить аномалию с обратным ходом системных часов. По умолчанию время тестирования – 3 секунды.

-V, --version – Вывести версию pg_test_timing и завершиться.

-?, --help – Вывести справку об аргументах командной строки pg_test_timing и завершиться.

4.3. Входные и выходные данные программы pg_waldump для вывода журнала упреждающей записи кластера БД PG360:

Параметры:

начальный_сегмент – Начать чтение с указанного файла сегмента журнала. Это неявно определяет каталог, в котором будут находиться файлы, и целевую линию времени.

конечный_сегмент – Остановиться после чтения указанного файла сегмента журнала.

-b, --bkp-details – Выводить подробные сведения о блоках-копиях страниц.

-e *конец,* --end=*конец* – Прекратить чтение в заданной позиции в WAL, а не читать поток до конца.

-f, --follow – Достигнув конца корректного WAL, проверять раз в секунду поступление новых записей WAL.

-n *предел,* --limit=*предел* – Вывести заданное число записей и остановиться.

-p *путь,* --path=*путь* – Задаёт каталог, содержащий файлы сегментов журнала, либо каталог с подкаталогом pg_wal, содержащим такие файлы. По умолчанию в поисках этих файлов просматривается текущий каталог, подкаталог pg_wal текущего каталога и подкаталог pg_wal каталога PGDATA.

-q, --quiet – Не выводить ничего кроме ошибок.

-r *менеджер_ресурсов,* --rmgr=*менеджер_ресурсов* – Выводить только записи, созданные указанным менеджером ресурсов. Когда в качестве имени менеджера передаётся list, программа выводит только список возможных имён менеджеров ресурсов и завершается.

-s *начало,* --start=*начало* – Позиция в WAL, с которой нужно начать чтение. По умолчанию чтение начинается с первой корректной записи журнала в самом первом из найденных файлов.

-t *линия_времени,* --timeline=*линия_времени* – Линия времени, из которой будут читаться записи журнала. По умолчанию используется значение, заданное параметром *начальный_сегмент*, если он присутствует, а иначе – 1.

-V, --version – Вывести версию pg_waldump и завершиться.

-x *ид_транзакции,* --xid=*ид_транзакции* – Вывести только записи, относящиеся к указанной транзакции.

-z, --stats[=record] – Вывести общую статистику (число и размер записей и образов полных страниц) вместо отдельных записей. Возможен вариант получения статистики по записям, а не по менеджерам ресурсов.

-?, --help – Вывести справку об аргументах командной строки pg_waldump и завершиться.

4.3. Входные и выходные данные программы postgres для управления сервером баз данных PG360:

Параметры:

-B *количество-буферов* – Устанавливает количество разделяемых между процессами буферов. Значение по умолчанию выбирается автоматически при развёртывании кластера с помощью initdb. Установка флага аналогична конфигурации параметра shared_buffers.

-c *имя=значение* – Устанавливает заданный параметр времени выполнения. Конфигурационные параметры, поддерживаемые PG360. Большинство других параметров командной строки на самом деле представляют собой краткие формы такого присваивания значений параметрам. Для установления нескольких параметров -c можно указывать многократно.

-C *имя* – Отображает значение заданного параметра времени выполнения и завершается. Можно применять на работающем сервере, при этом будут возвращены значения postgresql.conf с учётом проведённых в рамках вызова изменений. Значения, переданные при старте кластера, не отображаются.

-d *уровень-отладки* – Устанавливает уровень отладки (от 1 до 5). Чем выше значение, тем подробнее осуществляется вывод в журнал сервера. Также возможно передать -d 0 для отдельной сессии, что предотвратит в её рамках влияние выставленного для postgres значения.

-D *datadir* – Указывает размещение конфигурационных файлов базы в пределах файловой системы.

-e – Устанавливает формат вводимых дат по умолчанию в «European» с последовательностью значений DMY. Также влияет на вывод дня, идущего перед значением месяца.

-F – Отключает вызовы fsync для увеличения производительности, но с увеличением рисков потери данных в случае краха системы. Этот параметр работает аналогично параметру конфигурации fsync.

-h *компьютер* – Указывает IP-адрес или имя компьютера, на котором сервер postgres принимает клиентские подключения по TCP/IP. Значением может быть список адресов, разделённых запятыми, либо символ *, обозначающий все доступные интерфейсы. Если значение опущено, то подключения принимаются только через Unix-сокеты. По умолчанию принимаются подключения только к localhost.

-k *каталог* – Указывает каталог Unix-сокета, через который postgres будет принимать подключения. Значением параметра может быть список каталогов через запятую. Если это значение пустое, использование Unix-сокетов запрещается, разрешаются только подключения по TCP/IP. По умолчанию выбирается каталог /tmp, но его можно сменить на этапе компиляции.

-l – Включает поддержку безопасных соединений с использованием SSL шифрования.

-N *максимальное количество соединений* – Устанавливает максимально возможное количество одновременных клиентских соединений. Значение по умолчанию устанавливается автоматически на этапе развёртывания с помощью initdb.

-p *порт* – Указывает порт TCP/IP или расширение файла локального Unix-сокета, через который postgres принимает подключения клиентских приложений. По умолчанию принимает значение переменной окружения PGPORT, или, если значение PGPORT не установлено, то используется значение, установленное на этапе компиляции (обычно это 5432). Если значение порта меняется, то на стороне клиентов это необходимо учитывать, установив, либо PGPORT, либо флаг командной строки.

-s – Отображает информацию о времени и другую статистику после каждой выполненной команды.

-S *рабочая-память* – Указывает базовый объём памяти, который сервер будет использовать для сортировок и хеш– таблиц, прежде чем прибегнуть к использованию временных файлов на диске.

-V, --version – Отображает версию postgres и прерывает дальнейшее выполнение.

--*имя=значение* – Устанавливает заданный параметр времени выполнения.

--describe-config – Выводит значения конфигурационных переменных сервера, их описаний и значений по умолчанию в формате команды COPY со знаком табуляции в качестве разделителя. В основном это предназначено для средств администрирования.

-?, --help – Выводит помощь по аргументам команды postgres.

4.3. Входные и выходные данные программы clusterdb для повторной кластеризации таблиц базы данных PG360:

Параметры:

-a, --all – Кластеризовать все базы данных.

[-d] *имя_бд,* [--dbname=]*имя_бд* -Указывает имя базы данных для кластеризации, когда не используется параметр -a/--all. Если это указание отсутствует, имя базы определяется переменной окружения PGDATABASE. Если эта переменная не установлена, именем базы будет имя пользователя, указанное для подключения. В аргументе *имя_бд* может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

-e, --echo – Вывести команды к серверу, генерируемые при выполнении clusterdb.

-q, --quiet – Подавлять вывод сообщений о прогрессе выполнения.

-t *таблица,* --table=*таблица* – Кластеризовать *таблицу*. Возможно множественное использование параметра -t.

-v, --verbose – Вывести подробную информацию во время процесса.

-V, --version – Вывести версию clusterdb и прервать дальнейшее выполнение.

-?, --help – Вывести справку по аргументам команды clusterdb.

Параметры подключения:

-h *сервер,* --host=*сервер* – Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

-p *порт,* --port=*порт* – Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

-U *имя_пользователя,* --username=*имя_пользователя* – Имя пользователя, под которым производится подключение.

-w, --no-password – Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл .pgpass, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

-W, --password – Принудительно запрашивать пароль перед подключением к базе данных.

--maintenance-db=*имя_бд* – Указывает имя базы данных, к которой будет выполняться подключение для определения подлежащих кластеризации баз данных, когда используется ключ -a/--all. Если это имя не указано, будет выбрана база postgres, а если она не существует – template1. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке. Кроме того, все параметры в строке подключения, за исключением имени базы, будут использоваться и при подключении к другим базам данных.

4.3. Входные и выходные данные программы createdb для создания базы данных PG360:

Параметры:

имя_бд – Указывает имя создаваемой базы. Имя должно быть уникальным в рамках кластера PG360. По умолчанию в качестве имени базы данных берётся имя текущего системного пользователя.

Описание – Добавляет комментарий к создаваемой базе.

-D *табличное_пространство* – --tablespace=*табличное_пространство* – Указывает табличное пространство, используемое по умолчанию. Имя пространства обрабатывается аналогично идентификаторам, заключённым в двойные кавычки.

-e, --echo – Вывести команды к серверу, генерируемые при выполнении createdb.

-E *кодировка,* --encoding=*кодировка* – Указывает кодировку базы данных.

-l *локаль,* --locale=*локаль* – Указывает локаль базы данных. Имеет эффект одновременно установленных флагов --lc– collate и --lc-ctype.

--lc-collate=*локаль* – Устанавливает параметр LC_COLLATE для базы данных.

--lc-ctype=*локаль* – Устанавливает параметр LC_CTYPE для базы данных.

-O *владелец,* --owner=*владелец* – Указывает пользователя в качестве владельца создаваемой базы. Имя пользователя обрабатывается аналогично идентификаторам, заключённым в двойные кавычки.

-T *шаблон,* --template=*шаблон* – Указывает шаблон, на основе которого будет создана база данных. Имя шаблона обрабатывается аналогично идентификаторам, заключённым в двойные кавычки.

-V, --version – Вывести версию createdb и прервать дальнейшее исполнение.

-?, --help – Вывести помощь по команде createdb и прервать выполнение.

Флаги -D, -l, -E, -O и -T по назначению соответствуют флагам SQL-команды CREATE DATABASE. createdb также принимает из командной строки параметры подключения:

-h *сервер,* --host=*сервер* – Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

-p *порт,* – --port=*порт* – Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

-U *имя_пользователя,* --username=*имя_пользователя* – Имя пользователя, под которым производится подключение.

-w, --no-password – Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл .pgpass, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

-W, --password – Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как createdb запрашивает пароль автоматически, если сервер проверяет подлинность по паролю.

--maintenance-db=*имя_бд* – Указывает имя опорной базы данных, к которой будет произведено подключение для создания новой. Если имя не указано, будет выбрана база postgres, а если она не существует – template1. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

4.3. Входные и выходные данные программы createuser для создания новой учётной записи PG360:

Параметры:

имя_пользователя – Задаёт имя создаваемого пользователя PG360.

-c *номер,* --connection-limit=*номер* – Устанавливает максимальное допустимое количество соединений для создаваемого пользователя. По умолчанию ограничение в количестве соединений отсутствует.

-d, --createdb – Разрешает новому пользователю создавать базы данных.

-D, --no-createdb – Запрещает новому пользователю создавать базы данных. Это поведение по умолчанию.

-e, --echo – Вывести команды к серверу, генерируемые при выполнении createuser.

-E, --encrypted – Параметр является устаревшим, но в целях совместимости ещё работает.

-g *role,* --role=*role* – Указывает роль, к которой будет добавлена текущая роль в качестве члена группы. Допускается множественное использование флага -g.

-i, --inherit – Создаваемая роль автоматически унаследует права ролей, в которые она включается. Это поведение по умолчанию.

-I, --no-inherit – Роль не будет наследовать права ролей, в которые она включается.

--interactive – Запросить имя для создаваемого пользователя, а также значения для флагов -d/-D, -r/-R, -s/– S, если они явно не указаны в командной строке.

-l, --login – Новый пользователь сможет подключаться к серверу (то есть его имя может быть идентификатором начального пользователя сеанса). Это свойство по умолчанию.

-L, --no-login – Новый пользователь не сможет подключаться к серверу. (Роль без права входа на сервер тем не менее используется для управления разрешениями в базе данных.)

-P, --pwprompt – Если флаг указан, то createuser запросит пароль для создаваемого пользователя. Если не планируется аутентификация по паролю, то пароль можно не устанавливать.

-r, --createrole – Разрешает новому пользователю создавать другие роли, что означает наделение привилегией CREATEROLE.

-R, --no-createrole – Запрещает пользователю создавать новые роли. Это поведение по умолчанию.

-s, --superuser – Создаваемая роль будет иметь права суперпользователя.

S, --no-superuser – Новый пользователь не будет суперпользователем. Это поведение по умолчанию.

-V, --version – Вывести версию createuser и завершить выполнение.

--replication – Создаваемый пользователь будет наделён правом REPLICATION

--no-replication – Создаваемый пользователь не будет иметь привилегии REPLICATION.

-?, --help – Вывести помощь по команде createuser.

Параметры подключения:

-h *сервер,* --host=*сервер* – Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

-p *порт,* --port=*порт* – Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

-U *имя_пользователя,* --username=*имя_пользователя* – Имя пользователя для подключения (не имя создаваемого пользователя).

-w, --no-password – Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл .pgpass, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

-W, --password – Принудительно запрашивать пароль перед подключением к базе данных.

4.3. Входные и выходные данные программы dropdb для удаления базы данных PG360:

Параметры:

имя_бд – Указывает имя удаляемой базы данных.

-e, --echo – Вывести команды к серверу, генерируемые при выполнении dropdb.

-f, --force – Попытаться принудительно завершить все существующие подключения к целевой базе, прежде чем удалять её. Подробнее это описано в DROP DATABASE.

-i, --interactive – Выводит вопрос о подтверждении перед удалением.

-V, --version – Выводит версию dropdb.

--if-exists – Не считать ошибкой, если база данных не существует. В этом случае будет выдано замечание.

-?, --help – Вывести справку по команде dropdb.

dropdb также принимает из командной строки параметры подключения:

-h *сервер*, --host=*сервер* – Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

-p *порт*, --port=*порт* – Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

-U *имя_пользователя,* --username=*имя_пользователя* – Имя пользователя, под которым производится подключение.

-w, --no-password – Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл .pgpass, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

-W, --password – Принудительно запрашивать пароль перед подключением к базе данных.

--maintenance-db=*имя_бд* – Указывает имя опорной базы данных, к которой будет произведено подключение для удаления целевой. Если имя не указано, будет выбрана база postgres, а если она не существует (или именно она и удаляется) – template1. Здесь может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

4.3. Входные и выходные данные программы dropuser для удаления учётной записи пользователя PG360:

имя_пользователя – Указывает имя удаляемой роли PG360. Если передан флаг -i/--interactive, а имя не указано в параметрах команды, его необходимо будет ввести интерактивно.

-e, --echo – Вывести команды к серверу, генерируемые при выполнении dropuser.

-i, --interactive – Вывести подтверждение об удалении роли, и запросить её имя, если оно не указано в параметрах команды.

-V, --version – Вывести версию dropuser.

--if-exists – Перехватить ошибку, если пользователь не существует. В этом случае вместо ошибки будет выведено информационное сообщение.

-?, --help – Вывести справку по команде dropuser.

dropuser также принимает из командной строки параметры подключения:

-h *сервер*, --host=*сервер* – Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

-p *порт,* --port=*порт* – Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

-U *имя_пользователя,* --username=*имя_пользователя* – Имя пользователя, под которым производится текущее подключение к базе.

-w, --no-password – Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл .pgpass, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

-W, --password – Принудительно запрашивать пароль перед подключением к базе данных.

4.3. Входные и выходные данные программы ecpg для запуска встроенного C-препроцессора SQL:

-c – Автоматически генерировать код, написанный на языке C, из кода SQL. Сейчас это справедливо для EXEC SQL TYPE.

-C *режим* – Установить режим совместимости; *режим* может принимать значения: INFORMIX, INFORMIX_SE и ORACLE.

-D *символ* – Определить символ начала команд C-препроцессора.

-h – Обрабатывать заголовочные файлы. Когда добавляется этот параметр, расширением выходного файла становится не .c, а .h, и расширением входного файла по умолчанию считается не .pgc, а .pgh. Кроме этого с данным параметром подразумевается -c.

-i – Также разбирать и системные включения.

-I *каталог* – Указать дополнительный путь включаемых файлов, используемый при выполнении EXEC SQL INCLUDE. По умолчанию используются . (текущий каталог), /usr/local/include, каталог, задаваемый при компиляции PG360 (обычно – /usr/local/pgsql/include), и /usr/ include, в порядке, как это перечислено.

-o *имя_файла* – Задаёт *имя файла*, в который программа ecpg должна выводить результат. Указание -o – направляет результат в устройство стандартного вывода.

-r *параметр* – Определяет поведение времени выполнения. *Флаг* может принимать следующие значения:

no_indicator – Использовать специальные символы для представления значений null. Исторически некоторые базы данных используют такой подход.

prepare – Сформировать подготовленные выражения. libecpg сформирует кеш подготовленных выражений и будет использовать их при необходимости повторно. В случае переполнения кеша, libecpg освободит память за счёт вытеснения наименее используемых выражений.

Questionmarks – Разрешает использовать знак вопроса в качестве аргумента подстановки в целях совместимости. Ранее это было поведением по умолчанию.

-t, Включить автоматическую фиксацию транзакций. В этом режиме каждая SQL-команда будет автоматически фиксироваться, пока не будет явно включена в блок транзакции. В режиме по умолчанию команды фиксируются лишь при явном вызове EXEC SQL COMMIT.

-v – Вывести информацию о версии, а также путях поиска включаемых файлов.

--version – Вывести версию ecpg.

-?, --help – Вывести справку по команде ecpg.

Программы на C со встроенным SQL необходимо скомпоновать с библиотекой libecpg, например, используя флаг компоновщика -L/usr/local/pgsql/lib -lecpg.

4.3. Входные и выходные данные программы pg_basebackup для создания резервной копии кластера PG360:

-D *каталог,* --pgdata=*каталог* – Целевой каталог, куда будет записана копия. Если он не существует, pg_basebackup создаст его, а также отсутствующие родительские каталоги, при необходимости. Если он существует, он должен быть пустым.

Если копия создаётся в формате tar, в качестве целевого каталога можно задать – (минус), и тогда файл tar будет записан в stdout.

Этот флаг является обязательным.

-F *формат,* --format=*формат* – Устанавливает формат вывода. *формат* может принимать следующие значения:

p plain – Записывает выводимые данные в обычные файлы, сохраняя структуру каталогов данных и табличных пространств как на исходном сервере. Если в кластере нет дополнительных табличных пространств, вся база будет помещена в заданный каталог. Иначе основной каталог хранения данных будет помещён в целевой каталог, а все остальные табличные пространства – в те же абсолютные пути, в которых они располагаются на исходном сервере. (Чтобы изменить эти пути, необходимо воспользоваться параметром --tablespace-mapping).

Это формат по умолчанию.

t tar – Записывает в целевой каталог файлы в формате tar. Содержимое основного каталога данных будет записано в файл base.tar, а каждое дополнительное табличное пространство – в отдельный файл, содержащий в имени OID этого пространства.

Если в качестве имени целевого каталога задано – (минус), содержимое tar будет записано в устройство стандартного вывода, что позволяет, например, передать его программе gzip. Это возможно, только если в кластере нет дополнительных табличных пространств и не используется трансляция WAL.

-R, --write-recovery-conf – Создать файл standby.signal и добавить параметры конфигурации в файл postgresql.auto.conf в целевом каталоге (или внутри архива, если используется формат tar). Это упрощает настройку ведомого сервера при восстановлении этой копии.

В файл postgresql.auto.conf будут записаны параметры соединения и слот репликации, если его использует pg_basebackup, так что впоследствии при потоковой репликации будут использоваться те же параметры.

-T *старый_каталог=новый_каталог,* --tablespace-mapping=*старый_каталог=новый_каталог* – Переместить табличное пространство из *старого_каталога* в *новый_каталог* в процессе копирования. Чтобы перемещение произошло, в параметре *старый_каталог* путь табличного пространства должен задаваться в точности так, как он определён на исходном сервере. (Но не будет ошибкой, если на исходном сервере не окажется табличного пространства, на которое указывает *старый_каталог*.) В то же время, *новый_каталог* задаёт путь в файловой системе получающего сервера. Как и основной целевой каталог, *новый_каталог* может не существовать, но если он существует, он должен быть пустым. И *старый_каталог*, и *новый_каталог* должны задаваться абсолютными путями. Если в пути встречается символ =, его необходимо экранировать обратной косой чертой. Этот параметр можно добавить несколько раз для нескольких табличных пространств.

Если табличное пространство перемещается таким способом, символические ссылки внутри основного каталога хранения данных также приводятся в соответствие с новым местоположением. Таким образом, для экземпляра сервера подготавливается новый каталог данных, в котором все табличные пространства оказываются в новом расположении.

В настоящее время этот параметр работает только с обычным форматом вывода; если выбран формат tar, параметр игнорируется.

--waldir=*каталог_wal* – Задать каталог, в который будут записаны файлы WAL (журнал упреждающей записи). По умолчанию файлы WAL будут записываться в подкаталог pg_wal целевого каталога, но с помощью этого параметра их можно поместить в любое место. Путь *каталог_wal* должен быть абсолютным. Как и основной целевой каталог, *каталог_wal* может не

существовать, но если он существует, он должен быть пустым. Этот параметр можно задать, только если копия создаётся в простом формате.

-X *метод,* --wal-method=*метод* – Включает в резервную копию все необходимые файлы журнала упреждающей записи (файлы WAL). В том числе включаются все файлы журнала, сгенерированные в процессе создания резервной копии. Любой метод, кроме none, позволяет запустить сервер с восстановленным каталогом, не используя архив WAL; таким образом будет получена полностью самодостаточная резервная копия.

Поддерживаются следующие *методы* получения журналов упреждающей записи:

n none – Не включать журналы упреждающей записи в резервную копию.

f fetch – Файлы журнала упреждающей записи собираются в конце процесса копирования. Таким образом необходимо установить достаточно большое значение параметра wal_keep_size, чтобы избежать преждевременного удаления нужных данных журнала. В случае переработки этих данных до завершения процесса копирования возникнет ошибка, а копия будет непригодной к использованию.

Когда используется формат tar, файлы журнала упреждающей записи включаются в архив base.tar.

s stream – Передавать журнал упреждающей записи в процессе создания резервной копии. При выборе этого метода открывается второе соединение к серверу, через которое будет передаваться журнал упреждающей записи параллельно с созданием копии. Таким образом, этот метод требует использования не одного, а двух соединений репликации, но если клиент будет успевать получать данные журнала упреждающей записи, на исходном сервере не потребуется сохранять дополнительные журналы.

Когда используется формат tar, файлы журнала упреждающей записи сохраняются в отдельном архиве с именем pg_wal.tar. Это значение по умолчанию.

-z,--gzip – Включает gzip-сжатие выводимого tar-файла с уровнем компрессии по умолчанию. Сжатие поддерживается только для формата tar, при этом ко всем именам файлов tar добавляется суффикс .gz.

-Z *уровень,* --compress=*уровень* – Включает gzip-сжатие выводимого tar-файла и задаёт уровень сжатия от 0 (без сжатия) до 9 (максимальное сжатие). Сжатие поддерживается только для формата tar, при этом ко всем именам файлов tar добавляется суффикс .gz.

Аргументы командной строки управляющие формированием резервной копии и вызовом программы:

-c *fast*|*spread*, --checkpoint=*fast*|*spread* – Устанавливает режим контрольных точек: fast (быстрый) или spread (протяжённый, по умолчанию).

-C, --create-slot – Указывает, что до начала копирования должен быть создан слот репликации с именем, заданным в --slot. Если такой слот уже существует, выдаётся ошибка.

-l *метка,* --label=*метка* – Устанавливает метку для созданной резервной копии. Если не указана, то по умолчанию будет использовано значение «pg_basebackup base backup».

-n, --no-clean – По умолчанию, когда программа pg_basebackup прерывается с ошибкой, она удаляет все каталоги, которые она могла создать, прежде чем обнаружила, что не может завершить задание (например, целевой каталог и каталог журнала упреждающей записи). Данный ключ отключает эту очистку и тем самым полезен для отладки.

Каталоги табличных пространств не очищаются в любом случае.

-N, --no-sync – По умолчанию pg_basebackup ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром pg_basebackup завершается немедленно, то есть выполняется быстрее, но в случае неожиданного сбоя ОС резервная копия может оказаться испорченной. Вообще этот параметр предназначен прежде всего для тестирования, для производственной среды он не подходит.

-P, --progress – Включает отчёт о прогрессе. Если этот режим включён, то во время создания копии будет передаваться примерный процент выполнения. Так как данные в базе могут меняться во время копирования, это значение будет лишь приближённым и может достигать не точно 100%. В частности, когда в копию включается журнал WAL, конечный размер невозможно предсказать заранее, и в этом случае ожидаемый конечный размер будет увеличиваться, превысив ориентировочный полный размер без WAL.

-r *скорость_передачи,* --max-rate=*скорость_передачи* – Задаёт максимальную скорость, с которой данные будут загружаться с исходного сервера. Это может быть использовано для ограничения влияния pg_basebackup на сервер. Значение параметра задаётся в килобайтах в секунду. Для указания мегабайт в секунду нужно добавить к числу суффикс M. Так же принимается суффикс k, но он ничего не меняет. Допустимые значения лежат в диапазоне от 32 КБ/с до 1024 МБ/с.

Этот параметр всегда оказывает влияние на передачу каталога данных, а на передачу файлов WAL он влияет, только если выбран метод передачи fetch.

-S *имя_слота,* --slot=*имя_слота* – Этот параметр может применяться только вместе с -X stream. Он выбирает слот репликации, который будет использоваться для передачи WAL. Если базовая копия предназначена для использования на ведомом сервере с потоковой репликацией через слот, это же имя слота должно задаваться на ведомом в качестве primary_slot_name. Тем самым гарантируется, что ведущий сервер не удалит никакие необходимые данные WAL после того, как базовая копия будет получена, и до того, как начнётся потоковая репликация на новом ведомом.

В случае отсутствия ключа -C требуется, чтобы указанный слот репликации уже существовал.

Если этот ключ не указан и сервер поддерживает временные слоты репликации, для трансляции WAL автоматически используется временный слот репликации.

-v, --verbose – Включает режим подробного вывода. Будет выводится некоторая дополнительная информация при начале и завершении, а также имена обрабатываемых файлов, если включён отчёт о прогрессе.

--manifest-checksums=*алгоритм* – Задаёт алгоритм контрольных сумм, которые будут рассчитываться для всех файлов, описанных в манифесте копии.

Для проверки целостности копии по созданному манифесту можно воспользоваться программой pg_verifybackup.

--manifest-force-encode – Принудительно включает шестнадцатеричное кодирование всех имён файлов в манифесте. Если этот параметр не задаётся, в шестнадцатеричном виде кодируются только имена файлов не в кодировке UTF-8. Этот параметр в первую очередь предназначен для проверки того, что средства чтения манифеста могут правильно разобрать такие имена.

--no-estimate-size – Отключает расчёт примерного объёма данных, которые будут передаваться в процессе копирования, в результате чего столбец backup_total в представлении pg_stat_progress_basebackup будет содержать NULL.

Без этого указания процесс копирования начнётся с перечисления файлов для подсчёта размера всей базы данных, а затем продолжится отправкой непосредственно данных. Это может немного увеличить общую длительность процесса, в частности, пройдёт больше времени до начала передачи данных. Данный параметр полезен, когда время расчёта объёма оказывается слишком большим.

Этот параметр нельзя использовать вместе с параметром --progress.

--no-manifest – Отключает создание манифеста копии. Если этот флаг не указан, сервер будет формировать и передавать в составе копии манифест, который может быть проверен с использованием pg_verifybackup. Манифест представляет собой список всех файлов, включённых в копию, за исключением файлов WAL, которые могут быть в неё добавлены. Также в нём сохраняется размер, дата последнего изменения и, возможно, контрольная сумма каждого файла.

--no-slot – Предотвращает создание временного слота репликации для резервного копирования.

По умолчанию, если выбрана передача журнала, но имя слота в -S не задано, создаётся временный слот репликации (при условии, что это поддерживает исходный сервер).

Основное предназначение этого ключа в том, чтобы можно было сделать базовую резервную копию, когда на сервере нет свободных слотов репликации. Использование слота репликации почти всегда предпочтительнее, так как при этом предотвращается удаление сервером необходимых файлов WAL во время резервного копирования.

--no-verify-checksums – Отключает проверку контрольных сумм, если они включены на сервере, с которого делается резервная копия.

По умолчанию контрольные суммы проверяются, и при выявлении их несоответствия выдаётся ненулевой код завершения. Однако базовая резервная копия в этом случае не удаляется, как и с ключом --no-clean. Ошибки контрольных сумм также можно просмотреть в представлении pg_stat_database.

Параметры, управляющие подключением к исходному серверу:

-d *строка_подключения* − --dbname=*строка_подключения* – Указывает параметры подключения к серверу в формате строки подключения; они будут переопределять любые одноимённые параметры, заданные в командной строке.

Параметр называется --dbname для согласованности с другими клиентскими приложениями, но так как pg_basebackup не подключается к какой-либо конкретной базе, любое имя базы данных в строке подключения игнорируется.

-h *сервер,* --host=*сервер* – Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения PGHOST, если она установлена. В противном случае выполняется подключение к Unix-сокету.

-p *порт,* --port=*порт* – Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения PGPORT, если она установлена, либо числом, заданным при компиляции.

-s *interval,* --status-interval=*interval* – Указывает интервал в секундах между сообщениями о состоянии, передаваемыми исходному серверу. Чем меньше указанное значение, тем точнее будет информация о процессе резервного копирования на сервере. Нулевое значение полностью отключает периодическое обновление состояния, хотя эти сообщения будут всё равно посылаться по запросу сервера во избежание отключения по тайм-ауту. Значение по умолчанию – 10 секунд.

-U *имя_пользователя,* --username=*имя_пользователя* – Задаёт имя пользователя для подключения.

-w, --no-password – Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль нельзя получить другими средствами, например из файла
.pgpass, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

-W, --password – Принудительно запрашивать пароль перед подключением к исходному серверу.

Другие флаги:

-V , --version – Вывести версию pg_basebackup и завершиться.

-?, --help – Вывести справку по аргументам командной строки pg_basebackup и завершиться.

4.3. Входные и выходные данные программы pgbench для запуска теста производительности PG360:

имя_бд – Указывает имя базы, в которой будет проводиться тест. Если имя не задано, то используется значение переменной окружения PGDATABASE. Если и переменная не задана, то в качестве имени базы будет взято имя пользователя, под которым осуществляется подключение.

-i, --initialize – Требуется для вызова режима инициализации.

-I *этапы_инициализации,* --init-steps=*этапы_инициализации* – Выполнять только выбранные из всех обычных подготовительных этапов. В параметре *этапы_инициализации* отдельные символы для каждого этапа выбирают, какие этапы должны выполняться. Все этапы выполняются в определённом порядке. Список этапов по умолчанию: dtgvp. Полный перечень подготовительных этапов:

d (Drop, удалить) – Удалить все существующие таблицы pgbench.

t (create Tables, создать таблицы) – Создать таблицы, используемые стандартным сценарием pgbench, а именно:

pgbench_accounts, pgbench_branches, pgbench_history и pgbench_tellers.

g или G (Generate data, сгенерировать данные на стороне клиента или на стороне сервера) – Сгенерировать данные и загрузить их в стандартные таблицы, заменив все уже существующие данные.

С ключом g (выбирающим генерирование данных на стороне клиента), данные формируются в клиентском коде pgbench, а затем передаются на сервер. При этом соединение клиент/сервер нагружается командой COPY. Когда с ключом g генерируются данные для таблицы pgbench_accounts, после каждых 100000 строк выдаётся сообщение о прогрессе.

С ключом G (выбирающим генерирование данных на стороне сервера), клиентский код pgbench передаёт на сервер только небольшие запросы, а собственно формированием данных

занимается сервер. В этом случае сетевое соединение не нагружается, но возрастает нагрузка на сервер. При генерировании данных с ключом G никакие сообщения о ходе операции не выдаются.

По умолчанию при инициализации базы данные генерируются на стороне клиента (то есть подразумевается ключ g).

v (Vacuum, очистка) – Вызывать VACUUM для стандартных таблиц.

p (create Primary keys, создать первичные ключи) – Создать первичные ключи в стандартных таблицах.

f (create Foreign keys, создать внешние ключи) – Создать ограничения внешних ключей между стандартными таблицами.

-F *фактор_заполнения,* --fillfactor=*фактор_заполнения* – Создать таблицы pgbench_accounts, pgbench_tellers и pgbench_branches с заданным фактором заполнения. Значение по умолчанию – 100.

-n, --no-vacuum – Не выполнять очистку во время инициализации. (Этот параметр выключает этап инициализации v, даже если он был указан в -I.)

-q, --quiet – Переключить вывод в немногословный режим, когда выводится только одно сообщение о прогрессе в 5 секунд. В режиме по умолчанию одно сообщение выводится на каждые 100000 строк, при этом за секунду обычно выводится довольно много строк (особенно на хорошем оборудовании).

Этот параметр не оказывает влияния, если в -I выбран вариант G.

-s *коэффициент_масштаба,* --scale=*коэффициент_масштаба* – Умножить число генерируемых строк на заданный коэффициент. Например, с ключом -s 100 в таблицу pgbench_accounts будут записаны 10 000 000 строк. Значение по умолчанию – 1. При коэффициенте, равном 20 000 или больше, столбцы, содержащие идентификаторы счетов (столбцы aid), перейдут к большим целым числам (типу bigint), чтобы в них могли уместиться все возможные значения идентификаторов.

--foreign-keys – Создать ограничения внешних ключей между стандартными таблицами. (Этот ключ добавляет этап f к последовательности подготовительных этапов, если он отсутствует.)

--index-tablespace=*табл_пространство_индексов* – Создать индексы в указанном табличном пространстве, а не в пространстве по умолчанию.

--partition-method=*ИМЯ* – Создать секционированную таблицу pgbench_accounts, применив метод *ИМЯ* (это может быть range или hash). Для использования этого параметра необходимо, чтобы было задано ненулевое значение --partitions. Если этот параметр не указывается, подразумевается метод range.

--partitions=*ЧИСЛО* – Создать секционированную таблицу pgbench_accounts с заданным *ЧИСЛОМ* секций примерно равного размера в соответствии с масштабированным количеством счетов. По умолчанию подразумевается число 0, то есть таблица не секционируется.

--tablespace=*табличное_пространство* – Создать таблицы в указанном табличном пространстве, а не в пространстве по умолчанию.

--unlogged-tables – Создать все таблицы как нежурналируемые, а не как постоянные таблицы.

Параметры тестирования производительности

pgbench принимает следующие аргументы командной строки для тестирования производительности:

-b *имя_скрипта[@вес]*, --builtin=*имя_скрипта[@вес]* – Добавляет в список скриптов, которые будут выполняться, указанный встроенный скрипт. В число встроенных скриптов входят tpcb-like, simple-update и select-only. Также принимаются однозначные начала их имён. Со специальным именем list программа выводит список встроенных скриптов и немедленно завершается.

Дополнительно можно задать целочисленный вес после @, меняющий вероятность выбора этого скрипта относительно других. По умолчанию вес считается равным 1. Подробности следуют ниже.

-c *клиенты,* --client=*клиенты* – Число имитируемых клиентов, то есть число одновременных сеансов базы данных. Значение по умолчанию – 1.

-C, --connect – Устанавливать новое подключение для каждой транзакции вместо одного для каждого клиента.

-d, --debug – Выводить отладочные сообщения.

-D *имя_переменной=значение,* --define=*имя_переменной=значение* – Определить переменную для пользовательского скрипта. Параметр -D может добавляться неоднократно.

-f *имя_файла[@вес],* --file=*имя_файла[@вес]*

Добавить в список выполняемых скриптов скрипт транзакции из файла *имя_файла*.

-j *потоки,* --jobs=*потоки* – Число рабочих потоков в pgbench. Значение по умолчанию – 1.

-l, --log – Записать информацию о каждой транзакции в файл протокола. Подробности описаны ниже.

-L *предел,* --latency-limit=*предел* – Транзакции, продолжающиеся дольше указанного *предела* (в миллисекундах), подсчитываются и отмечаются отдельно, как *опаздывающие*.

В режиме ограничения скорости (--rate=...) транзакции, которые отстают от графика более чем на заданный *предел* (в мс) и поэтому никак не могут уложиться в отведённый интервал, не передаются серверу вовсе. Они подсчитываются и отмечаются отдельно как *пропущенные*.

-M *режим_запросов,* --protocol=*режим_запросов*

Протокол, выбираемый для передачи запросов на сервер:

1. simple: использовать протокол простых запросов.
2. extended: использовать протокол расширенных запросов.
3. prepared: использовать протокол расширенных запросов с подготовленными операторами.

В режиме prepared pgbench повторно использует результат разбора запроса, начиная со второй итерации, и поэтому работает быстрее, чем в других режимах.

По умолчанию выбирается протокол простых запросов.

-n, --no-vacuum – Не производить очистку таблиц перед запуском теста. Этот параметр *необходим*, если применять собственный сценарий, не затрагивающий стандартные таблицы pgbench_accounts, pgbench_branches, pgbench_history и pgbench_tellers.

-N, --skip-some-updates – Запустить встроенный упрощённый скрипт simple-update. Краткий вариант записи -b simple– update.

-P *сек,* --progress=*сек* – Выводить отчёт о прогрессе через заданное число секунд (*сек*). Выдаваемый отчёт включает время, прошедшее с момента запуска, скорость (в TPS) с момента предыдущего отчёта, а также среднее время ожидания транзакций и стандартное отклонение. В режиме ограничения скорости (-R) время ожидания вычисляется относительно назначенного времени запуска транзакции, а не фактического времени её начала, так что оно включает и среднее время отставания от графика.

-r, --report-latencies – Выводить по завершении тестировании средняя время ожидания операторов (время выполнения с точки зрения клиента) для каждой команды. Подробности описаны ниже.

-R *скорость передачи,* --rate=*скорость передачи* – Выполнять транзакции, ориентируясь на заданную скорость, а не максимально быстро (по умолчанию). Скорость задаётся в транзакциях в секунду. Если заданная скорость превышает максимально возможную, это ограничение скорости не повлияет на результаты.

Для получения нужной скорости транзакции запускаются со случайными задержками, имеющими распределение Пуассона. При этом запланированное время запуска отсчитывается от начального времени, а не от завершения предыдущей транзакции. Это означает, что если какие-то транзакции отстанут от изначально рассчитанного времени завершения, всё же возможно, что последующие нагонят график.

В режиме ограничения скорости время ожидания транзакций, выводимое по итогам тестирования, вычисляется, исходя из запланированного времени запуска, так что в него входит время, которое очередная транзакция должна была ждать завершения предыдущей транзакции. Это время называется временем отклонения от графика, и его среднее и максимальное значения выводятся отдельно. Время ожидания транзакций с момента их фактического запуска, то есть время, потраченное на выполнение транзакций в базе данных, можно получить, если вычесть время отклонения от графика из времени ожидания транзакций.

Если ограничение --latency-limit задаётся вместе с --rate, транзакция может заведомо не вписываться в отведённое ей время, если предыдущая транзакция завершится слишком поздно, так как ожидаемое время окончания транзакции отсчитывается от времени запуска по графику. Такие транзакции не передаются серверу, а пропускаются и подсчитываются отдельно.

Большое значение отклонения от графика свидетельствует о том, что система не успевает выполнять транзакции с заданной скоростью и выбранным числом клиентов и потоков. Когда среднее время ожидания транзакции превышает запланированный интервал между транзакциями, каждая последующая транзакция будет отставать от графика, и чем дольше будет выполняться тестирование, тем больше будет отставание. Когда это наблюдается, нужно уменьшить скорость транзакций.

-s *коэффициент_масштаба,* --scale=*коэффициент_масштаба* – Показать заданный коэффициент масштаба в выводе pgbench.

-S , --select-only – Запустить встроенный скрипт select-only (только выборка). Краткий вариант записи -b select– only.

-t *транзакции,* --transactions=*транзакции* – Число транзакций, которые будут выполняться каждым клиентом (по умолчанию 10).

-T *секунды,* --time=*секунды* – Выполнять тест с ограничением по времени (в секундах), а не по числу транзакций для каждого клиента. Параметры -t и -T являются взаимоисключающими.

-v, --vacuum-all – Очищать все четыре стандартные таблицы перед запуском теста. Без параметров -n и -v pgbench будет очищать от старых записей таблицы pgbench_tellers и pgbench_branches, а также опустошать pgbench_history.

--aggregate-interval=*секунды* – Длительность интервала агрегации (в секундах). Может использоваться только с ключом -l. С данным параметром в протокол выводится сводка по интервалам, как описано ниже.

--log-prefix=*префикс* – Задать префикс имён файлов для файлов протоколов, создаваемых с ключом --log. Префикс по умолчанию – pgbench_log.

--progress-timestamp – При отображении прогресса (с параметром -P) выводить текущее время (в формате Unix), а не количество секунд от начала запуска. Время задаётся в секундах с точностью до миллисекунд.

--random-seed=*затравка* – Установить затравку для генератора случайных чисел. Инициализирует генератор случайных чисел, который затем выдаёт последовательность начальных состояний отдельных генераторов для каждого потока. *затравка* может принимать следующие значения: time (по умолчанию, затравка базируется на текущем времени), rand (задействовать надёжный генератор случайных чисел или выдать ошибку, если он отсутствует) или беззнаковое десятичное число. Генератор случайных чисел может вызываться явно из скрипта pgbench (функциями random...) или неявно (например, для планирования выполнения транзакций с ключом -– rate). В случае установки значения явным образом оно выводится в терминале. Любое значение, допустимое в качестве *затравки*, можно также задать в переменной окружения PGBENCH_RANDOM_SEED. Чтобы заданная затравка применялась во всех возможных случаях использования, необходимо задать этот параметр первым или установить переменную окружения.

--sampling-rate=*скорость передачи* – Частота выборки для записи данных в протокол, изменяя которую можно уменьшить объём протокола. При указании этого параметра в протокол выводится информация только о заданном проценте транзакций. Со значением 1.0 в нём будут отмечаться все транзакции, а с 0.05 только 5%.

--show-script=*имя_скрипта* – Вывести код встроенного скрипта *имя_скрипта* в stderr и сразу завершиться.

Программа pgbench также принимает общие аргументы командой строки, определяющие параметры подключения:

-h *компьютер* , --host=*компьютер* – Адрес сервера баз данных.

-p *порт,*--port=*порт* – Номер порта сервера баз данных.

-U *имя_пользователя,* --username=*имя_пользователя* – Имя пользователя для подключения

-V, --version – Вывести версию pgbench и завершиться.

-?, --help – Вывести справку об аргументах командной строки pgbench и завершиться.

Код завершения:

В случае успешного выполнения возвращается код 0. Код завершения 1 указывает на статичные проблемы, например, ошибки в параметрах командной строки. При возникновении ошибок во время выполнения, например при обращении к базе данных или выполнении скрипта, выдаётся код завершения 2. В последнем случае pgbench выведет частичные результаты.

4.3. Входные и выходные данные программы pg_config для вывода информации об установленной версии PG360:

--bindir – Вывести расположение исполняемых файлов. Можно использовать, например, для поиска программы psql. Обычно там же находится и сама программа pg_config.

--docdir – Вывести расположение файлов документации.

--htmldir – Вывести расположение файлов документации в формате HTML.

--includedir – Вывести расположение заголовочных C-файлов клиентских интерфейсов.

--pkgincludedir – Вывести расположение других заголовочных C-файлов.

--includedir-server – Вывести расположение заголовочных C-файлов для программирования серверной части.

--libdir – Вывести расположение библиотек объектного кода.

--pkglibdir – Вывести расположение динамически подгружаемых модулей, либо путь, где сервер должен их искать. По этому пути также могут размещаться и другие архитектурно-зависимые файлы.

--localedir – Вывести расположение файлов поддержки локалей. Если поддержка локалей не была сконфигурирована на этапе сборки PG360, будет выведена пустая строка.

--mandir – Вывести расположение страниц руководства man.

--sharedir – Вывести расположение архитектурно-независимых вспомогательных файлов.

--sysconfdir – Вывести расположение системных конфигурационных файлов.

--pgxs Вывести расположение файлов сборки расширений.

--configure – Вывести список параметров configure, использованных при сборке PG360.

--cc – Вывести использованное при сборке PG360 значение переменной CC. Оно отражает, какой C-компилятор применялся.

--cppflags – Вывести использованное при сборке PG360 значение переменной CPPFLAGS. Оно отражает флаги C-компилятора, применённые для препроцессора. Обычно это флаги -I.

--cflags – Вывести использованное при сборке PG360 значение переменной CFLAGS. Оно отражает флаги C-компилятора, применённые при сборке.

--cflags_sl – Вывести использованное при сборке PG360 значение переменной CFLAGS_SL. Оно отражает дополнительные флаги C-компилятора для сборки разделяемых библиотек.

--ldflags – Вывести использованное при сборке PG360 значение переменной LDFLAGS. Оно отражает флаги компоновщика.

--ldflags_ex – Вывести использованное при сборке PG360 значение переменной LDFLAGS_EX. Оно отражает флаги компоновщика, использованные при сборке лишь исполняемых файлов.

--ldflags_sl – Вывести использованное при сборке PG360 значение переменной LDFLAGS_SL. Оно отражает флаги компоновщика, использованные при сборке лишь разделяемых библиотек.

--libs – Вывести использованное при сборке PG360 значение переменной LIBS. Обычно оно отражает флаги подключения внешних библиотек к PG360, переданные с ключом -l.

--version – Вывести версию PG360.

-?, --help – Вывести справку по команде pg_config.

Если одновременно передано несколько параметров, то выводимая информация будет следовать согласно их порядку. Если параметры не переданы, то будет выведена вся информация с подписями, к чему она относится.

4.3. Входные и выходные данные программы pg_dump для выгрузки базы данных PG360 в виде скрипта или в архивном формате.

Параметры командной строки для управления содержимым и форматом вывода:

имя_бд – Указывает имя базы данных, из которой будут выгружаться данные. Если имя не задано, то используется значение переменной окружения PGDATABASE. Если и переменная не задана, то в качестве имени базы будет взято имя пользователя, под которым осуществляется подключение.

-a, --data-only – Выводить только данные, но не схемы объектов (DDL). Будут копироваться данные таблиц, большие объекты, значения последовательностей.

-b, --blobs – Включить большие объекты в выгрузку. Это поведение по умолчанию при отсутствии ключей --schema, --table или --schema-only. Таким образом, ключ -b полезен, лишь когда нужно добавить большие объекты при выгрузке только избранной схемы или таблицы. Большие объекты относятся к данным, и поэтому будут выгружаться, когда используется ключ --data-only, но не ключ --schema-only.

-B, --no-blobs – Исключить из выгрузки большие объекты.

Когда задаётся и -b, и -B, большие объекты при выгрузке данных будут выводиться.

-c, --clean – Включить в выходной файл команды удаления (DROP) объектов базы данных перед командами создания (CREATE) этих объектов. Если дополнительно не указать флаг --if-exists, то при восстановлении в базу данных, где некоторые объекты отсутствуют, попытка удаления несуществующего объекта будет приводить к ошибке, которую можно игнорировать.

Этот параметр игнорируется, когда данные выгружаются в архивных форматах (не в текстовом). Для таких форматов данный параметр можно указать при вызове pg_restore.

-C, --create – Сформировать в начале вывода команду для создания базы данных и затем подключения к ней. В этом случае не важно, какая база указана в параметрах подключения перед выполнением скрипта. Также, если указан ключ --clean, то скрипт сначала удалит, а затем пересоздаст базу данных перед подключением к ней.

С ключом --create в выходной файл также включается комментарий к базе данных (если он задан) и все назначения переменных конфигурации, связанные с базой данных, то есть все команды ALTER DATABASE ... SET ... и ALTER ROLE ... IN DATABASE ... SET ..., ссылающиеся на эту базу данных. Также выгружаются права доступа к самой базе данных, если не добавлен ключ --no-acl.

Этот параметр игнорируется, когда данные выгружаются в архивных форматах (не в текстовом). Для таких форматов данный параметр можно указать при вызове pg_restore.

-E *кодировка*, --encoding=*кодировка* – Создать копию в заданной кодировке. По умолчанию копия создаётся в кодировке, используемой базой данных.

-f *файл,* --file=*файл* – Отправить вывод в указанный файл. Параметр можно не указывать, если используется формат с выводом в файл. В этом случае будет использован стандартный вывод. Однако для формата с выводом в каталог параметр является обязательным и должен задавать путь

к каталогу. В этом случае целевой каталог будет создан командой pg_dump и не должен существовать заранее.

-F *format,* --format=*format* – Указывает формат вывода копии. *format* может принимать следующие значения:

p plain – Сформировать текстовый SQL-скрипт. Это поведение по умолчанию.

c custom – Выгрузить данные в специальном архивном формате, пригодном для дальнейшего использования программой pg_restore. Наряду с форматом directory является наиболее гибким форматом, позволяющим вручную выбирать и сортировать восстанавливаемые объекты. Вывод в этом формате по умолчанию сжимается.

d directory – Выгрузить данные в формате каталога. Этот формат пригоден для дальнейшего использования программой pg_restore. При этом будет создан каталог, в котором для каждой таблицы и большого объекта будут созданы отдельные файлы, а также файл оглавления в машинно-читаемом формате, понятном для pg_restore. С полученной резервной копией можно работать штатными средствами Unix, например, несжатую копию можно сжать посредством gzip. Этот формат по умолчанию сжимается, а также поддерживает работу в несколько потоков.

t tar – Выгрузить данные в формате tar, для дальнейшего использования с программой pg_restore. Этот формат совместим с форматом вывода в каталог: если архив распаковать, получится корректная копия в формате каталога. Однако формат tar не поддерживает сжатие. Также, применяя формат tar, при восстановлении нельзя изменить относительный порядок элементов данных

-j *число_заданий,* --jobs=*число_заданий* – Осуществить выгрузку в параллельном режиме, обрабатывая одновременно несколько таблиц (в количестве *число_заданий*). Это может сократить время, необходимое для выгрузки, но увеличивает нагрузку на сервер. Этот параметр можно использовать только с форматом вывода в каталог, так как это единственный формат, позволяющий нескольким процессам записывать данные одновременно.

pg_dump откроет *число_заданий* + 1 соединений с базой данных. Таким образом необходимо обеспечить достаточное значение параметра max_connections.

-n *шаблон,* --schema=*шаблон* – Выгрузить только схемы, соответствующие *шаблону*; вместе с этими схемами будут выгружены и все содержащиеся в них объекты. Когда этот параметр отсутствует, выгружаются все несистемные схемы в целевой базе данных. Чтобы выгрузить несколько схем, ключ -n можно указать несколько раз. Параметр *шаблон* интерпретируется по тем же правилам, что и шаблон в командах psql \d, так что несколько схем можно выбрать и шаблоном со знаками подстановки. Используя знаки подстановки, при необходимости нужно заключать шаблон в кавычки.

-N *шаблон,* --exclude-schema=*шаблон* – Не выгружать схемы, соответствующие *шаблону*. Шаблон интерпретируется по тем же правилам, что и для параметра -n. Параметр -N можно использовать в команде несколько раз для исключения схем, соответствующих нескольким шаблонам.

При одновременном использовании параметров -n и -N будут выгружаться схемы, соответствующие шаблону параметра -n и не противоречащие шаблону параметра -N.

-O, --no-owner – Не формировать команды, устанавливающие владельца объектов базы данных. По умолчанию pg_dump генерирует команды ALTER OWNER или SET SESSION

AUTHORIZATION для назначения владельцев объектов базы. Эти команды завершатся неудачно, если скрипт будет запущен не суперпользователем или не владельцем объектов. Чтобы создать скрипт, который можно выполнить при восстановлении от лица произвольного пользователя и назначить его в качестве владельца объектов восстанавливаемой базы, необходимо указать флаг -O.

Этот параметр игнорируется, когда данные выгружаются в архивных форматах (не в текстовом). Для таких форматов данный параметр можно указать при вызове pg_restore.

-R, --no-reconnect – Параметр является устаревшим, но в целях совместимости ещё работает.

-s, --schema-only – Выгружать только определения объектов (схемы), без данных.

-S *имя_пользователя,* --superuser=*имя_пользователя* Указать суперпользователя, который будет использоваться для отключения триггеров. Параметр имеет значение только вместе с --disable-triggers. Обычно его лучше не использовать, а запускать полученный скрипт от имени суперпользователя.

-t *шаблон,* --table=*шаблон* – Выгрузить только таблицы, соответствующие *шаблону*. Чтобы выбрать несколько таблиц, ключ -t можно указать несколько раз. Параметр *шаблон* интерпретируется по тем же правилам, что и шаблон в командах psql \d, так что несколько таблиц можно выбрать и шаблоном со знаками подстановки. Используя знаки подстановки, при необходимости нужно заключать шаблон в кавычки.

Параметры -n и -N не действуют в присутствии параметра -t, так как отобранные им таблицы всё равно будут выгружены, а не табличные объекты выгружаться не будут.

-T *шаблон* , --exclude-table=*шаблон* – Не выгружать таблицы, соответствующие *шаблону*. Шаблон интерпретируется по тем же правилам, что и для параметра -t. Параметр -T можно использовать в команде несколько раз для исключения таблиц, соответствующих нескольким шаблонам.

При одновременном использовании параметров -t и -T будут выгружаться таблицы, соответствующие шаблону параметра -t и не противоречащие шаблону параметра -T.

-v , --verbose – Включить подробный режим. pg_dump будет выводить в стандартный поток ошибок подробные комментарии к объектам, включая время начала и окончания выгрузки, а также сообщения о прогрессе выполнения.

-V, --version – Вывести версию pg_dump.

-x, --no-privileges, --no-acl – Не выгружать права доступа (команды GRANT/REVOKE).

-Z *0..9* , --compress=*0..9* – Установить уровень сжатия данных. Ноль означает, что сжатие выключено. Для специального формата и формата каталога будут сжиматься файлы отдельных таблиц. По умолчанию применяется умеренный уровень сжатия. Если указать отличный от нулевого уровень сжатия для простого формата, то сжиматься будет весь выходной файл, как это было бы при передаче файла команде gzip. Однако по умолчанию для простого формата сжатие не производится. Формат tar в настоящий момент не поддерживает сжатие.

--binary-upgrade – Этот параметр предназначен для программ обновления сервера. Использование для иных целей не рекомендуется и не поддерживается. Поведение параметра может быть изменено в последующих версиях без предварительного уведомления.

--column-inserts, --attribute-inserts – Выгружать данные таблиц в виде команд INSERT с явным указанием столбцов (INSERT INTO *таблица* (*столбец*, ...) VALUES ...). Скорость восстановления при этом значительно снизится, но данный вариант оправдан, когда загружать данные нужно не в PG360. При этом в случае каких-либо ошибок при загрузке данных будут потеряны только строки INSERT, где возникли ошибки, но не всё содержимое таблицы.

--disable-dollar-quoting – Этот параметр запрещает заключать в доллары тело функций, что оставляет возможность только заключать их в кавычки, применяя стандартный синтаксис SQL.

--disable-triggers – Используется при выгрузке одних данных. Указывает pg_dump включать в вывод команды для временного выключения триггеров при восстановлении в целевой базе данных. Применяется в ситуациях, когда существуют проверки ссылочной целостности или другие триггеры, которые необходимо выключить на время восстановления.

В настоящее время команды, генерируемые с параметром --disable-triggers, должны исполняться от имени суперпользователя. Таким образом, необходимо также передавать флаг -S, либо при восстановлении выполнять скрипт от имени суперпользователя.

Этот параметр игнорируется, когда данные выгружаются в архивных форматах (не в текстовом). Для таких форматов данный параметр можно указать при вызове pg_restore.

--enable-row-security – Этот параметр имеет смысл только при выгрузке содержимого таблицы, для которой включена защита строк. По умолчанию pg_dump устанавливает для row_security значение off, чтобы убедиться, что выгружаются все данные из таблицы. Если пользователь не имеет достаточных прав для обхода защиты строк, выдаётся ошибка. Этот параметр указывает pg_dump включить row_security, что позволит пользователю выгрузить часть содержимого таблицы, к которой он имеет доступ.

В настоящее время для использования этого параметра обычно желательно, чтобы данные были выгружены в формате INSERT, так как команда COPY FROM в процессе восстановления не поддерживает защиту строк.

--exclude-table-data=*шаблон* – Не выгружать содержимое таблиц, соответствующих *шаблону*. Шаблон таблицы интерпретируется по тем же правилам, что и для параметра -t. Параметр --exclude– table-data можно использовать в команде несколько раз для исключения таблиц, соответствующих нескольким шаблонам. Полезно, когда нужно получить определение таблицы, без содержимого.

Чтобы не выгружать содержимое всех таблиц базы используется параметр --schema-only.

--extra-float-digits=*число_цифр* – Выводить числа с плавающей точкой не с максимальной точностью, а с заданным значением extra_float_digits. При выгрузке данных в целях резервного копирования данный параметр использовать не следует.

--if-exists – При очистке целевой базы использовать условные команды (добавлять предложение IF EXISTS). Применяется только с параметром --clean.

--include-foreign-data=*сторонний_сервер* – Выгрузить данные всех сторонних таблиц со стороннего сервера, имя которого соответствует шаблону *сторонний_сервер*. Для выгрузки данных с нескольких сторонних серверов параметр --include-foreign-data можно использовать несколько раз. К тому же, значение *сторонний_сервер* интерпретируется как шаблон, согласно правилам, используемым командами \d программы psql. Поэтому несколько сторонних серверов можно также выбрать, используя в шаблоне символы подстановки. Когда используются символы

подстановки, шаблон лучше экранировать кавычками, чтобы командная оболочка ОС не интерпретировала их по-своему. Единственное отличие от вышеупомянутых правил – шаблон не может быть пустым.

--inserts – Выгружать данные таблиц в виде команд INSERT вместо COPY. Скорость восстановления при этом значительно снизится, но данный вариант оправдан, когда загружать данные нужно не в PG360. При этом в случае каких-либо ошибок при загрузке данных будут потеряны только строки INSERT, где возникли ошибки, но не всё содержимое таблицы. Восстановление может работать некорректно, если у таблицы изменён порядок столбцов. В такой ситуации можно использовать параметр --column-inserts, для которого порядок столбцов не важен, но он работает ещё медленнее.

--load-via-partition-root – При выгрузке данных для секции таблицы выводить команды COPY или INSERT, ссылающиеся на корневую таблицу в иерархии секционирования, а не на эту секцию.

--lock-wait-timeout=*время_ожидания* – Не ждать бесконечно получения разделяемых блокировок таблиц в начале процедуры выгрузки. Вместо этого выдать ошибку, если не удастся заблокировать таблицы за указанное *время_ожидания*. Это время можно задать в любом из форматов, принимаемых командой SET statement_timeout.

--no-comments – Не выгружать комментарии.

--no-publications – Не выгружать публикации.

--no-security-labels – Не выгружать метки безопасности.

--no-subscriptions – Не выгружать подписки.

--no-sync – По умолчанию pg_dump ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром pg_dump завершается немедленно, то есть выполняется быстрее, но в случае неожиданного сбоя ОС выгруженные данные могут оказаться испорченными. Вообще этот параметр предназначен прежде всего для тестирования, для производственной среды он не подходит.

--no-synchronized-snapshots – Позволяет запускать pg_dump -j на серверах с версией ниже чем 9.2. Подробнее в описании параметра -j.

--no-tablespaces – Не формировать команды для указания табличных пространств. При восстановлении все объекты будут создаваться в табличном пространстве по умолчанию.

Этот параметр игнорируется, когда данные выгружаются в архивных форматах (не в текстовом). Для таких форматов данный параметр можно указать при вызове pg_restore.

--no-unlogged-table-data – Не выгружать данные нежурналируемых таблиц. Параметр не влияет на выгрузку определений таблиц, он только подавляет вывод содержимого таблиц. С резервного сервера содержимое нежурналируемых таблиц не выгружается никогда.

--on-conflict-do-nothing – Добавить предложения ON CONFLICT DO NOTHING в команды INSERT. Это указание допускается только при выборе режима --inserts, --column-inserts или --rows-per-insert.

--quote-all-identifiers – Принудительно экранировать все идентификаторы.

--rows-per-insert=*число_строк* – Выгружать данные таблиц в виде команд INSERT вместо COPY. В данном параметре задаётся максимальное число строк для одной команды INSERT. Указанное в нём значение должно быть больше 0. При этом в случае каких-либо ошибок при

загрузке данных будут потеряны только строки INSERT, где возникли ошибки, но не всё содержимое таблицы.

--section=*имя_секции* – Выгружать лишь указанную секцию. Имя секции может принимать значения pre-data, data или post-data. Для выгрузки нескольких секций, параметр можно использовать несколько раз в одной команде. По умолчанию резервируются все секции.

Секция data содержит непосредственно данные таблиц, больших объектов и значения последовательностей. Секция post-data содержит определения индексов, триггеров, правил и ограничений (кроме ограничений проверки, созданных без NOT VALID). Секция pre-data включает определения остальных элементов.

--serializable-deferrable – Использовать при выгрузке транзакцию с уровнем изоляции serializable для получения снимка, согласованного с последующими состояниями базы.

--snapshot=*имя_снимка* – Использовать заданный синхронный снимок при выгрузке данных из базы.

--strict-names – Требует, чтобы каждому указанию схемы (-n/--schema) и таблицы (-t/--table) соответствовала минимум одна схема/таблица в выгружаемой базе данных.

--use-set-session-authorization – Выводить команды SET SESSION AUTHORIZATION, соответствующие стандарту, вместо ALTER OWNER, для назначения владельцев объектов. В результате выгруженный скрипт будет более стандартизированным, но может не восстановиться корректно, в зависимости от истории объектов. Кроме того, для использования SET SESSION AUTHORIZATION при восстановлении нужны права суперпользователя, в то время как ALTER OWNER требует меньших привилегий.

-?, --help – Показать справку по аргументам командной строки pg_dump и завершиться.

Параметры управления подключением:

-d *имя_бд,* --dbname=*имя_бд* – Указывает имя базы данных для подключения. Равнозначно указанию *имя_бд* в первом аргументе, не являющемся ключом, в командной строке. Вместо имени может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

-h *сервер,* --host=*сервер* – Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения PGHOST, если она установлена. В противном случае выполняется подключение к Unix-сокету.

-p *порт,* --port=*порт* – Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения PGPORT, если она установлена, либо числом, заданным при компиляции.

-U *имя_пользователя,* --username=*имя_пользователя* – Имя пользователя, под которым производится подключение.

-w, --no-password – Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл .pgpass, попытка соединения не удастся.

-W, --password – Принудительно запрашивать пароль перед подключением к базе данных.

--role=*имя роли* – Задаёт имя роли, которая будет осуществлять выгрузку. Получив это имя, pg_dump выполнит SET ROLE *имя_роли* после подключения к базе данных.

4.3. Входные и выходные данные программы pg_dumpall для выгрузки кластера баз данных PG360 в формате скрипта.

Параметры командной строки для управления содержимым и форматом вывода:

-a, --data-only – Выгружать только данные, без схемы (определений данных).

-c, --clean – Добавить команды SQL для удаления (DROP) баз данных перед командами, создающими их. В дополнение к ним добавляются команды DROP для ролей и табличных пространств.

-E *кодировка,* --encoding=*кодировка* – Создать копию в заданной кодировке. По умолчанию копия создаётся в кодировке базы данных. (Другой способ достичь того же результата – задать желаемую кодировку в переменной окружения PGCLIENTENCODING.)

-f *имя_файла,* --file=*имя_файла* – Направить вывод в указанный файл. Если этот параметр опущен, скрипт записывается в стандартный вывод.

-g, --globals-only – Выгружать только глобальные объекты (роли и табличные пространства), без баз данных.

-O, --no-owner – Не генерировать команды, устанавливающие владение объектами, как в исходной базе данных. По умолчанию, pg_dumpall генерирует команды ALTER OWNER или SET SESSION AUTHORIZATION, восстанавливающие исходных владельцев для создаваемых элементов схемы. Однако выполнить эти команды сможет только суперпользователь (или пользователь, владеющий всеми объектами, создаваемыми скриптом). Чтобы получить скрипт, который сможет восстановить любой пользователь (но при этом он станет владельцем всех объектов), используется -O.

-r, --roles-only – Выгружать только роли, без баз данных и табличных пространств.

-s, --schema-only – Выгружать только определения объектов (схемы), без данных.

-S *имя_пользователя,* --superuser=*имя_пользователя* – Указать суперпользователя, который будет использоваться для отключения триггеров. Параметр имеет значение только вместе с --disable-triggers. Обычно его лучше не использовать, а запускать полученный скрипт от имени суперпользователя.

-t, --tablespaces-only – Выгружать только табличные пространства, без баз данных и ролей.

-v, --verbose – Включить режим подробных сообщений. В этом режиме pg_dumpall записывает в выходной файл время начала/завершения выгрузки, а в стандартный канал ошибок – сообщения о процессе. При этом подробные сообщения будет также выводить pg_dump.

-V, --version – Сообщить версию pg_dumpall и завершиться.

-x, --no-privileges, --no-acl – Не выгружать права доступа (команды GRANT/REVOKE).

--binary-upgrade – Этот параметр предназначен для программ обновления сервера. Использование для иных целей не рекомендуется и не поддерживается. Поведение параметра может быть изменено в последующих версиях без предварительного уведомления.

--column-inserts, --attribute-inserts – Выгружать данные в виде команд INSERT с явно задаваемыми именами столбцов (INSERT INTO *таблица* (*столбец*, ...) VALUES ...). При этом восстановление будет очень медленным; в основном это применяется для выгрузки данных, которые затем будут загружаться не в PG360.

--disable-dollar-quoting – Этот параметр запрещает заключать в доллары тело функций, что оставляет возможность только заключать их в кавычки, применяя стандартный синтаксис SQL.

--disable-triggers – Этот параметр действует только при выгрузке одних данных. С ним pg_dumpall добавляет команды, отключающие триггеры в целевых таблицах на время загрузки данных. Необходимо использовать его, если в таблицах определены проверки ссылочной целостности или другие триггеры, которые не нужно выполнять в процессе загрузки данных.

В настоящее время команды, генерируемые с параметром --disable-triggers, должны исполняться от имени суперпользователя. Таким образом, необходимо также передавать флаг -S, либо при восстановлении выполнять скрипт от имени суперпользователя.

--exclude-database=*шаблон* – Не выгружать базы данных, имена которых соответствуют *шаблону*. Исключить имена по нескольким шаблонам можно, добавив несколько ключей --exclude-database. Параметр *шаблон* в данном аргументе обрабатывается по тем же правилам, что и в командах psql \d, что позволяет также исключить несколько баз данных, добавив в шаблон звёздочку.

--extra-float-digits=*число_цифр* – Выводить числа с плавающей точкой не с максимальной точностью, а с заданным значением extra_float_digits. При выгрузке данных в целях резервного копирования данный параметр использовать не следует.

--if-exists – Использовать условные команды (т. е. добавлять предложение IF EXISTS) при удалении базы данных и других объектов. Этот параметр принимается, только если также указан параметр --clean.

--inserts – Выгружать данные в виде команд INSERT, а не COPY. При этом восстановление значительно замедлится; в основном это применяется для выгрузки данных, которые затем будут загружаться не в PG360.

--load-via-partition-root – При выгрузке данных для секции таблицы выводить команды COPY или INSERT, ссылающиеся на корневую таблицу в иерархии секционирования, а не на эту секцию. В результате при загрузке данных подходящая секция будет выбираться заново для каждой строки.

--lock-wait-timeout=*время_ожидания* – Не ждать бесконечно получения разделяемых блокировок таблиц в начале процедуры выгрузки. Вместо этого выдать ошибку, если не удастся заблокировать таблицы за указанное *время_ожидания*. Это время можно задать в любом из форматов, принимаемых командой SET statement_timeout.

--no-comments – Не выгружать комментарии.

--no-publications – Не выгружать публикации.

--no-role-passwords – Не выгружать пароли ролей. При восстановлении все роли получат пароль NULL и не смогут пройти проверку подлинности, пока им не будут назначены пароли. Так как значения паролей не нужны, когда используется это указание, информация о ролях считывается из системного представления pg_roles, а не из pg_authid. Таким образом, данный вариант может быть также полезен, если доступ к pg_authid ограничен политикой безопасности.

--no-security-labels – Не выгружать метки безопасности.

--no-subscriptions – Не выгружать подписки.

--no-sync – По умолчанию pg_dumpall ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром pg_dumpall завершается немедленно, то есть выполняется быстрее, но в случае неожиданного сбоя ОС выгруженные данные могут оказаться поврежденными.

--no-tablespaces – Не выводить команды, создающие или выбирающие табличные пространства для объектов. С этим параметром все объекты будут созданы в пространстве по умолчанию, установленном во время восстановления.

--no-unlogged-table-data – Не выгружать содержимое нежурналируемых таблиц. Этот параметр не влияет на то, как выгружаются определения этих таблиц (схема); он отключает только выгрузку данных из них.

--on-conflict-do-nothing – Добавить предложения ON CONFLICT DO NOTHING в команды INSERT. Это указание допускается только при выборе режима --inserts или --column-inserts.

--quote-all-identifiers – Принудительно экранировать все идентификаторы.

--rows-per-insert=*число_строк* – Выгружать данные таблиц в виде команд INSERT вместо COPY. В данном параметре задаётся максимальное число строк для одной команды INSERT. Указанное в нём значение должно быть больше 0. При этом в случае каких-либо ошибок при восстановлении данных будут потеряны только строки INSERT, где возникли ошибки, но не всё содержимое таблицы.

--use-set-session-authorization – Выводить команды SET SESSION AUTHORIZATION, соответствующие стандарту, вместо ALTER OWNER, для назначения владельцев объектов. В результате выгруженный скрипт будет более стандартизированным, но может не восстановиться корректно, в зависимости от истории объектов.

-?, --help – Показать справку по аргументам командной строки pg_dumpall и завершиться.

Параметры управления подключением:

-d *строка_подключения*, --dbname=*строка_подключения* – Указывает параметры подключения к серверу в формате строки подключения; они будут переопределять любые одноимённые параметры, заданные в командной строке.

Этот параметр называется --dbname для согласованности с другими клиентскими приложениями, но так как pg_dumpall подключается не к одной базе данных, имя базы в строке подключения игнорируется. Чтобы указать имя базы данных для начального подключения, которое будет использоваться для выгрузки глобальных объектов и обнаружения других выгружаемых баз, необходимо воспользоваться параметром -l.

-h *сервер,* --host=*сервер* – Указывает имя компьютера, на котором работает сервер баз данных. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения PGHOST, если она установлена. В противном случае выполняется подключение к Unix-сокету.

-l *имя_бд,* --database=*имя_бд* – Задаёт имя базы данных, через подключение к которой будут выгружаться глобальные объекты и находиться другие выгружаемые базы. По умолчанию используется база postgres, а в случае её отсутствия – template1.

-p *порт,* --port=*порт* – Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения PGPORT, если она установлена, либо числом, заданным при компиляции.

-U *имя_пользователя,* --username=*имя_пользователя* – Имя пользователя, под которым производится подключение.

-w, --no-password – Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл

.pgpass, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

-W, --password – Принудительно запрашивать пароль перед подключением к базе данных.

--role=*имя роли* – Задаёт имя роли, которая будет осуществлять выгрузку. Получив это имя, pg_dumpall выполнит SET ROLE *имя_роли* после подключения к базе данных.

4.3. Входные и выходные данные программы pg_isready для проверки соединения с сервером PG360:

-d *имя_бд*, --dbname=*имя_бд* – Указывает имя базы данных для подключения. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

-h *компьютер,* --host=*компьютер* – Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

-p *порт,* --port=*порт* – Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной среды PGPORT, если она установлена, либо числом, заданным при компиляции, обычно 5432.

-q, --quiet – Не выводить сообщение о состоянии.

-t *секунды,* --timeout=*секунды* – Максимальное время ожидания (в секундах) при попытке подключения, по истечении которого констатируется, что сервер не отвечает. Значение по умолчанию – 3 секунды.

-U *имя_пользователя,* --username=*имя_пользователя* – Подключиться к базе данных с заданным именем пользователя вместо подразумеваемого по умолчанию.

-V, --version – Сообщить версию pg_isready и завершиться.

-?, --help – Показать справку по аргументам командной строки pg_isready и завершиться.

Код завершения:

Программа pg_isready возвращает в оболочку 0, если сервер принимает подключения, 1, если он сбрасывает подключения (например, во время загрузки), 2, если при попытке подключения не получен ответ, и 3, если попытки подключения не было (например, из-за некорректных параметров).

4.3. Входные и выходные данные программы pg_receivewal для приема журнала упреждающей записи с сервера PG360:

-D *каталог*, --directory=*каталог* – Каталог, в который будут записываться данные.

Этот параметр является обязательным.

-E *lsn,* --endpos=*lsn* – Автоматически прекратить репликацию и завершить работу с кодом выхода 0 (без ошибки) при достижении заданного LSN.

Если будет получена запись с LSN, равным заданному *lsn*, она будет обработана.

--if-not-exists – Не выдавать ошибку, когда указан параметр --create-slot и слот с заданным именем уже существует.

-n, --no-loop – Не повторять цикл при ошибках подключения, а сразу завершать работу, возвращая ошибку.

--no-sync – С этим ключом pg_receivewal не будет принудительно сбрасывать данные WAL на диск. Этот вариант быстрее, но при последующем сбое ОС сегменты WAL могут оказаться

испорченными. Обычно этот ключ полезен при тестировании, но при создании архива WAL в производственной среде использовать его не следует.

Этот ключ несовместим с --synchronous.

-s *интервал,* --status-interval=*интервал* – Указывает интервал в секундах между отправками серверу пакетов состояния. Это позволяет упростить мониторинг прогресса. Чтобы выключить периодическое обновление состояния, необходимо установить значение в ноль. При этом обновление будет отправляться по запросу сервера для избежания отсоединения по истечению времени. Значение по умолчанию составляет 10 секунд.

-S *имя_слота,* --slot=*имя_слота* – Указывает pg_receivewal использовать существующий слот репликации. Когда задан этот параметр, pg_receivewal будет сообщать серверу текущую позицию сохранения, отмечая, какой сегмент был сохранён на диске, чтобы сервер мог удалить этот сегмент, если он больше не нужен.

Когда клиент репликации pg_receivewal настроен на сервере как синхронный ведомый сервер, для используемого слота репликации серверу будет передаваться позиция сохранённых данных, но только когда файл WAL закрывается. Таким образом, в такой конфигурации транзакции на ведущем сервере будут ожидать завершения продолжительное время и по сути будут работать неудовлетворительно. Чтобы эта конфигурация работала корректно, нужно дополнительно указать параметр --synchronous.

--synchronous – Сохранять данные WAL на диск сразу после того, как они были получены. Также передавать пакет состояния сразу после сохранения, вне зависимости от --status-interval.

Этот параметр следует указывать, если клиент репликации pg_receivewal настроен на сервере как синхронный ведомый, чтобы обеспечить своевременную передачу ответа серверу.

-v, --verbose – Включает режим подробных сообщений.

-Z *уровень,* --compress=*уровень* – Включает gzip-сжатие журналов упреждающей записи и задаёт уровень сжатия от 0 (без сжатия) до 9 (максимальное сжатие). При этом ко всем именам файлов tar добавляется суффикс .gz.

Параметры управления подключением:

-d *строка_подключения*, --dbname=*строка_подключения* – Указывает параметры подключения к серверу в формате строки подключения; они будут переопределять любые одноимённые параметры, заданные в командной строке.

Параметр называется --dbname для согласованности с другими клиентскими приложениями, но так как pg_receivewal не подключается к какой-либо конкретной базе, это имя в строке подключения игнорируется.

-h *сервер,* --host=*сервер* – Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения PGHOST, если она установлена. В противном случае выполняется подключение к Unix-сокету.

-p *порт,* --port=*порт* – Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения PGPORT, если она установлена, либо числом, заданным при компиляции.

-U *имя_пользователя,* --username=*имя_пользователя* – Имя пользователя для подключения.

-w, --no-password – Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл .pgpass, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

-W, --password – Принудительно запрашивать пароль перед подключением к базе данных.

pg_receivewal может выполнить одно из двух действий в отношении слотов физической репликации:

--create-slot – Создать слот физической репликации с именем, заданным аргументом --slot, и завершиться.

--drop-slot – Удалить слот репликации с именем, заданным аргументом --slot, и завершиться.

Другие флаги:

-V , --version – Сообщить версию pg_receivewal и завершиться.

-?, --help – Вывести справку по аргументам командной строки pg_receivewal и завершиться.

Код завершения

pg_receivewal завершится с кодом 0 при прерывании сигналом SIGINT. (Это штатный способ его завершения, поэтому получение этого сигнала не считается ошибкой.) При критических ошибках или получении других сигналов код завершения будет ненулевым.

4.3. Входные и выходные данные программы pg_ recvlogical для управления потоками логического декодирования PG360.

Для выбора действия указывается минимум один из этих параметров:

--create-slot – Создать новый слот логической репликации с именем, заданным аргументом --slot, используя модуль вывода, заданный аргументом --plugin, для базы данных, указанной в --dbname.

--drop-slot – Удалить слот репликации с именем, заданным аргументом --slot, и завершиться.

--start – Начать приём потока изменений из слота логической репликации с именем, заданным аргументом --slot, и продолжать до сигнала прерывания. Если передача потока прерывается на другой стороне из-за выключения или остановки сервера, цикл подключения и передачи повторяется (если не добавлен параметр --no-loop).

Формат потока определяется модулем вывода, выбранным при создании слота.

Для получения потока подключаться нужно к той же базе, для которой создавался слот.

Параметры --create-slot и --start исключают друг друга. Действие --drop-slot несовместимо с любыми другими действиями.

Следующие параметры командной строки управляют расположением и форматом выводимых данных, а также другим поведением репликации:

-E *lsn*, --endpos=*lsn* – В режиме --start автоматически закончить репликацию и выйти с кодом обычного завершения 0, когда при приёме данных достигается указанный LSN. Если этот ключ указывается не в режиме --start, выдаётся ошибка.

Если встречается запись с LSN, в точности равным *lsn*, эта запись будет выведена.

С указанием --endpos границы транзакций не отслеживаются, так что вывод программы может оказаться обрезанным посередине транзакции. Частично полученная транзакция не будет

считаться принятой и будет воспроизведена заново при следующем чтении из этого слота. Отдельные сообщения не обрезаются никогда.

-f *имя_файла,* --file=*имя_файла* – Записывать полученные и декодированные данные транзакций в указанный файл. Для вывода в stdout укажите – (минус).

-F *секунды,* --fsync-interval=*секунды* – Устанавливает, как часто pg_recvlogical будет вызывать fsync(), чтобы гарантировать, что выходной файл надёжно сохранён на диске.

Сервер время от времени даёт клиенту команду сохранить данные и сообщить сохранённую позицию, но этот параметр позволяет выполнять сохранение чаще.

При значении, равном 0, функция fsync() вообще не вызывается, но серверу сообщается новая позиция. Это может привести к потере данных в случае сбоя.

-I *lsn,* --startpos=*lsn* – В режиме --start репликация начнётся с данного LSN. В других режимах игнорируется.

--if-not-exists – Не выдавать ошибку, когда указан параметр --create-slot и слот с заданным именем уже существует.

-n, --no-loop – Когда подключение к серверу потеряно, не повторять цикл, просто завершить работу.

-o *имя*[=*значение*], --option=*имя*[=*значение*] – Передаёт параметр *имя_параметра* модулю вывода, при этом может быть передано и его *значение*. Набор параметров и их действия зависят от выбранного модуля вывода.

-P *модуль,* --plugin=*модуль* – Использовать указанный модуль вывода логического декодирования при создании слота. Этот параметр не действует, если слот уже существует.

-s *секунды,* --status-interval=*секунды* – Этот параметр действует так же, как одноимённый параметр pg_receivewal.

-S *имя_слота,* --slot=*имя_слота* – Этот параметр задаёт имя слота логической репликации, который будет использоваться в режиме --start, создаваться в режиме --create-slot или удаляться в режиме --drop-slot.

-v, --verbose – Включает режим подробных сообщений.

Параметры управления подключением.

-d *имя_бд,* --dbname=*имя_бд* – Имя базы данных для подключения. Как именно используется данная база, рассказывается в описании действий программы. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке. По умолчанию в качестве имени базы выбирается имя пользователя.

-h *имя_компьютера-или-ipб* —host=*имя_компьютера-или-ip* – Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения PGHOST, если она установлена. В противном случае выполняется подключение к Unix-сокету.

-p *порт,*--port=*порт* – Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения PGPORT, если она установлена, либо числом, заданным при компиляции.

-U *user,* —username=*user* – Имя пользователя для подключения. По умолчанию это имя текущего пользователя ОС.

-wб —no-password – Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл .pgpass, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

-Wб —password – Принудительно запрашивать пароль перед подключением к базе данных.

Дополнительные параметры:

-V,--version – Сообщить версию pg_recvlogical и завершиться.

-?, --help – Показать справку по аргументам командной строки pg_recvlogical и завершиться.

4.3. Входные и выходные данные программы pg_restore для восстановления базы данных PG360 из файла архива, созданного командой pg_dump:

имя_файла – Указывает расположение восстанавливаемого файла архива (или каталога, для архива в формате каталога). По умолчанию используется устройство стандартного ввода.

-a, --data-only – Восстанавливать только данные, без схемы (определений данных). При этом восстанавливаются данные таблиц, большие объекты и значения последовательностей, имеющиеся в архиве.

-c, --clean – Удалить (DROP) объекты базы данных, прежде чем пересоздавать их. (Без дополнительного указания --if-exists при этом могут выдаваться безвредные сообщения об ошибках, если таких объектов не окажется в целевой базе данных.)

-C, --create – Создать базу данных, прежде чем восстанавливать данные. Если также указан параметр -– clean, удалить и пересоздать целевую базу данных перед подключением к ней.

С ключом --create программа pg_restore также восстанавливает комментарий к базе данных (если он задан) и все назначения переменных конфигурации, связанные с базой данных, то есть все команды ALTER DATABASE ... SET ... и ALTER ROLE ... IN DATABASE … SET ..., ссылающиеся на эту базу данных. Также восстанавливаются права доступа к самой базе данных, если не добавлен ключ --no-acl.

С этим параметром база, заданная параметром -d, применяется только для подключения и выполнения начальных команд DROP DATABASE и CREATE DATABASE. Все данные восстанавливаются в базу данных, имя которой записано в архиве.

-d *имя_бд,* --dbname=*имя_бд* – Подключиться к базе данных *имя_базы* и восстановить данные непосредственно в неё. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

-e, --exit-on-error – Завершать работу в случае возникновения ошибки при выполнении команд SQL в базе данных. По умолчанию процесс восстановления продолжается, а по его окончании выдаётся число ошибок.

-f *имя_файла,* --file=*имя_файла* – Задаёт файл для вывода сгенерированного скрипта или списка объектов, получаемого с параметром -l. Чтобы выбрать stdout, необходимо использовать -.

-F *формат,* --format=*формат* – Задаёт формат архива. Указывать формат необязательно, так как pg_restore определяет формат автоматически. Но если формат задаётся, допускается один из этих вариантов:

c custom – Архив сохранён в специальном формате pg_dump.

d directory – Архив сохранён в каталоге.

t tar – Архив сохранён в формате tar.

-I *индекс,* --index=*индекс* – Восстановить определение только заданного индекса. Добавив дополнительные ключи -I, можно указать несколько индексов.

-j *число-заданий,* --jobs=*число-заданий* – Выполнять наиболее длительные этапы pg_restore (в частности, загрузку данных, создание индексов или ограничений) параллельно, используя несколько заданий (в количестве, не превышающем *число-заданий*).

Каждое задание выполняется в отдельном задании или потоке, в зависимости от ОС, и использует отдельное подключение к серверу.

Оптимальное значение этого параметра зависит от аппаратной конфигурации сервера, клиента и сети. В частности, имеет значение количество процессорных ядер и устройство дискового хранилища. В качестве начального значения можно выбрать число ядер на сервере, но и при увеличении этого значения во многих случаях восстановление будет быстрее.

Этот параметр поддерживается только с архивом в специальном формате или в каталоге. Входные данные должны поступать из обычного файла или каталога, а не из канала или стандартного устройства ввода. Кроме того, несколько заданий не могут выполняться в сочетании с параметром --single-transaction.

-l, --list – Вывести оглавление архива.

-L *файл-список,* --use-list=*файл-список* – Восстановить из архива только элементы, перечисленные в *файле-списке*, и в том порядке, в каком они идут в этом файле.

Данный *файл-список* обычно представляет собой отредактированный результат предыдущей операции -l. Строки в нём могут быть переставлены или удалены, а также могут быть закомментированы точкой с запятой (;), добавленной в начале строки.

-n *схема,* --schema=*схема* – Восстановить только объекты в указанной схеме. Добавив дополнительные ключи -n, можно указать несколько схем. Этот параметр можно сочетать с -t, чтобы восстановить только определённую таблицу.

-N *схема,* --exclude-schema=*схема* – Не восстанавливать объекты в указанной схеме. Добавив дополнительные ключи -N, можно исключить несколько схем.

Когда и с ключом -n, и с ключом -N передаётся имя одной схемы, ключ -N выигрывает и схема исключается.

-O, --no-owner – Не генерировать команды, устанавливающие владение объектами, как в исходной базе данных. По умолчанию, pg_restore генерирует команды ALTER OWNER или SET SESSION AUTHORIZATION, восстанавливающие исходных владельцев создаваемых элементов схемы. Однако эти команды можно будет выполнить, только если к базе данных первоначально подключается суперпользователь (или пользователь, владеющими всеми объектами в скрипте). Чтобы получить скрипт, который сможет восстановить любой подключающийся пользователь (но при этом он станет владельцем всех созданных объектов), используется -O.

-P *имя-функции(тип-аргумента[, ...]),* --function=*имя-функции(тип-аргумента[, ...])* – Восстановить только указанную функцию. При этом важно записать имя функции и аргументы в точности так, как они фигурируют в оглавлении файла архива. Добавив дополнительные ключи -P, можно указать несколько функций.

-s, --schema-only – Восстановить только схему (определения данных), без данных, в объёме, в котором элементы схемы представлены в архиве.

Действие параметра противоположно действию --data-only. Это похоже на указание -– section=pre-data --section=post-data, но по историческим причинам не равнозначно ему.

-S *имя_пользователя,* --superuser=*имя_пользователя* – Задаёт имя суперпользователя, полномочия которого будут использоваться для отключения триггеров. Этот параметр применяется только с параметром --disable-triggers.

-t *таблица,* --table=*таблица* – Восстановить определение и/или данные только указанной таблицы. В этом контексте под «таблицей» подразумеваются также представления, материализованные представления, последовательности и сторонние таблицы. Чтобы выбрать несколько таблиц, ключ -t можно указать несколько раз. Этот параметр можно скомбинировать с -n, чтобы выбрать таблицу(ы) в определённой схеме.

-T *триггер,* --trigger=*триггер* – Восстановить только указанный триггер. Добавив дополнительные ключи -T, можно указать несколько триггеров.

-v, --verbose – Включает режим подробных сообщений.

-V, --version – Сообщить версию pg_restore и завершиться.

-x, --no-privileges, --no-acl – Не восстанавливать права доступа (не выполнять команды GRANT/REVOKE).

-1, --single-transaction – Произвести восстановление в одной транзакции (то есть, завернуть выполняемые команды в BEGIN/COMMIT). При этом гарантируется, что либо все команды будут выполнены успешно, либо не будет никаких изменений. Этот режим подразумевает --exit-on-error.

--disable-triggers – Этот параметр действует только при выгрузке одних данных. С ним pg_restore выполняет команды, отключающие триггеры в целевых таблицах на время восстановления данных.

Команды, генерируемые с --disable-triggers, должны выполнятся суперпользователем. Поэтому необходимо также задать имя суперпользователя в параметре – S или, что предпочтительнее, запускать pg_restore от имени суперпользователя PG360.

--enable-row-security – Этот параметр имеет смысл только при восстановлении содержимого таблицы, для которой включена защита строк. По умолчанию pg_restore устанавливает для row_security значение off для уверенности, что в таблице восстановлены все данные. Если у пользователя недостаточно прав для обхода защиты строк, выдаётся ошибка. Этот параметр указывает pg_restore установить в row_security значение on, чтобы пользователь мог попытаться восстановить содержимое таблицы с включённой защитой строк. Однако и при этом возможна ошибка, если пользователь не будет иметь права добавлять в эту таблицу выгруженные строки данных.

--if-exists – При удалении объектов базы использовать условные команды (то есть добавлять предложение IF EXISTS). Применяется только с параметром --clean.

--no-comments – Не выводить команды, восстанавливающие комментарии, даже если они содержатся в архиве.

--no-data-for-failed-tables – По умолчанию данные восстанавливаются даже при ошибке команды создания таблицы (например, когда она уже существует). С этим параметром данные в таком случае не восстанавливаются.

--no-publications – Не выводить команды, восстанавливающие публикации, даже если они содержатся в архиве.

--no-security-labels – Не выводить команды, восстанавливающие метки безопасности, даже если они содержатся в архиве.

--no-subscriptions – Не выводить команды, восстанавливающие подписки, даже если они содержатся в архиве.

--no-tablespaces – Не формировать команды для указания табличных пространств. При восстановлении все объекты будут создаваться в табличном пространстве по умолчанию.

--section=*имя секции* – Восстановить только указанный раздел. В качестве имени раздела можно задать pre-data, data или post-data.

--strict-names – Требует, чтобы каждому указанию схемы (-n/--schema) и таблицы (-t/--table) соответствовала минимум одна схема/таблица в файле резервной копии.

--use-set-session-authorization – Выводить команды SET SESSION AUTHORIZATION, соответствующие стандарту, вместо ALTER OWNER, для назначения владельцев объектов. В результате выгруженный скрипт будет более стандартизированным, но может не восстановиться корректно, в зависимости от истории объектов.

-?, --help – Показать справку по аргументам командной строки pg_restore и завершиться.

pg_restore также принимает в качестве параметров соединения следующие аргументы командной строки:

-h *сервер,* --host=*сервер* – Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения PGHOST, если она установлена. В противном случае выполняется подключение к Unix-сокету.

-p *порт,* --port=*порт* – Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения PGPORT, если она установлена, либо числом, заданным при компиляции.

-U *имя_пользователя,* --username=*имя_пользователя* – Имя пользователя, под которым производится подключение.

-w, --no-password – Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл .pgpass, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

-W, --password – Принудительно запрашивать пароль перед подключением к базе данных.

--role=*имя роли* – Задаёт имя роли, которая будет осуществлять восстановление. Получив это имя, pg_restore выполнит SET ROLE *имя_роли* после подключения к базе данных.

4.3. Входные и выходные данные программы pg_verifybackup для проверки целостности базовой копии кластера PG360:

-e , --exit-on-error – Завершиться при первой же выявленной проблеме. В отсутствие этого указания pg_verifybackup продолжает проверку копии после обнаружения первой ошибки и сообщает обо всех ошибках.

-i *путь,* --ignore=*путь* – Игнорировать указанный файл или каталог, который может быть задан относительным путём, при сравнении списка файлов данных, фактически присутствующих в

копии, со списком в файле backup_manifest. Если указан путь к каталогу, из рассмотрения исключается всё дерево подкаталогов, начиная с указанного. В случае совпадения относительного пути файла с указанным никакие сообщения о дополнительных или пропавших файлах, а также об изменении размера или несовпадении контрольных сумм файлов, выдаваться не будут. Этот параметр можно задать несколько раз.

-m *путь,* --manifest-path=*путь* – Использовать файл манифеста по заданному пути вместо файла, расположенного в корневом каталоге копии.

-n, --no-parse-wal – Не пытаться разобрать данные журнала упреждающей записи, которые могут понадобиться для восстановления проверяемой копии.

-q, --quiet – Не выводить ничего, если копия проходит проверку успешно.

-s, --skip-checksums – Не проверять контрольные суммы файлов данных. При этом тем не менее будет проверяться отсутствие или наличие файлов и их размеры. В таком режиме проверка выполняется гораздо быстрее, так как собственно содержимое файлов читать не требуется.

-w *путь,* --wal-directory=*путь* – Проверять файлы WAL, находящиеся в указанном каталоге, а не в pg_wal.

-V , --version – Сообщить версию pg_verifybackup и завершиться.

-?, --help – Вывести справку об аргументах командной строки pg_verifybackup и завершиться.

4.3. Входные и выходные данные программы psql – интерактивного терминала PG360:

-a , --echo-all – Отправляет в стандартный вывод все непустые входные строки по мере их чтения. (Это не относится к строкам, считанным в интерактивном режиме.) Эквивалентно установке переменной ECHO в значение all.

-A, --no-align – Переключает на невыровненный режим вывода. (По умолчанию используется другой режим, aligned.) Равнозначно команде \pset format unaligned.

-b, --echo-errors – Выводит все команды SQL с ошибками в стандартный канал ошибок. Равнозначно присваиванию переменной ECHO значения errors.

-c *команда,* --command=*команда* – Передаёт psql *команду* для выполнения. Этот ключ можно повторять и комбинировать в любом порядке с ключом -f. Когда указывается -c или -f, psql не читает команды со стандартного ввода; вместо этого она завершается сразу после обработки всех ключей -c и -f по порядку.

Заданная *команда* должна быть либо командной строкой, которая полностью интерпретируется сервером (т. е. не использует специфические функции psql), либо одиночной командой с обратной косой чертой. Таким образом, используя -c, нельзя смешивать метакоманды SQL и psql. Но это можно сделать, передав несколько ключей -c или передав строку в psql через канал:

```
psql -c '\x' -c 'SELECT * FROM foo;'
```

или

```
echo '\x \\ SELECT * FROM foo;' | psql
```

(\\ – разделитель метакоманд.)

Каждая строка SQL-команд, заданная ключом -c, передаётся на сервер как один запрос. Поэтому сервер выполняет её в одной транзакции, даже когда эта строка содержит несколько команд SQL, если только в ней не содержатся явные команды BEGIN/COMMIT, разделяющие её

на несколько транзакций. Кроме того, psql печатает результат только последней SQL-команды в строке.

--csv – Переключает в режим вывода CSV (Comma Separated Values, Значения, разделённые запятыми). Равнозначно команде \pset format csv.

-d *имя_бд,* --dbname=*имя_бд* – Указывает имя базы данных для подключения. Равнозначно указанию *имя_бд* в первом аргументе, не являющемся ключом, в командной строке. Вместо имени может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

-e, --echo-queries – Посылает все команды SQL, отправленные на сервер, ещё и на стандартный вывод. Эквивалентно установке переменной ECHO в значение queries.

-E, --echo-hidden – Отображает фактические запросы, генерируемые \d и другими командами, начинающимися с \. Это можно использовать для изучения внутренних операций в psql. Эквивалентно установке переменной ECHO_HIDDEN значения on.

-f *имя_файла,* --file=*имя_файла* – Читает команды из файла *имя_файла*, а не из стандартного ввода. Этот ключ можно повторять и комбинировать в любом порядке с ключом -c. Если указан ключ -c или -f, программа psql не читает команды со стандартного ввода; вместо этого она завершается после обработки всех ключей -c и -f по очереди. Не считая этого, данный ключ по большому счёту равнозначен метакоманде \i.

Если *имя_файла* задано символом – (минус), считывается стандартный ввод до признака конца файла или до метакоманды \q. Это позволяет перемежать интерактивный ввод с вводом из файлов.

-F *разделитель,* --field-separator=*разделитель* – Использование *разделитель* в качестве разделителя полей при невыровненном режиме вывода. Эквивалентно \pset fieldsep или \f.

-h *компьютер,* --host=*компьютер* – Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

-H, --html – Переключает в режим вывода HTML. Равнозначно команде \pset format html или \H.

-l, --list – Выводит список всех доступных баз данных и завершает работу. Другие параметры, не связанные с соединением, игнорируются. Это похоже на метакоманду \list.

Когда используется этот аргумент, psql будет подключаться к базе данных postgres, если только в командной строке не задана другая база данных (в параметре -d или не через параметры, а, например, через запись службы, но не через переменную окружения).

-L *имя_файла,* --log-file=*имя_файла* – В дополнение к обычному выводу, записывает вывод результатов всех запросов в файл *имя_файла*.

-n, --no-readline – Отключает использование Readline для редактирования командной строки и использования истории команд.

-o *имя_файла,* --output=*имя_файла* – Записывает вывод результатов всех запросов в файл *имя_файла*. Эквивалентно команде \o.

-p *порт,* --port=*порт* – Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной среды PGPORT, если она установлена, либо числом, заданным при компиляции, обычно 5432.

-P *присваивание,* --pset=*присваивание* – Задаёт параметры печати, в стиле команды \pset. Имя параметра и значение разделяются знаком равенства, а не пробела.

-q, --quiet – Указывает, что psql должен работать без вывода дополнительных сообщений. По умолчанию, выводятся приветствия и различные информационные сообщения. Этого не произойдёт с использованием данного параметра. Полезно вместе с параметром -c. Этот же эффект можно получить, установив для переменной QUIET значение on.

-R *разделитель,* --record-separator=*разделитель* – Использовать *разделитель* как разделитель записей при невыровненном режиме вывода. Равнозначно команде \pset recordsep.

-s, --single-step – Запуск в пошаговом режиме. Это означает, что пользователь будет подтверждать выполнение каждой команды, отправляемой на сервер, с возможностью отменить выполнение. Используется для отладки скриптов.

-S, --single-line – Запуск в однострочном режиме, при котором символ новой строки завершает SQL-команды, так же как это делает точка с запятой.

-t, --tuples-only – Отключает вывод имён столбцов и результирующей строки с количеством выбранных записей. Равнозначно команде \t или \pset tuples_only.

-T *параметры_таблицы,* --table-attr=*параметры_таблицы* – Задаёт атрибуты, которые будут вставлены в тег HTML table.

-U *имя_пользователя,* --username=*имя_пользователя* – Использовать для подключения к базе данных *имя_пользователя* вместо подразумеваемого по умолчанию.

-v *присваивание,* --set=*присваивание,* --variable=*присваивание* – Выполняет присваивание значения переменной, как метакоманда \set.

-V, --version – Выводит версию psql и завершает работу.

-w, --no-password – Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль нельзя получить из других источников, например из файла .pgpass, попытка соединения не удастся.

-W, --password – Принудительно запрашивать пароль перед подключением к базе данных, даже если он не будет использоваться.

Если сервер требует аутентификацию по паролю и пароль нельзя получить из других источников, например из файла .pgpass, psql запросит пароль в любом случае. Однако чтобы понять, что требуется пароль, psql лишний раз подключится к серверу. Поэтому иногда имеет смысл ввести -W, чтобы исключить эту ненужную попытку подключения.

-x, --expanded – Включает режим развёрнутого вывода таблицы.

-X, --no-psqlrc – Не читать стартовые файлы (ни общесистемный файл psqlrc, ни пользовательский файл ~/.psqlrc).

-z, --field-separator-zero – Установить нулевой байт в качестве разделителя полей для невыровненного режима вывода. Равнозначно команде \pset fieldsep_zero.

-0, --record-separator-zero – Установить нулевой байт в качестве разделителя записей для невыровненного режима вывода.

-1, --single-transaction – Этот параметр может применяться только в сочетании с одним или несколькими параметрами -c и/или -f. С ним psql выполняет команду BEGIN перед обработкой первого такого параметра и COMMIT после последнего, заворачивая таким образом все команды в

одну транзакцию. Это гарантирует, что либо все команды завершатся успешно, либо никакие изменения не сохранятся.

Если в самих этих командах содержатся операторы BEGIN, COMMIT или ROLLBACK, этот параметр не даст желаемого эффекта. Кроме того, если какая-либо отдельная команда не может выполняться внутри блока транзакции, с этим параметром вся транзакция прервётся с ошибкой.

-?, --help[=*тема*] – Показать справку по psql и завершиться. Необязательный параметр *тема* (по умолчанию options) выбирает описание интересующей части psql: commands описывает команды psql с обратной косой чертой; options описывает параметры командной строки, которые можно передать psql; a variables выдаёт справку по переменным конфигурации psql.

Код завершения

При нормальном завершении psql возвращает 0 в командную оболочку ОС, 1 – если произошла фатальная ошибка в самом psql (например, нехватка памяти, файл не найден), 2 – при неудачном соединении с сервером неинтерактивного сеанса, 3 – при ошибке в скрипте и установленной переменной ON_ERROR_STOP.

4.3. Входные и выходные данные программы reindexdb для перестроения индексов в базе данных PG360:

-a, --all – Переиндексировать все базы данных.

--concurrently – Использовать режим CONCURRENTLY.

[-d] *имя_бд,* [--dbname=]*имя_бд* – Указывает имя базы данных для переиндексации, когда не используется параметр -a/--all. Если это указание отсутствует, имя базы определяется переменной окружения PGDATABASE. Если эта переменная не установлена, именем базы будет имя пользователя, указанное для подключения. В аргументе *имя_бд* может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

-e, --echo – Выводить команды, которые reindexdb генерирует и передаёт серверу.

-i *индекс,* --index=*индекс* – Пересоздать только указанный *индекс*. Добавив дополнительные ключи -i, можно пересоздать несколько индексов.

-j *число_заданий,* --jobs=*число_заданий* – Выполнять команды переиндексации в параллельном режиме, запуская их одновременно в количестве *число_заданий*. Это может сократить время обработки, но при этом увеличить нагрузку на сервер.

reindexdb будет устанавливать несколько подключений к базе данных (в количестве *число_заданий*), необходимо убедиться в том, что значение max_connections достаточно велико, чтобы все эти подключения были приняты.

Этот параметр несовместим с параметрами --index и --system.

-q, --quiet – Подавлять вывод сообщений о прогрессе выполнения.

-s, --system – Переиндексировать только системные каталоги базы данных.

-S *схема,* --schema=*схема* – Переиндексировать только указанную *схему*. Переиндексировать несколько схем можно, добавив несколько ключей -S.

-t *таблица,* --table=*таблица* – Переиндексировать только указанную *таблицу*. Переиндексировать несколько таблиц можно, добавив несколько ключей -t.

-v, --verbose – Вывести подробную информацию во время процесса.

-V, --version – Сообщить версию reindexdb и завершиться.

-?, --help – Показать справку по аргументам командной строки reindexdb и завершиться.

Программа reindexdb также принимает следующие аргументы командной строки в качестве параметров подключения:

-h *сервер,* --host=*сервер* – Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

-p *порт,* --port=*порт* – Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

-U *имя_пользователя,* --username=*имя_пользователя* – Имя пользователя, под которым производится подключение.

-w, --no-password – Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл .pgpass, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

-W, --password – Принудительно запрашивать пароль перед подключением к базе данных.

--maintenance-db=*имя_бд* – Указывает имя базы данных, к которой будет выполняться подключение для определения подлежащих переиндексации баз данных, когда используется ключ -a/--all. Если это имя не указано, будет выбрана база postgres, а если она не существует – template1. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке. Кроме того, все параметры в строке подключения, за исключением имени базы, будут использоваться и при подключении к другим базам данных.

4.3. Входные и выходные данные программы vacuumdb для выполнения очистки и анализа базы данных PG360:

-a, --all – Очистить все базы данных.

[-d] *имя_бд,* [--dbname=]*имя_бд* – Указывает имя базы данных для очистки или анализа, когда не используется параметр -a/--all. Если это указание отсутствует, имя базы определяется переменной окружения PGDATABASE. Если эта переменная не задана, именем базы будет имя пользователя, указанное для подключения. В аргументе *имя_бд* может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

--disable-page-skipping – Запретить пропуск страниц в зависимости от содержимого карты видимости.

-e, --echo – Выводить команды, которые vacuumdb генерирует и передаёт серверу.

-f, --full – Произвести «полную» очистку.

-F, --freeze – Агрессивно «замораживать» версии строк.

-j *число_заданий,* --jobs=*число_заданий* – Выполнять команды очистки и анализа в параллельном режиме, запуская их одновременно в количестве *число_заданий*. Это может сократить время обработки, но при этом увеличить нагрузку на сервер.

vacuumdb будет устанавливать несколько подключений к базе данных (в количестве *число_заданий*), так что необходимо убедиться в том, что значение max_connections достаточно велико, чтобы все эти подключения были приняты.

Использование этого режима с параметром -f (FULL) может привести к отказам из-за взаимоблокировок, если параллельно начнут обрабатываться определённые системные каталоги.

--min-mxid-age *возраст_мультитранзакции* – Выполнять команды очистки и анализа только для таблиц, имеющих не менее чем заданный *возраст_мультитранзакции*.

--min-xid-age *возраст_транзакции* – Выполнять команды очистки и анализа только для таблиц, имеющих не менее чем заданный *возраст_транзакции*.

-P *параллельные_исполнители,*

--parallel=*параллельные_исполнители* – Задаёт количество параллельных исполнителей для *параллельной очистки*. Это позволяет в ходе очистки задействовать мощности нескольких процессоров для обработки индексов.

-q, --quiet – Подавлять вывод сообщений о прогрессе выполнения.

--skip-locked – Пропускать отношения, которые не удаётся немедленно заблокировать для обработки.

-t *таблица* [(*столбец* [,...])], --table=*таблица* [(*столбец* [,...])] – Производить очистку или анализ только указанной *таблицы*. Имена столбцов можно указать только в сочетании с параметрами --analyze и --analyze-only. Добавив дополнительные ключи -t, можно обработать несколько таблиц.

-v, --verbose – Вывести подробную информацию во время процесса.

-V, --version – Сообщить версию vacuumdb и завершиться.

-z, --analyze – Также вычислить статистику для оптимизатора.

-Z, --analyze-only – Только вычислить статистику для оптимизатора (не производить очистку).

--analyze-in-stages – Только вычислить статистику для оптимизатора (без очистки), подобно --analyze-only. Но для скорейшего получения статистики, выполнить анализ в несколько проходов (в настоящее время, три) с разными параметрами.

-?, --help – Показать справку по аргументам командной строки vacuumdb и завершиться.

Программа vacuumdb также принимает следующие аргументы командной строки в качестве параметров подключения:

-h *сервер,* --host=*сервер* – Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

-p *порт,* --port=*порт* – Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

-U *имя_пользователя,* --username=*имя_пользователя* – Имя пользователя, под которым производится подключение.

-w, --no-password – Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл .pgpass, попытка соединения не удастся.

-W, --password – Принудительно запрашивать пароль перед подключением к базе данных.

--maintenance-db=*имя_бд* – Указывает имя базы данных, к которой будет выполняться подключение для определения подлежащих очистке баз данных, когда используется ключ -a/--all. Если это имя не указано, будет выбрана база postgres, а если она не существует – template1. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке

подключения переопределяют одноимённые параметры, заданные в командной строке. Кроме того, все параметры в строке подключения, за исключением имени базы, будут использоваться и при подключении к другим базам данных.

5. СООБЩЕНИЯ

5.1. Во время сеанса работы сервера PG360 могут выдаваться различные сообщения, которым назначены пятисимвольные коды ошибок, соответствующие кодам «SQLSTATE», описанным в стандарте SQL. Согласно стандарту, первые два символа кода ошибки обозначают класс ошибок, а последние три символа обозначают определённое условие в этом классе.

В Таблице 5.1 перечислены все коды ошибок и классы ошибок. Для каждого класса ошибок имеется код ошибки с последними тремя символами 000 – это ошибки, которые относятся к некоторому классу, но не имеют более определённого кода.

Символ, указанный в столбце «Имя условия», определяет условие в PL/pgSQL. Имена условий могут записываться в верхнем или нижнем регистре.

Таблица 5.1. Коды и классы ошибок PG360

Код ошибки	**Имя условия**
Класс 00 — Успешное завершение	
00000	successful_completion
Класс 01 — Предупреждение	
01000	warning
0100C	dynamic_result_sets_returned
01008	implicit_zero_bit_padding
01003	null_value_eliminated_in_set_function
01007	privilege_not_granted
01006	privilege_not_revoked
01004	string_data_right_truncation
01P01	deprecated_feature
Класс 02 — Нет данных (это также класс предупреждений согласно стандарту SQL)	
02000	no_data
02001	no_additional_dynamic_result_sets_returned
Класс 03 — SQL-оператор ещё не завершён	
03000	sql_statement_not_yet_complete
Класс 08 — Исключение, связанное с подключением	
08000	connection_exception
08003	connection_does_not_exist
08006	connection_failure
08001	sqlclient_unable_to_establish_sqlconnection
08004	sqlserver_rejected_establishment_of_sqlconnection
08007	transaction_resolution_unknown
08P01	protocol_violation
Класс 09 — Исключение с действием триггера	
09000	triggered_action_exception
Класс 0A — Неподдерживаемая функциональность	

Код ошибки	Имя условия
0A000	feature_not_supported
Класс 0B — Неверное начало транзакции	
0B000	invalid_transaction_initiation
Класс 0F — Исключение с указателем на данные	
0F000	locator_exception
0F001	invalid_locator_specification
Класс 0L — Неверный праводатель	
0L000	invalid_grantor
0LP01	invalid_grant_operation
Класс 0P — Неверное указание роли	
0P000	invalid_role_specification
Класс 0Z — Исключение диагностики	
0Z000	diagnostics_exception
0Z002	stacked_diagnostics_accessed_without_active_handler
Класс 20 — Case не найден	
20000	case_not_found
Класс 21 — Нарушение количества	
21000	cardinality_violation
Класс 22 — Исключение в данных	
22000	data_exception
2202E	array_subscript_error
22021	character_not_in_repertoire
22008	datetime_field_overflow
22012	division_by_zero
22005	error_in_assignment
2200B	escape_character_conflict
22022	indicator_overflow
22015	interval_field_overflow
2201E	invalid_argument_for_logarithm
22014	invalid_argument_for_ntile_function
22016	invalid_argument_for_nth_value_function
2201F	invalid_argument_for_power_function
2201G	invalid_argument_for_width_bucket_function
22018	invalid_character_value_for_cast
22007	invalid_datetime_format
22019	invalid_escape_character
2200D	invalid_escape_octet
22025	invalid_escape_sequence
22P06	nonstandard_use_of_escape_character
22010	invalid_indicator_parameter_value
22023	invalid_parameter_value
22013	invalid_preceding_or_following_size

Код ошибки	Имя условия
2201B	invalid_regular_expression
2201W	invalid_row_count_in_limit_clause
2201X	invalid_row_count_in_result_offset_clause
2202H	invalid_tablesample_argument
2202G	invalid_tablesample_repeat
22009	invalid_time_zone_displacement_value
2200C	invalid_use_of_escape_character
2200G	most_specific_type_mismatch
22004	null_value_not_allowed
22002	null_value_no_indicator_parameter
22003	numeric_value_out_of_range
2200H	sequence_generator_limit_exceeded
22026	string_data_length_mismatch
22001	string_data_right_truncation
22011	substring_error
22027	trim_error
22024	unterminated_c_string
2200F	zero_length_character_string
22P01	floating_point_exception
22P02	invalid_text_representation
22P03	invalid_binary_representation
22P04	bad_copy_file_format
22P05	untranslatable_character
2200L	not_an_xml_document
2200M	invalid_xml_document
2200N	invalid_xml_content
2200S	invalid_xml_comment
2200T	invalid_xml_processing_instruction
22030	duplicate_json_object_key_value
22031	invalid_argument_for_sql_json_datetime_function
22032	invalid_json_text
22033	invalid_sql_json_subscript
22034	more_than_one_sql_json_item
22035	no_sql_json_item
22036	non_numeric_sql_json_item
22037	non_unique_keys_in_a_json_object
22038	singleton_sql_json_item_required
22039	sql_json_array_not_found
2203A	sql_json_member_not_found
2203B	sql_json_number_not_found
2203C	sql_json_object_not_found
2203D	too_many_json_array_elements

Код ошибки	Имя условия
2203E	too_many_json_object_members
2203F	sql_json_scalar_required
Класс 23 — Нарушение ограничения целостности	
23000	integrity_constraint_violation
23001	restrict_violation
23502	not_null_violation
23503	foreign_key_violation
23505	unique_violation
23514	check_violation
23P01	exclusion_violation
Класс 24 — Неверное состояние курсора	
24000	invalid_cursor_state
Класс 25 — Неверное состояние транзакции	
25000	invalid_transaction_state
25001	active_sql_transaction
25002	branch_transaction_already_active
25008	held_cursor_requires_same_isolation_level
25003	inappropriate_access_mode_for_branch_transaction
25004	inappropriate_isolation_level_for_branch_transaction
25005	no_active_sql_transaction_for_branch_transaction
25006	read_only_sql_transaction
25007	schema_and_data_statement_mixing_not_supported
25P01	no_active_sql_transaction
25P02	in_failed_sql_transaction
25P03	idle_in_transaction_session_timeout
Класс 26 — Неверное имя SQL-оператора	
26000	invalid_sql_statement_name
Класс 27 — Нарушение при изменении данных в триггере	
27000	triggered_data_change_violation
Класс 28 — Неверное указание авторизации	
28000	invalid_authorization_specification
28P01	invalid_password
Класс 2B — Зависимые описания привилегий всё ещё существуют	
2B000	dependent_privilege_descriptors_still_exist
2BP01	dependent_objects_still_exist
Класс 2D — Неверное завершение транзакции	
2D000	invalid_transaction_termination
Класс 2F — Исключение в подпрограмме SQL	
2F000	sql_routine_exception
2F005	function_executed_no_return_statement
2F002	modifying_sql_data_not_permitted
2F003	prohibited_sql_statement_attempted

Код ошибки	Имя условия
2F004	reading_sql_data_not_permitted
Класс 34 — Неверное имя курсора	
34000	invalid_cursor_name
Класс 38 — Исключение во внешней подпрограмме	
38000	external_routine_exception
38001	containing_sql_not_permitted
38002	modifying_sql_data_not_permitted
38003	prohibited_sql_statement_attempted
38004	reading_sql_data_not_permitted
Класс 39 — Исключение при вызове внешней подпрограммы	
39000	external_routine_invocation_exception
39001	invalid_sqlstate_returned
39004	null_value_not_allowed
39P01	trigger_protocol_violated
39P02	srf_protocol_violated
39P03	event_trigger_protocol_violated
Класс 3B — Исключение точки сохранения	
3B000	savepoint_exception
3B001	invalid_savepoint_specification
Класс 3D — Неверное имя каталога	
3D000	invalid_catalog_name
Класс 3F — Неверное имя схемы	
3F000	invalid_schema_name
Класс 40 — Откат транзакции	
40000	transaction_rollback
40002	transaction_integrity_constraint_violation
40001	serialization_failure
40003	statement_completion_unknown
40P01	deadlock_detected
Класс 42 — Ошибка синтаксиса или нарушение правила доступа	
42000	syntax_error_or_access_rule_violation
42601	syntax_error
42501	insufficient_privilege
42846	cannot_coerce
42803	grouping_error
42P20	windowing_error
42P19	invalid_recursion
42830	invalid_foreign_key
42602	invalid_name
42622	name_too_long
42939	reserved_name
42804	datatype_mismatch

Код ошибки	Имя условия
42P18	indeterminate_datatype
42P21	collation_mismatch
42P22	indeterminate_collation
42809	wrong_object_type
428C9	generated_always
42703	undefined_column
42883	undefined_function
42P01	undefined_table
42P02	undefined_parameter
42704	undefined_object
42701	duplicate_column
42P03	duplicate_cursor
42P04	duplicate_database
42723	duplicate_function
42P05	duplicate_prepared_statement
42P06	duplicate_schema
42P07	duplicate_table
42712	duplicate_alias
42710	duplicate_object
42702	ambiguous_column
42725	ambiguous_function
42P08	ambiguous_parameter
42P09	ambiguous_alias
42P10	invalid_column_reference
42611	invalid_column_definition
42P11	invalid_cursor_definition
42P12	invalid_database_definition
42P13	invalid_function_definition
42P14	invalid_prepared_statement_definition
42P15	invalid_schema_definition
42P16	invalid_table_definition
42P17	invalid_object_definition
Класс 44 — Нарушение WITH CHECK OPTION	
44000	with_check_option_violation
Класс 53 — Нехватка ресурсов	
53000	insufficient_resources
53100	disk_full
53200	out_of_memory
53300	too_many_connections
53400	configuration_limit_exceeded
Класс 54 — Превышение ограничения программы	
54000	program_limit_exceeded

Код ошибки	Имя условия
54001	statement_too_complex
54011	too_many_columns
54023	too_many_arguments
Класс 55 — Объект не в требуемом состоянии	
55000	object_not_in_prerequisite_state
55006	object_in_use
55P02	cant_change_runtime_param
55P03	lock_not_available
55P04	unsafe_new_enum_value_usage
Класс 57 — Вмешательство оператора	
57000	operator_intervention
57014	query_canceled
57P01	admin_shutdown
57P02	crash_shutdown
57P03	cannot_connect_now
57P04	database_dropped
Класс 58 — Ошибка системы (ошибка, внешняя по отношению к PG360)	
58000	system_error
58030	io_error
58P01	undefined_file
58P02	duplicate_file
Класс 72 — Ошибка снимка	
72000	snapshot_too_old
Класс F0 — Ошибка файла конфигурации	
F0000	config_file_error
F0001	lock_file_exists
Класс HV — Ошибка обёртки сторонних данных (SQL/MED)	
HV000	fdw_error
HV005	fdw_column_name_not_found
HV002	fdw_dynamic_parameter_value_needed
HV010	fdw_function_sequence_error
HV021	fdw_inconsistent_descriptor_information
HV024	fdw_invalid_attribute_value
HV007	fdw_invalid_column_name
HV008	fdw_invalid_column_number
HV004	fdw_invalid_data_type
HV006	fdw_invalid_data_type_descriptors
HV091	fdw_invalid_descriptor_field_identifier
HV00B	fdw_invalid_handle
HV00C	fdw_invalid_option_index
HV00D	fdw_invalid_option_name

Код ошибки	Имя условия
HV090	fdw_invalid_string_length_or_buffer_length
HV00A	fdw_invalid_string_format
HV009	fdw_invalid_use_of_null_pointer
HV014	fdw_too_many_handles
HV001	fdw_out_of_memory
HV00P	fdw_no_schemas
HV00J	fdw_option_name_not_found
HV00K	fdw_reply_handle
HV00Q	fdw_schema_not_found
HV00R	fdw_table_not_found
HV00L	fdw_unable_to_create_execution
HV00M	fdw_unable_to_create_reply
HV00N	fdw_unable_to_establish_connection
Класс P0 — Ошибка PL/pgSQL	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows
P0004	assert_failure
Класс XX — Внутренняя ошибка	
XX000	internal_error
XX001	data_corrupted
XX002	index_corrupted

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ

В настоящем документе приняты следующие сокращения:

БД	–	база данных;
ОС	–	операционная система;
ПК	–	персональный компьютер;
СУБД	–	система управления базами данных.

ПРИЛОЖЕНИЕ А

Перечень информационных схем PG360

A.1. information_schema_catalog_name

Таблица information_schema_catalog_name содержит одну строку и один столбец с именем текущей базы данных.

Таблица A.1. Столбцы information_schema_catalog_name

Тип столбца	**Описание**
catalog_name sql_identifier	Имя базы данных, содержащей эту информационную схему

A.2. administrable_role_authorizations

Представление administrable_role_authorizations описывает все роли, для которых текущий пользователь является администратором.

Таблица A.2. Столбцы administrable_role_authorizations

Тип столбца	**Описание**
grantee sql_identifier	Имя роли, которой было разрешено участие в целевой роли (может быть текущий пользователь, либо другая роль, в случае вложенного членства)
role_name sql_identifier	Имя целевой роли
is_grantable yes_or_no	Всегда YES

A.3. applicable_roles

Представление applicable_roles описывает все роли, права которых может использовать текущий пользователь. Это означает, что существует некоторая цепочка ролей от текущего пользователя к целевой роли. Роль самого пользователя также считается применимой. Набор применимых ролей обычно используется для проверки разрешений.

Таблица A.3. Столбцы applicable_roles

Тип столбца	**Описание**
grantee sql_identifier	Имя роли, которой было разрешено участие в целевой роли (может быть текущий пользователь, либо другая роль, в случае вложенного членства)
role_name sql_identifier	Имя целевой роли
is_grantable yes_or_no	YES, если субъект является администратором для этой роли, или NO в противном случае

A.4. attributes

Представление attributes содержит информацию об атрибутах составных типов данных, определённых в базе. В нём показываются только те атрибуты, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторое право для использования типа).

Таблица A.4. Столбцы attributes

Тип столбца	**Описание**
udt_catalog sql_identifier	Имя базы данных, содержащей тип данных (всегда текущая база)
udt_schema sql_identifier	Имя схемы, содержащей тип данных
udt_name sql_identifier	Имя типа данных
attribute_name sql_identifier	Имя атрибута
ordinal_position cardinal_number	Порядковый номер атрибута внутри типа данных (нумерация начинается с 1)
attribute_default character_data	Выражение по умолчанию для атрибута
is_nullable yes_or_no	YES, если атрибут может содержать NULL, или NO, если он не принимает NULL
data_type character_data	Тип данных атрибута, если это встроенный тип, либо ARRAY, если это массив, иначе — USER-DEFINED
character_maximum_length cardinal_number	Если в data_type указан тип текстовой или битовой строки, это поле задаёт её объявленную максимальную длину; NULL для всех других типов данных, либо если максимальная длина не объявлена
character_octet_length cardinal_number	Если в data_type указан тип символьной строки, это поле задаёт её максимально возможный размер в октетах (байтах); NULL для всех других типов данных. Максимальный размер в октетах зависит от объявленной максимальной длины в символах (см. выше) и от кодировки сервера.
collation_catalog sql_identifier	Имя базы данных, содержащей правило сортировки атрибута (это всегда текущая база), либо NULL, если это правило по умолчанию или тип данных атрибута несортируемый
collation_schema sql_identifier	Имя схемы, содержащей правило сортировки атрибута, либо NULL, если это правило по умолчанию или тип данных атрибута несортируемый
collation_name sql_identifier	Имя правила сортировки атрибута, либо NULL, если это правило по умолчанию или атрибут несортируемый
numeric_precision cardinal_number	Если в data_type указан числовой тип, этот столбец содержит точность (объявленную или неявную) типа для этого атрибута. Точность определяет число значащих цифр. Она может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу numeric_precision_radix . Для всех других типов данных этот

Тип столбца	Описание
	столбец содержит NULL.
numeric_precision_radix cardinal_number	Если в data_type указан числовой тип, в этом столбце определяется, по какому основанию задаются значения в столбцах numeric_precision и numeric_scale . Возможные варианты: 2 или 10. Для всех других типов данных этот столбец содержит NULL.
numeric_scale cardinal_number	Если в data_type указан точный числовой тип, этот столбец содержит масштаб (объявленный или неявный) типа для этого атрибута. Масштаб определяет число значащих цифр справа от десятичной точки. Он может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу numeric_ precision_radix . Для всех других типов данных этот столбец содержит NULL.
datetime_precision cardinal_number	Если в data_type указан тип даты, времени, отметки времени или интервала, этот столбец содержит точность (объявленную или неявную) в долях секунды типа этого атрибута, то есть число десятичных цифр, сохраняемых после десятичной точки в значении секунд. Для всех других типов данных этот столбец содержит NULL.
interval_type character_data	Если в data_type указан тип интервала, этот столбец определяет, какие поля принимает интервал в этом атрибуте, например: YEAR TO MONTH, DAY TO SECOND и т. д. Если ограничения для полей не заданы (то есть, интервал принимает все поля), и для любых других типов данных это поле содержит NULL.
attribute_udt_catalog sql_identifier	Имя базы данных, в которой определён тип данных атрибута (всегда текущая база)
attribute_udt_schema sql_identifier	Имя схемы, в которой определён тип данных атрибута
attribute_udt_name sql_identifier	Имя типа данных атрибута
maximum_cardinality cardinal_number	Всегда NULL, так как массивы имеют неограниченную максимальную ёмкость в PG360
dtd_identifier sql_identifier	Идентификатор дескриптора типа данных столбца, уникальный среди всех дескрипторов типов, относящихся к таблице. Он в основном полезен для соединения с другими экземплярами таких идентификаторов. (Конкретный формат идентификатора не определён и не гарантируется, что он останется неизменным в будущих версиях.)

A.5. character_sets

Представление character_sets описывает наборы символов, доступные в текущей базе данных.

Таблица A.5. Столбцы character_sets

Тип столбца	**Описание**
character_set_catalog sql_identifier	Наборы символов в настоящее время не представлены в виде объектов схемы, так что этот столбец содержит NULL.
character_set_schema sql_identifier	Наборы символов в настоящее время не представлены в виде объектов схемы, так что этот столбец содержит NULL.
character_set_name sql_identifier	Имя набора символов, в настоящее время в качестве этого имени показывается имя кодировки базы данных
character_repertoire sql_identifier	Совокупность символов — UCS для кодировки UTF8, либо просто имя кодировки
form_of_use sql_identifier	Форма кодировки символов, то же, что и кодировка базы данных
default_collate_catalog sql_identifier	Имя базы данных, содержащей правило сортировки по умолчанию (всегда текущая база, если это правило установлено)
default_collate_schema sql_identifier	Имя схемы, содержащей правило сортировки по умолчанию
default_collate_name sql_identifier	Имя правила сортировки по умолчанию. Правилом сортировки по умолчанию считается правило, соответствующее параметрам COLLATE и CTYPE текущей базы данных. Если такого правила нет, данный столбец и связанные столбцы схемы и каталога содержат NULL.

A.6 check_constraint_routine_usage

Представление check_constraint_routine_usage описывает подпрограммы (функции и процедуры), участвующие в проверках ограничений. В нём показываются только те подпрограммы, которые принадлежат текущей активной роли.

Таблица A.6. Столбцы check_constraint_routine_usage

Тип столбца	**Описание**
constraint_catalog sql_identifier	Имя базы данных, содержащей ограничение (всегда текущая база)
constraint_schema sql_identifier	Имя схемы, содержащей ограничение
constraint_name sql_identifier	Имя ограничения
specific_catalog sql_identifier	Имя базы данных, содержащей функцию (всегда текущая база)
specific_schema sql_identifier	Имя схемы, содержащей функцию
specific_name sql_identifier	«Однозначное имя» функции

A.7. check_constraints

Представление check_constraints показывает все ограничения-проверки, либо определённые для таблицы или домена, либо принадлежащие текущей активной роли. (Владелец таблицы или домена является владельцем ограничения.)

Таблица А.7. Столбцы check_constraints

Тип столбца	**Описание**
constraint_catalog sql_identifier	Имя базы данных, содержащей ограничение (всегда текущая база)
constraint_schema sql_identifier	Имя схемы, содержащей ограничение
constraint_name sql_identifier	Имя ограничения
check_clause character_data	Выражение проверки для ограничения

A.8. collations

Представление collations показывает правила сортировки, доступные в текущей базе данных.

Таблица А.8. Столбцы collations

Тип столбца	**Описание**
collation_catalog sql_identifier	Имя базы данных, содержащей правило сортировки (всегда текущая база)
collation_schema sql_identifier	Имя схемы, содержащей правило сортировки
collation_name sql_identifier	Имя правила сортировки
pad_attribute character_data	Всегда NO PAD

A.9. collation_character_set_applicability

Представление collation_character_set_applicability показывает, к каким наборам символов применимы доступные правила сортировки.

Таблица А.9. Столбцы collation_character_set_applicability

Тип столбца	**Описание**
collation_catalog sql_identifier	Имя базы данных, содержащей правило сортировки (всегда текущая база)
collation_schema sql_identifier	Имя схемы, содержащей правило сортировки
collation_name sql_identifier	Имя правила сортировки
character_set_catalog sql_identifier	Наборы символов в настоящее время не представлены в виде объектов схемы, так что этот столбец содержит NULL
character_set_schema sql_identifier	Наборы символов в настоящее время не представлены в виде объектов схемы, так что этот столбец содержит NULL
character_set_name sql_identifier	Имя набора символов

A.10.column_column_usage

Представление column_column_usage описывает все генерируемые столбцы, которые зависят от других базовых столбцов в той же таблице. В нём показываются только таблицы, принадлежащие текущей активной роли.

Таблица A.10. Столбцы column_column_usage

Тип столбца	**Описание**
table_catalog sql_identifier	Имя базы данных, содержащей таблицу (всегда текущая база)
table_schema sql_identifier	Имя схемы, содержащей таблицу
table_name sql_identifier	Имя таблицы
column_name sql_identifier	Имя базового столбца, от которого зависит генерируемый
dependent_column sql_identifier	Имя генерируемого столбца

A.11. column_domain_usage

Представление column_domain_usage описывает все столбцы (таблиц или представлений), которые используют какие-либо домены, определённые в базе данных и принадлежащие текущей активной роли.

Таблица A.11. Столбцы column_domain_usage

Тип столбца	**Описание**
domain_catalog sql_identifier	Имя базы данных, содержащей домен (всегда текущая база)
domain_schema sql_identifier	Имя схемы, содержащей домен
domain_name sql_identifier	Имя домена
table_catalog sql_identifier	Имя базы данных, содержащей таблицу (всегда текущая база)
table_schema sql_identifier	Имя схемы, содержащей таблицу
table_name sql_identifier	Имя таблицы
column_name sql_identifier	Имя столбца

A.12. column_options

Представление column_options показывает все параметры, определённые для столбцов сторонней таблицы в текущей базе данных. В нём отражаются только те столбцы сторонних таблиц, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица A.12. Столбцы column_options

Тип столбца	**Описание**
table_catalog sql_identifier	Имя базы данных, содержащей стороннюю таблицу (всегда текущая база)
table_schema sql_identifier	Имя схемы, содержащей стороннюю таблицу
table_name sql_identifier	Имя сторонней таблицы
column_name sql_identifier	Имя столбца
option_name sql_identifier	Имя параметра
option_value character_data	Значение параметра

A.13. column_privileges

Представление column_privileges описывает все права, назначенные текущей активной роли или текущей активной ролью для столбцов. Оно содержит отдельную строку для каждой комбинации столбца, праводателя и правообладателя.

Если право даётся для всей таблицы, оно будет показываться как право для каждого столбца, но только для типов прав, применимых к столбцам: SELECT, INSERT, UPDATE, REFERENCES.

Таблица A.13. Столбцы column_privileges

Тип столбца	**Описание**
grantor sql_identifier	Имя роли, давшей право (праводатель)
grantee sql_identifier	Имя роли, которой было дано право (правообладатель)
table_catalog sql_identifier	Имя базы данных, содержащей таблицу с этим столбцом (всегда текущая база)
table_schema sql_identifier	Имя схемы, содержащей таблицу с этим столбцом
table_name sql_identifier	Имя таблицы с этим столбцом
column_name sql_identifier	Имя столбца
privilege_type character_data	Тип права: SELECT, INSERT, UPDATE или REFERENCES
is_grantable yes_or_no	YES, если право может передаваться, или NO в противном случае

A.14. column_udt_usage

Представление column_udt_usage описывает все столбцы, которые используют типы данных, принадлежащие текущей активной роли.

Таблица A.14. Столбцы column_udt_usage

Тип столбца	**Описание**
udt_catalog sql_identifier	Имя базы данных, в которой определён тип (если применимо, нижележащий тип домена) столбца (всегда текущая база)
udt_schema sql_identifier	Имя схемы, в которой определён тип (если применимо, нижележащий тип домена) столбца (всегда текущая база)
udt_name sql_identifier	Имя типа данных столбца (если применимо, нижележащий тип домена)
table_catalog sql_identifier	Имя базы данных, содержащей таблицу (всегда текущая база)
table_schema sql_identifier	Имя схемы, содержащей таблицу
table_name sql_identifier	Имя таблицы
column_name sql_identifier	Имя столбца

A.15.columns

Представление columns содержит информацию обо всех столбцах таблиц (или столбцах представлений) в базе данных. Системные столбцы (ctid и т. д.) в нём не отображаются. В нём показываются только те столбцы, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица A.15. Столбцы columns

Тип столбца	**Описание**
table_catalog sql_identifier	Имя базы данных, содержащей таблицу (всегда текущая база)
table_schema sql_identifier	Имя схемы, содержащей таблицу
table_name sql_identifier	Имя таблицы
column_name sql_identifier	Имя столбца
ordinal_position cardinal_number	Порядковый номер столбца в таблице (нумерация начинается с 1)
column_default character_data	Выражение по умолчанию для столбца
is_nullable yes_or_no	YES, если столбец может содержать NULL, или NO, если он не принимает NULL. Не будет принимать NULL столбец с ограничением NOT NULL, но возможны и другие варианты.
data_type character_data	Тип данных столбца, если это встроенный тип, либо ARRAY, если это массив (в этом случае необходимо обратиться к представлению element_types), иначе — USER-DEFINED (в этом случае тип определяется в udt_name и связанных столбцах). Если столбец основан на домене, данный столбец показывает нижележащий тип домена (а сам домен показывается в domain_name и связанных столбцах).
character_maximum_length cardinal_number	Если в data_type указан тип текстовой или битовой строки, это поле задаёт её объявленную максимальную длину; NULL для всех других типов данных, либо если максимальная длина не объявлена.
character_octet_length cardinal_number	Если в data_type указан тип символьной строки, это поле задаёт её максимально возможный размер в октетах (байтах); NULL для всех других типов данных. Максимальный размер в октетах зависит от объявленной максимальной длины в символах (см. выше) и от кодировки сервера.
numeric_precision cardinal_number	Если в data_type указан числовой тип, этот столбец содержит точность (объявленную или неявную) типа для целевого столбца. Точность определяет число значащих цифр. Она может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу numeric_precision_radix . Для всех других типов данных этот столбец содержит NULL.
numeric_precision_radix cardinal_number	Если в data_type указан числовой тип, в этом столбце определяется, по какому основанию задаются значения в столбцах numeric_precision и numeric_scale . Возможные варианты: 2 или 10. Для всех других типов данных этот столбец содержит NULL.
numeric_scale cardinal_number	Если в data_type указан точный числовой тип, этот столбец содержит масштаб (объявленный или неявный) типа для целевого столбца. Масштаб определяет число значащих цифр справа от десятичной точки. Он может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу numeric_ precision_radix . Для всех других типов данных этот столбец содержит NULL.

Тип столбца	Описание
datetime_precision cardinal_number	Если в data_type указан тип даты, времени, отметки времени или интервала, этот столбец содержит точность (объявленную или неявную) в долях секунды типа для целевого столбца, то есть число десятичных цифр, сохраняемых после десятичной точки в значении секунд. Для всех других типов данных этот столбец содержит NULL.
interval_type character_data	Если в data_type указан тип интервала, этот столбец определяет, какие поля принимает интервал в целевом столбце, например: YEAR TO MONTH, DAY TO SECOND и т. д. Если ограничения для полей не заданы (то есть, интервал принимает все поля), и для любых других типов данных это поле содержит NULL.
collation_catalog sql_identifier	Имя базы данных, содержащей правило сортировки столбца (это всегда текущая база), либо NULL, если это правило по умолчанию или тип данных столбца несортируемый
collation_schema sql_identifier	Имя схемы, содержащей правило сортировки столбца, либо NULL, если это правило по умолчанию или тип данных столбца несортируемый
collation_name sql_identifier	Имя правила сортировки столбца, либо NULL, если это правило по умолчанию или тип данных столбца несортируемый
domain_catalog sql_identifier	Если целевой столбец имеет тип домена, этот столбец содержит имя базы данных, в которой определён домен (всегда текущая база), иначе — NULL.
domain_schema sql_identifier	Если целевой столбец имеет тип домена, этот столбец содержит имя схемы, в которой определён домен, иначе — NULL.
domain_name sql_identifier	Если целевой столбец имеет тип домена, этот столбец содержит имя домена, иначе — NULL.
udt_catalog sql_identifier	Имя базы данных, в которой определён тип (если применимо, нижележащий тип домена) столбца (всегда текущая база)
udt_schema sql_identifier	Имя схемы, в которой определён тип (если применимо, нижележащий тип домена) столбца (всегда текущая база)
udt_name sql_identifier	Имя типа данных столбца (если применимо, нижележащий тип домена)
maximum_cardinality cardinal_number	Всегда NULL, так как массивы имеют неограниченную максимальную ёмкость в PG360
dtd_identifier sql_identifier	Идентификатор дескриптора типа данных столбца, уникальный среди всех дескрипторов типов, относящихся к таблице. Он в основном полезен для соединения с другими экземплярами таких идентификаторов. (Конкретный формат идентификатора не определён и не гарантируется, что он останется неизменным в будущих версиях.)
is_identity yes_or_no	Если целевой столбец является столбцом идентификации, значение YES, иначе — NO.

Тип столбца	Описание
identity_generation character_data	Если целевой столбец является столбцом идентификации, значение ALWAYS или BY DEFAULT, отражающее определение столбца.
identity_start character_data	Если целевой столбец является столбцом идентификации, начальное значение внутренней последовательности, иначе — NULL.
identity_increment character_data	Если целевой столбец является столбцом идентификации, шаг внутренней последовательности, иначе — NULL.
identity_maximum character_data	Если целевой столбец является столбцом идентификации, максимальное значение внутренней последовательности, иначе — NULL.
identity_minimum character_data	Если целевой столбец является столбцом идентификации, минимальное значение внутренней последовательности, иначе — NULL.
identity_cycle yes_or_no	Если целевой столбец является столбцом идентификации, YES показывает, что внутренняя последовательность зацикливается, NO — не зацикливается, иначе — NULL.
is_generated character_data	ALWAYS, если целевой столбец является генерируемым, иначе — NEVER.
generation_expression character_data	Генерирующее выражение, если целевой столбец является генерируемым, иначе — NULL.
is_updatable yes_or_no	YES, если столбец допускает изменение, или NO в противном случае (столбцы в базовых таблицах всегда изменяемые, но в представлениях — не обязательно)

A.16. constraint_column_usage

Представление constraint_column_usage описывает все столбцы в текущей базе данных, связанные с некоторым ограничением. В нём показываются только столбцы таблиц, принадлежащих текущей активной роли. Для ограничений-проверок это представление содержит столбцы, задействованные в выражении проверки. Для ограничений внешнего ключа оно содержит столбцы, на которые ссылается внешний ключ, а для ограничений уникальности или первичного ключа — ограничиваемые столбцы.

Таблица A.16. Столбцы constraint_column_usage

Тип столбца	Описание
table_catalog sql_identifier	Имя базы данных, содержащей таблицу со столбцом, задействованным в некотором ограничении (всегда текущая база)
table_schema sql_identifier	Имя схемы, содержащей таблицу со столбцом, задействованным в некотором ограничении
table_name sql_identifier	Имя таблицы со столбцом, задействованным в некотором ограничении
column_name sql_identifier	Имя столбца, задействованного в некотором ограничении
constraint_catalog sql_identifier	Имя базы данных, содержащей ограничение (всегда текущая база)
constraint_schema sql_identifier	Имя схемы, содержащей ограничение
constraint_name sql_identifier	Имя ограничения

A.17. constraint_table_usage

Представление constraint_table_usage описывает все таблицы в текущей базе данных, связанные с некоторым ограничением и принадлежащие текущей активной роли. Для ограничений внешнего ключа это представление показывает таблицу, на которую ссылается ограничение. Для ограничений уникальности или первичного ключа в этом представлении показывается таблица, которой принадлежит ограничение. Ограничения-проверки и ограничения NOT NULL в нём не отражаются.

Таблица A.17. Столбцы constraint_table_usage

Тип столбца	**Описание**
table_catalog sql_identifier	Имя базы данных, которая содержит таблицу, задействованную некоторым ограничением (всегда текущая база)
table_schema sql_identifier	Имя схемы, которая содержит таблицу, задействованную некоторым ограничением
table_name sql_identifier	Имя таблицы, задействованной некоторым ограничением
constraint_catalog sql_identifier	Имя базы данных, содержащей ограничение (всегда текущая база)
constraint_schema sql_identifier	Имя схемы, содержащей ограничение
constraint_name sql_identifier	Имя ограничения

A.18. data_type_privileges

Представление data_type_privileges описывает все дескрипторы типов данных, к которым имеет доступ текущий пользователь, являясь их владельцем или имея некоторые права для них. Дескриптор типа данных формируется, когда тип данных задействуется в определении столбца таблицы, домена или функции (в качестве типа параметра или результата), и хранит некоторую информацию о том, как этот тип используется в данном случае. Каждому дескриптору типа данных назначается уникальный идентификатор, уникальный среди всех дескрипторов типов, назначаемых одному объекту (таблица, домен, функция).

Таблица A.18. Столбцы data_type_privileges

Тип столбца	**Описание**
object_catalog sql_identifier	Имя базы данных, содержащей описываемый объект (всегда текущая база)
object_schema sql_identifier	Имя схемы, содержащей описываемый объект
object_name sql_identifier	Имя описываемого объекта
object_type character_data	Тип описываемого объекта: TABLE (дескриптор типа данных относится к столбцу этой таблицы), DOMAIN (дескриптор типа данных относится к домену) или ROUTINE (дескриптор типа данных относится к типу данных параметра или результата функции).
dtd_identifier sql_identifier	Идентификатор дескриптора типа данных, уникальный среди дескрипторов типов для этого же объекта.

A.19. domain_constraints

Представление domain_constraints показывает все ограничения, принадлежащие доменам, определённым в текущей базе данных. В нём отражаются только те домены, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица A.19. Столбцы domain_constraints

Тип столбца	**Описание**
constraint_catalog sql_identifier	Имя базы данных, содержащей ограничение (всегда текущая база)
constraint_schema sql_identifier	Имя схемы, содержащей ограничение
constraint_name sql_identifier	Имя ограничения
domain_catalog sql_identifier	Имя базы данных, содержащей домен (всегда текущая база)
domain_schema sql_identifier	Имя схемы, содержащей домен
domain_name sql_identifier	Имя домена
is_deferrable yes_or_no	YES, если ограничение откладываемое, или NO в противном случае
initially_deferred yes_or_no	YES, если ограничение откладываемое и отложенное изначально, или NO в противном случае

A.20. domain_udt_usage

Представление domain_udt_usage описывает все домены, которые используют типы данных, принадлежащие текущей активной роли.

Таблица A.20. Столбцы domain_udt_usage

Тип столбца	**Описание**
udt_catalog sql_identifier	Имя базы данных, в которой определён тип данных домена (всегда текущая база)
udt_schema sql_identifier	Имя схемы, в которой определён тип данных домена
udt_name sql_identifier	Имя типа данных домена
domain_catalog sql_identifier	Имя базы данных, содержащей домен (всегда текущая база)
domain_schema sql_identifier	Имя схемы, содержащей домен
domain_name sql_identifier	Имя домена

A.21. domains

Представление domains показывает все домены, определённые в текущей базе данных. В нём показываются только те домены, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица A.21. Столбцы domains

Тип столбца	**Описание**
domain_catalog sql_identifier	Имя базы данных, содержащей домен (всегда текущая база)
domain_schema sql_identifier	Имя схемы, содержащей домен
domain_name sql_identifier	Имя домена

Тип столбца	Описание
data_type character_data	Тип данных домена, если это встроенный тип, либо ARRAY, если это массив (в этом случае необходимо обратиться к представлению element_types), иначе — USER-DEFINED (в этом случае тип определяется в udt_name и связанных столбцах).
character_maximum_length cardinal_number	Если домен имеет тип текстовой или битовой строки, это поле задаёт её объявленную максимальную длину; NULL для всех других типов данных, или если максимальная длина не объявлена.
character_octet_length cardinal_number	Если домен имеет тип символьной строки, это поле задаёт её максимально возможный размер в октетах (байтах); NULL для всех других типов данных. Максимальный размер в октетах зависит от объявленной максимальной длины в символах (см. выше) и от кодировки сервера.
collation_catalog sql_identifier	Имя базы данных, содержащей правило сортировки домена (это всегда текущая база), либо NULL, если это правило по умолчанию или тип домена несортируемый
collation_schema sql_identifier	Имя схемы, содержащей правило сортировки домена, либо NULL, если это правило по умолчанию или тип домена несортируемый
collation_name sql_identifier	Имя правила сортировки домена, либо NULL, если это правило по умолчанию или тип домена несортируемый
numeric_precision cardinal_number	Если домен имеет числовой тип, этот столбец содержит точность (объявленную или неявную) типа для этого домена. Точность определяет число значащих цифр. Она может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу numeric_precision_radix . Для всех других типов данных этот столбец содержит NULL.
numeric_precision_radix cardinal_number	Если домен имеет числовой тип, в этом столбце определяется, по какому основанию задаются значения в столбцах numeric_precision и numeric_scale . Возможные варианты: 2 и 10. Для всех других типов данных этот столбец содержит NULL.
numeric_scale cardinal_number	Если домен имеет точный числовой тип, этот столбец содержит масштаб (объявленный или неявный) типа для этого домена. Масштаб определяет число значащих цифр справа от десятичной точки. Он может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу numeric_precision_radix . Для всех других типов данных этот столбец содержит NULL.
datetime_precision cardinal_number	Если в data_type указан тип даты, времени, отметки времени или интервала, этот столбец содержит точность (объявленную или неявную) в долях секунды типа для этого домена, то есть число десятичных цифр, сохраняемых после десятичной точки в значении секунд. Для всех других типов данных этот столбец содержит NULL.

Тип столбца	Описание
interval_type character_data	Если в data_type указан тип интервала, этот столбец определяет, какие поля принимает интервал в домене, например: YEAR TO MONTH, DAY TO SECOND и т. д. Если ограничения для полей не заданы (то есть, интервал принимает все поля), и для любых других типов данных это поле содержит NULL.
domain_default character_data	Выражение по умолчанию для домена
udt_catalog sql_identifier	Имя базы данных, в которой определён тип данных домена (всегда текущая база)
udt_schema sql_identifier	Имя схемы, в которой определён тип данных домена
udt_name sql_identifier	Имя типа данных домена
maximum_cardinality cardinal_number	Всегда NULL, так как массивы имеют неограниченную максимальную ёмкость в PG360
dtd_identifier sql_identifier	Идентификатор дескриптора типа данных целевого домена, уникальный среди всех дескрипторов типов, относящихся к домену (что тривиально, так как домен содержит только один дескриптор типа). Он в основном полезен для соединения с другими экземплярами таких идентификаторов. (Конкретный формат идентификатора не определён и не гарантируется, что он останется неизменным в будущих версиях.)

A.22. element_types

Представление element_types показывает дескрипторы типов элементов массива. Когда столбец таблицы, атрибут составного типа, параметр или результат функции объявлены с типом массива, соответствующее представление информационной схемы будет содержать только ARRAY в столбце data_type.

Это представление показывает только те объекты, к которым имеет доступ текущий пользователь, являясь владельцем или имея некоторые права.

Таблица A.22. Столбцы element_types

Тип столбца	Описание
object_catalog sql_identifier	Имя базы данных, содержащей объект, связанный с описываемым массивом (всегда текущая база)
object_schema sql_identifier	Имя схемы, содержащей объект, связанный с описываемым массивом
object_name sql_identifier	Имя объекта, связанного с описываемым массивом
object_type character_data	Тип объекта, связанного с описываемым массивом: TABLE (массив задействован в столбце этой таблицы), USER-DEFINED TYPE (массив задействован в атрибуте составного типа), DOMAIN (массив задействован в домене), ROUTINE (массив задействован в типе данных параметра или результата функции).

Тип столбца	Описание
collection_type_identifier sql_identifier	Идентификатор дескриптора типа данных для описываемого массива. Его можно использовать для соединения со столбцами dtd_identifier других представлений информационной схемы.
data_type character_data	Тип данных элементов массива, если это встроенный тип, либо USER-DEFINED (в этом случае тип определяется в udt_name и связанных столбцах).
character_maximum_length cardinal_number	Всегда NULL
character_octet_length cardinal_number	Всегда NULL
collation_catalog sql_identifier	Имя базы данных, содержащей правило сортировки типа элемента (это всегда текущая база), либо NULL, если это правило по умолчанию или тип элемента несортируемый
collation_schema sql_identifier	Имя схемы, содержащей правило сортировки типа элемента, либо NULL, если это правило по умолчанию или тип элемента несортируемый
collation_name sql_identifier	Имя правила сортировки типа элемента, либо NULL, если это правило по умолчанию или тип элемента несортируемый
numeric_precision cardinal_number	Всегда NULL
numeric_precision_radix cardinal_number	Всегда NULL
numeric_scale cardinal_number	Всегда NULL
datetime_precision cardinal_number	Всегда NULL
interval_type character_data	Всегда NULL
interval_precision cardinal_number	Всегда NULL
domain_default character_data	Ещё не реализовано
udt_catalog sql_identifier	Имя базы данных, в которой определён тип данных элемента (всегда текущая база)
udt_schema sql_identifier	Имя схемы, в которой определён тип данных элемента
udt_name sql_identifier	Имя типа данных элемента
maximum_cardinality cardinal_number	Всегда NULL
dtd_identifier sql_identifier	Идентификатор дескриптора типа данных элемента. В настоящее время бесполезен.

A.23. enabled_roles

Представление enabled_roles описывает «доступные роли». Список доступных ролей рекурсивно определяется как роль текущего пользователя плюс роли, данные доступным ролям, с автоматическим наследованием.

Таблица A.23. Столбцы enabled_roles

Тип столбца	**Описание**
role_name sql_identifier	Имя целевой роли

A.24. foreign_data_wrapper_options

Представление foreign_data_wrapper_options показывает все параметры, определённые для обёрток сторонних данных в текущей базе. В нём отражаются только те обёртки сторонних данных, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица A.24. Столбцы foreign_data_wrapper_options

Тип столбца	**Описание**
foreign_data_wrapper_catalog sql_identifier	Имя базы данных, в которой определена обёртка сторонних данных (всегда текущая база)
foreign_data_wrapper_name sql_identifier	Имя обёртки сторонних данных
option_name sql_identifier	Имя параметра
option_value character_data	Значение параметра

A.25. foreign_data_wrappers

Представление foreign_data_wrappers показывает все обёртки сторонних данных, определённые в текущей базе. В нём показываются только те обёртки сторонних данных, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица A.25. Столбцы foreign_data_wrappers

Тип столбца	**Описание**
foreign_data_wrapper_catalog sql_identifier	Имя базы данных, в которой определена обёртка сторонних данных (всегда текущая база)
foreign_data_wrapper_name sql_identifier	Имя обёртки сторонних данных
authorization_identifier sql_identifier	Имя владельца стороннего сервера
library_name character_data	Имя файла библиотеки, реализующей эту обёртку сторонних данных
foreign_data_wrapper_language character_data	Язык, на котором реализована эта обёртка сторонних данных

A.26. foreign_server_options

Представление foreign_server_options показывает все параметры, определённые для сторонних серверов в текущей базе данных. В нём отражаются только те сторонние серверы, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица A.26. Столбцы foreign_server_options

Тип столбца	**Описание**
foreign_server_catalog sql_identifier	Имя базы данных, в которой определён сторонний сервер (всегда текущая база)
foreign_server_name sql_identifier	Имя стороннего сервера
option_name sql_identifier	Имя параметра
option_value character_data	Значение параметра

A.27. foreign_servers

Представление foreign_servers показывает все сторонние серверы, определённые в текущей базе данных. В нём показываются только те сторонние серверы, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица A.27. Столбцы foreign_servers

Тип столбца	**Описание**
foreign_server_catalog sql_identifier	Имя базы данных, в которой определён сторонний сервер (всегда текущая база)
foreign_server_name sql_identifier	Имя стороннего сервера
foreign_data_wrapper_catalog sql_identifier	Имя базы данных, в которой определена обёртка сторонних данных, используемая сторонним сервером (всегда текущая база)
foreign_data_wrapper_name sql_identifier	Имя обёртки сторонних данных, используемой сторонним сервером
foreign_server_type character_data	Информация о типе стороннего сервера, если она была указана при его создании
foreign_server_version character_data	Информация о версии стороннего сервера, если она была указана при его создании
authorization_identifier sql_identifier	Имя владельца стороннего сервера

A.28. foreign_table_options

Представление foreign_table_options показывает все параметры, определённые для сторонних таблиц в текущей базе данных. В нём отражаются только те сторонние таблицы, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица A.28. Столбцы foreign_table_options

Тип столбца	**Описание**
foreign_table_catalog sql_identifier	Имя базы данных, содержащей стороннюю таблицу (всегда текущая база)

Тип столбца	Описание
foreign_table_schema sql_identifier	Имя схемы, содержащей стороннюю таблицу
foreign_table_name sql_identifier	Имя сторонней таблицы
option_name sql_identifier	Имя параметра
option_value character_data	Значение параметра

A.29.foreign_tables

Представление foreign_tables показывает все сторонние таблицы, определённые в текущей базе данных. В нём показываются только те сторонние таблицы, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица A.29. Столбцы foreign_tables

Тип столбца	Описание
foreign_table_catalog sql_identifier	Имя базы данных, в которой определена сторонняя таблица (всегда текущая база)
foreign_table_schema sql_identifier	Имя схемы, содержащей стороннюю таблицу
foreign_table_name sql_identifier	Имя сторонней таблицы
foreign_server_catalog sql_identifier	Имя базы данных, в которой определён сторонний сервер (всегда текущая база)
foreign_server_name sql_identifier	Имя стороннего сервера

A.30. key_column_usage

Представление key_column_usage описывает все столбцы в текущей базе, с которыми связано какое-либо ограничение уникальности, либо ограничение первичного или внешнего ключа. Ограничения-проверки в этом представлении не показываются. В нём показываются только те столбцы, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица A.30. Столбцы key_column_usage

Тип столбца	Описание
constraint_catalog sql_identifier	Имя базы данных, содержащей ограничение (всегда текущая база)
constraint_schema sql_identifier	Имя схемы, содержащей ограничение
constraint_name sql_identifier	Имя ограничения
table_catalog sql_identifier	Имя базы данных, содержащей таблицу со столбцом, подчиняющимся этому ограничению (всегда текущая база)
table_schema sql_identifier	Имя схемы, содержащей таблицу со столбцом, подчиняющимся этому ограничению
table_name sql_identifier	Имя таблицы со столбцом, подчиняющимся этому ограничению
column_name sql_identifier	Имя столбца, подчиняющегося этому ограничению

Тип столбца	Описание
ordinal_position cardinal_number	Порядковый номер столбца в ключе ограничения (нумерация начинается с 1)
position_in_unique_constraint cardinal_number	Для ограничения внешнего ключа это порядковый номер целевого столбца в его ограничении уникальности (нумерация начинается с 1); в противном случае NULL

A.31. parameters

Представление parameters содержит информацию о параметрах (аргументах) всех функций в текущей базе данных. В нём отражаются только функции, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица A.31. Столбцы parameters

Тип столбца	Описание
specific_catalog sql_identifier	Имя базы данных, содержащей функцию (всегда текущая база)
specific_schema sql_identifier	Имя схемы, содержащей функцию
specific_name sql_identifier	«Однозначное имя» функции
ordinal_position cardinal_number	Порядковый номер параметра в списке аргументов функции (нумерация начинается с 1)
parameter_mode character_data	IN для входного параметра, OUT для выходного, INOUT — для входного и выходного параметра.
parameter_name sql_identifier	Имя параметра, либо NULL, если параметр безымянный
data_type character_data	Тип данных параметра, если это встроенный тип, либо ARRAY, если это массив (в этом случае необходимо обратиться к представлению element_types), иначе — USER-DEFINED (в этом случае тип определяется в udt_name и связанных столбцах).
character_maximum_length cardinal_number	Всегда NULL
character_octet_length cardinal_number	Всегда NULL
collation_catalog sql_identifier	Всегда NULL
collation_schema sql_identifier	Всегда NULL
collation_name sql_identifier	Всегда NULL
numeric_precision cardinal_number	Всегда NULL
numeric_precision_radix cardinal_number	Всегда NULL
numeric_scale cardinal_number	Всегда NULL
datetime_precision cardinal_number	Всегда NULL

Тип столбца	Описание
interval_type character_data	Всегда NULL
interval_precision cardinal_number	Всегда NULL
udt_catalog sql_identifier	Имя базы данных, в которой определён тип данных параметра (всегда текущая база)
udt_schema sql_identifier	Имя схемы, в которой определён тип данных параметра
udt_name sql_identifier	Имя типа данных параметра
maximum_cardinality cardinal_number	Всегда NULL
dtd_identifier sql_identifier	Идентификатор дескриптора типа данных параметра, уникальный среди всех дескрипторов типов, относящихся к функции.
parameter_default character_data	Выражение параметра по умолчанию, либо NULL, если такого выражения нет или функция не принадлежит текущей активной роли.

A.32. referential_constraints

Представление referential_constraints содержит все ссылочные ограничения (внешнего ключа) в текущей базе данных. В нём показываются только ограничения, в которых ссылающаяся таблица доступна текущему пользователю на запись (он является её владельцем или имеет не только право SELECT).

Таблица A.32. Столбцы referential_constraints

Тип столбца	Описание
constraint_catalog sql_identifier	Имя базы данных, содержащей ограничение (всегда текущая база)
constraint_schema sql_identifier	Имя схемы, содержащей ограничение
constraint_name sql_identifier	Имя ограничения
unique_constraint_catalog sql_identifier	Имя базы данных, содержащей ограничение уникальности или первичный ключ, на которые ссылается ограничение внешнего ключа (всегда текущая база)
unique_constraint_schema sql_identifier	Имя схемы, содержащей ограничение уникальности или первичный ключ, на которые ссылается ограничение внешнего ключа
unique_constraint_name sql_identifier	Имя ограничения уникальности или первичного ключа, на которые ссылается ограничение внешнего ключа
match_option character_data	Тип совпадения для ограничения внешнего ключа: FULL, PARTIAL или NONE.
update_rule character_data	Правило изменения для ограничения внешнего ключа: CASCADE, SET NULL, SET DEFAULT, RESTRICT или NO ACTION.
delete_rule character_data	Правило удаления для ограничения внешнего ключа: CASCADE, SET NULL, SET DEFAULT, RESTRICT или NO ACTION.

A.33. role_column_grants

Представление role_column_grants описывает все назначенные для столбцов права, в которых праводателем или правообладателем является текущая активная роль.

Таблица A.33. Столбцы role_column_grants

Тип столбца	**Описание**
grantor sql_identifier	Имя роли, давшей право (праводатель)
grantee sql_identifier	Имя роли, которой было дано право (правообладатель)
table_catalog sql_identifier	Имя базы данных, содержащей таблицу с этим столбцом (всегда текущая база)
table_schema sql_identifier	Имя схемы, содержащей таблицу с этим столбцом
table_name sql_identifier	Имя таблицы с этим столбцом
column_name sql_identifier	Имя столбца
privilege_type character_data	Тип права: SELECT, INSERT, UPDATE или REFERENCES
is_grantable yes_or_no	YES, если право может передаваться, или NO в противном случае

A.34. role_routine_grants

Представление role_routine_grants описывает все назначенные для функций права, в которых праводателем или правообладателем является текущая активная роль.

Таблица A.34. Столбцы role_routine_grants

Тип столбца	**Описание**
grantor sql_identifier	Имя роли, давшей право (праводатель)
grantee sql_identifier	Имя роли, которой было дано право (правообладатель)
specific_catalog sql_identifier	Имя базы данных, содержащей функцию (всегда текущая база)
specific_schema sql_identifier	Имя схемы, содержащей функцию
specific_name sql_identifier	«Однозначное имя» функции
routine_catalog sql_identifier	Имя базы данных, содержащей функцию (всегда текущая база)
routine_schema sql_identifier	Имя схемы, содержащей функцию
routine_name sql_identifier	Имя функции (может дублироваться в случае перегрузки)
privilege_type character_data	Всегда EXECUTE (единственный тип прав для функций)
is_grantable yes_or_no	YES, если право может передаваться, или NO в противном случае

A.35. role_table_grants

Представление role_table_grants описывает все назначенные для таблиц и представлений права, в которых праводателем или правообладателем является текущая активная роль.

Таблица A.35. Столбцы role_table_grants

Тип столбца	**Описание**
grantor sql_identifier	Имя роли, давшей право (праводатель)
grantee sql_identifier	Имя роли, которой было дано право (правообладатель)

Тип столбца	Описание
table_catalog sql_identifier	Имя базы данных, содержащей таблицу (всегда текущая база)
table_schema sql_identifier	Имя схемы, содержащей таблицу
table_name sql_identifier	Имя таблицы
privilege_type character_data	Тип права: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES или TRIGGER
is_grantable yes_or_no	YES, если право может передаваться, или NO в противном случае
with_hierarchy yes_or_no	YES для права SELECT, а для других — NO.

A.36. role_udt_grants

Представление role_udt_grants предназначено для отображения прав USAGE, назначенных для пользовательских типов, в которых праводателем или правообладателем является текущая активная роль.

Таблица A.36. Столбцы role_udt_grants

Тип столбца	Описание
grantor sql_identifier	Имя роли, которая дала это право
grantee sql_identifier	Имя роли, которой было дано это право
udt_catalog sql_identifier	Имя базы данных, содержащей тип (всегда текущая база)
udt_schema sql_identifier	Имя схемы, содержащей тип
udt_name sql_identifier	Имя типа
privilege_type character_data	Всегда TYPE USAGE
is_grantable yes_or_no	YES, если право может передаваться, или NO в противном случае

A.37. role_usage_grants

Представление role_usage_grants описывает права USAGE, назначенные для объектов различных типов, в которых праводателем или правообладателем является текущая активная роль.

Таблица A.37. Столбцы role_usage_grants

Тип столбца	Описание
grantor sql_identifier	Имя роли, которая дала это право
grantee sql_identifier	Имя роли, которой было дано это право
object_catalog sql_identifier	Имя базы данных, содержащей объект (всегда текущая база)
object_schema sql_identifier	Имя схемы, содержащей объект, если это применимо, иначе пустая строка
object_name sql_identifier	Имя объекта
object_type character_data	COLLATION или DOMAIN или FOREIGN DATA WRAPPER или FOREIGN SERVER или SEQUENCE
privilege_type character_data	Всегда USAGE

Тип столбца	Описание
is_grantable yes_or_no	YES, если право может передаваться, или NO в противном случае

A.38. routine_privileges

Представление routine_privileges описывает все права, назначенные текущей активной роли или текущей активной ролью для функций, содержащее отдельный столбец для каждой комбинации функции, праводателя и правообладателя.

Таблица A.38. Столбцы routine_privileges

Тип столбца	Описание
grantor sql_identifier	Имя роли, давшей право (праводатель)
grantee sql_identifier	Имя роли, которой было дано право (правообладатель)
specific_catalog sql_identifier	Имя базы данных, содержащей функцию (всегда текущая база)
specific_schema sql_identifier	Имя схемы, содержащей функцию
specific_name sql_identifier	«Однозначное имя» функции
routine_catalog sql_identifier	Имя базы данных, содержащей функцию (всегда текущая база)
routine_schema sql_identifier	Имя схемы, содержащей функцию
routine_name sql_identifier	Имя функции (может дублироваться в случае перегрузки)
privilege_type character_data	Всегда EXECUTE (единственный тип прав для функций)
is_grantable yes_or_no	YES, если право может передаваться, или NO в противном случае

A.39. routines

Представление routines отображает все функции и процедуры в текущей базе данных, в котором показываются только те функции и процедуры, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица A.39. Столбцы routines

Тип столбца	Описание
specific_catalog sql_identifier	Имя базы данных, содержащей функцию (всегда текущая база)
specific_schema sql_identifier	Имя схемы, содержащей функцию
specific_name sql_identifier	«Однозначное имя» функции. Это имя однозначным образом идентифицирует функцию в схеме, даже если реальное имя функции перегружено. Формат однозначных имён не определён, так что его следует использовать только для сравнения с другими экземплярами однозначных имён подпрограмм.
routine_catalog sql_identifier	Имя базы данных, содержащей функцию (всегда текущая база)
routine_schema sql_identifier	Имя схемы, содержащей функцию

Тип столбца	**Описание**
routine_name sql_identifier	Имя функции (может дублироваться в случае перегрузки)
routine_type character_data	FUNCTION для функций, PROCEDURE для процедур
data_type character_data	Тип данных результата функции, если это встроенный тип, либо ARRAY, если это массив (в этом случае необходимо обратиться к представлению element_types), иначе — USER-DEFINED (в этом случае тип определяется в type_udt_name и связанных столбцах). Для процедуры данное поле содержит NULL.
character_maximum_length cardinal_number	Всегда NULL
character_octet_length cardinal_number	Всегда NULL
collation_catalog sql_identifier	Всегда NULL
collation_schema sql_identifier	Всегда NULL
collation_name sql_identifier	Всегда NULL
numeric_precision cardinal_number	Всегда NULL
numeric_precision_radix cardinal_number	Всегда NULL
numeric_scale cardinal_number	Всегда NULL
datetime_precision cardinal_number	Всегда NULL
interval_type character_data	Всегда NULL
interval_precision cardinal_number	Всегда NULL
type_udt_catalog sql_identifier	Имя базы данных, в которой определён тип данных результата функции (всегда текущая база). Для процедуры данное поле содержит NULL.
type_udt_schema sql_identifier	Имя схемы, в которой определён тип данных результата функции. Для процедуры данное поле содержит NULL.
type_udt_name sql_identifier	Имя типа данных результата функции. Для процедуры данное поле содержит NULL.
maximum_cardinality cardinal_number	Всегда NULL

Тип столбца	Описание
dtd_identifier sql_identifier	Идентификатор дескриптора типа данных результата функции, уникальный среди всех дескрипторов типов, относящихся к функции. Он в основном полезен для соединения с другими экземплярами таких идентификаторов. (Конкретный формат идентификатора не определён и не гарантируется, что он останется неизменным в будущих версиях.)
routine_body character_data	Если функция написана на SQL, это поле содержит SQL, иначе EXTERNAL.
routine_definition character_data	Исходный текст функции (NULL, если функция не принадлежит текущей активной роли). (Согласно стандарту SQL, этот столбец актуален, только если в routine_body указано SQL, но в PG360 он будет содержать любой исходный текст, заданный при создании функции.)
external_name character_data	Если это функция на C, этот столбец содержит внешнее имя (объектный символ) функции, иначе — NULL. (Это будет то же значение, что содержит столбец routine_ definition .)
external_language character_data	Язык, на котором написана функция
parameter_style character_data	Всегда GENERAL
is_deterministic yes_or_no	Если функция объявлена как постоянная (IMMUTABLE), этот столбец содержит YES, иначе — NO.
sql_data_access character_data	Всегда MODIFIES, что означает, что функция может модифицировать данные SQL.
is_null_call yes_or_no	Если функция автоматически возвращает NULL, когда один из аргументов NULL, этот столбец содержит YES, иначе — NO. Для процедуры он содержит NULL.
schema_level_routine yes_or_no	Всегда YES
security_type character_data .	Если функция выполняется с правами вызывающего пользователя, этот столбец содержит INVOKER, а если с правами пользователя, создавшего её, то — DEFINER
is_udt_dependent yes_or_no	Всегда NO

A.40. schemata

Представление schemata показывает все схемы в текущей базе данных, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица A.40. Столбцы schemata

Тип столбца	Описание
catalog_name sql_identifier	Имя базы данных, содержащей схему (всегда текущая база)
schema_name sql_identifier	Имя схемы
schema_owner sql_identifier	Имя владельца схемы

A.41. sequences

Представление sequences показывает все последовательности, определённые в текущей базе данных, в котором показываются только те последовательности, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица A.41. Столбцы sequences

Тип столбца	**Описание**
sequence_catalog sql_identifier	Имя базы данных, содержащей последовательность (всегда текущая база)
sequence_schema sql_identifier	Имя схемы, содержащей последовательность
sequence_name sql_identifier	Имя последовательности
data_type character_data	Тип данных последовательности.
numeric_precision cardinal_number	Содержит точность (объявленную или неявную) типа данных последовательности (см. выше). Точность определяет число значащих цифр, выражающуюся в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу numeric_precision_radix .
numeric_precision_radix cardinal_number	Этот столбец определяет, по какому основанию задаются значения в столбцах numeric_ precision и numeric_scale . Возможные варианты: 2 и 10.
numeric_scale cardinal_number	Содержит масштаб (объявленный или неявный) типа данных последовательности. Масштаб определяет число значащих цифр справа от десятичной точки, выражающийся в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу numeric_precision_radix .
start_value character_data	Начальное значение последовательности
minimum_value character_data	Минимальное значение последовательности
maximum_value character_data	Максимальное значение последовательности
increment character_data	Шаг увеличения последовательности
cycle_option yes_or_no	YES, если последовательность зацикливается, или NO в противном случае

A.42. sql_features

Таблица sql_features содержит информацию о функциональности.

Таблица A.42. Столбцы sql_features

Тип столбца	**Описание**
feature_id character_data	Строка идентификатора функциональности
feature_name character_data	Описательное название функциональности
sub_feature_idcharacter_data	Строка идентификатора подчинённой возможности, либо строка нулевой длины, если это не подчинённая возможность
sub_feature_name character_data	Описательное название подчинённой возможности, либо строка нулевой длины, если это не подчинённая возможность

Тип столбца	Описание
is_supported yes_or_no	YES, если функциональность полностью поддерживается текущей версией PG360, либо NO в противном случае
is_verified_by character_data	Всегда NULL
comments character_data	Необязательный комментарий о поддерживаемом состоянии функциональности

A.43. sql_implementation_info

Таблица sql_implementation_info содержит информацию о различных аспектах, которые в стандарте SQL оставлены на усмотрение реализации. Эта информация в основном предназначена для применения в контексте интерфейса ODBC

Таблица A.43. Столбцы sql_implementation_info

Тип столбца	Описание
implementation_info_id character_data	Строка идентификатора элемента особенности реализации
implementation_info_name character_data	Описательное название элемента особенности реализации
integer_value cardinal_number	Значение элемента особенности реализации, либо NULL, если его значение содержится в столбце character_value
character_value character_data	Значение элемента особенности реализации, либо NULL, если его значение содержится в столбце integer_value
comments character_data	Необязательный комментарий, относящийся к элементу особенности реализации

A.44. sql_parts

Таблица sql_parts содержит информацию о различных частях стандарта SQL, поддерживаемых PG360.

Таблица A.44. Столбцы sql_parts

Тип столбца	Описание
feature_id character_data	Строка идентификатора, содержащая номер части
feature_name character_data	Описательное название части
is_supported yes_or_no	YES, если часть полностью поддерживается текущей версией PG360, либо NO в противном случае
is_verified_by character_data	Всегда NULL
comments character_data	Необязательный комментарий о поддерживаемом состоянии части

A.45. sql_sizing

Таблица sql_sizing содержит информацию о различных ограничениях размера и максимальных значениях в PG360. Эта информация в основном предназначена для применения в контексте интерфейса ODBC.

Таблица А.45. Столбцы sql_sizing

Тип столбца	Описание
sizing_id cardinal_number	Идентификатор элемента размеров
sizing_name character_data	Описательное название элемента размеров
supported_value cardinal_number	Значение элемента размеров, или 0, если размер неограниченный или не может быть определён, либо NULL, если функциональность, к которой относится размер, не поддерживается
comments character_data	Необязательный комментарий, относящийся к элементу размеров

A.46. table_constraints

Представление table_constraints показывает все ограничения, принадлежащие таблицам, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права, кроме SELECT).

Таблица А.46. Столбцы table_constraints

Тип столбца	Описание
constraint_catalog sql_identifier	Имя базы данных, содержащей ограничение (всегда текущая база)
constraint_schema sql_identifier	Имя схемы, содержащей ограничение
constraint_name sql_identifier	Имя ограничения
table_catalog sql_identifier	Имя базы данных, содержащей таблицу (всегда текущая база)
table_schema sql_identifier	Имя схемы, содержащей таблицу
table_name sql_identifier	Имя таблицы
constraint_type character_data	Тип ограничения: CHECK, FOREIGN KEY, PRIMARY KEY или UNIQUE
is_deferrable yes_or_no	YES, если ограничение откладываемое, или NO в противном случае
initially_deferred yes_or_no	YES, если ограничение откладываемое и отложенное изначально, или NO в противном случае

A.47. table_privileges

Представление table_privileges описывает все права, назначенные текущей активной роли или текущей активной ролью для таблиц и представлений, содержащее отдельную строку для каждой комбинации таблицы, праводателя и правообладателя.

Таблица А.47. Столбцы table_privileges

Тип столбца	Описание
grantor sql_identifier	Имя роли, давшей право (праводатель)
grantee sql_identifier	Имя роли, которой было дано право (правообладатель)
table_catalog sql_identifier	Имя базы данных, содержащей таблицу (всегда текущая база)
table_schema sql_identifier	Имя схемы, содержащей таблицу
table_name sql_identifier	Имя таблицы

Тип столбца	Описание
privilege_type character_data	Тип права: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES или TRIGGER
is_grantable yes_or_no	YES, если право может передаваться, или NO в противном случае
with_hierarchy yes_or_no	В стандарте SQL имеется отдельное подчинённое разрешение WITH HIERARCHY OPTION, позволяющее выполнять определённые операции в иерархии наследования таблиц. В PG360 так действует право SELECT, так что в этом столбце выводится YES для права SELECT, а для других — NO.

A.48. tables

Представление tables показывает все таблицы и представления, определённые в текущей базе данных. В нём показываются только те таблицы и представления, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица A.48. Столбцы tables

Тип столбца	Описание
table_catalog sql_identifier	Имя базы данных, содержащей таблицу (всегда текущая база)
table_schema sql_identifier	Имя схемы, содержащей таблицу
table_name sql_identifier	Имя таблицы
table_type character_data	Тип таблицы: BASE TABLE для постоянных базовых таблиц (таблиц обычного типа), VIEW для представлений, FOREIGN для сторонних таблиц, либо LOCAL TEMPORARY для временных таблиц
user_defined_type_catalog sql_identifier	Если таблица является типизированной, это имя базы данных, содержащей нижележащий тип данных (всегда текущая база), иначе — NULL.
user_defined_type_schema sql_identifier	Если таблица является типизированной, это имя схемы, содержащей нижележащий тип данных, иначе — NULL.
user_defined_type_name sql_identifier	Если таблица является типизированной, это имя типа данных, иначе — NULL.
is_insertable_into yes_or_no	YES, если в эту таблицу можно добавлять данные, или NO в противном случае (Базовые таблицы всегда допускают добавление данных, но представления — не обязательно.)
is_typed yes_or_no	YES, если эта таблица является типизированной, иначе — NO
commit_action character_data	Ещё не реализовано

A.49. transforms

Представление transforms содержит информацию о трансформациях, определённых в текущей базе данных.

Таблица A.49. Столбцы transforms

Тип столбца	**Описание**
udt_catalog sql_identifier	Имя базы данных, содержащей тип, для которого предназначена трансформация (всегда текущая база)
udt_schema sql_identifier	Имя схемы, содержащей тип, для которого предназначена трансформация
udt_name sql_identifier	Имя типа, для которого предназначена трансформация
specific_catalog sql_identifier	Имя базы данных, содержащей функцию (всегда текущая база)
specific_schema sql_identifier	Имя схемы, содержащей функцию
specific_name sql_identifier	«Однозначное имя» функции
group_name sql_identifier	Стандарт SQL позволяет определять «группы» трансформаций и выбирать группу во время выполнения. PG360 это не поддерживает. Вместо этого трансформации привязываются к языкам. В качестве компромисса со стандартом, это поле содержит язык, для которого предназначена трансформация.
transform_type character_data	FROM SQL или TO SQL

A.50. triggered_update_columns

Для триггеров в текущей базе данных, установленных для списка столбцов (например, UPDATE OF column1, column2), представление triggered_update_columns показывает эти столбцы. Триггеры, для которых не задаётся список столбцов, в этом представлении не отражаются. В нём показываются только столбцы, доступные текущему пользователю (как владельцу или имеющему некоторые права, кроме SELECT).

Таблица A.50. Столбцы triggered_update_columns

Тип столбца	**Описание**
trigger_catalog sql_identifier	Имя базы данных, содержащей триггер (всегда текущая база)
trigger_schema sql_identifier	Имя схемы, содержащей триггер
trigger_name sql_identifier	Имя триггера
event_object_catalog sql_identifier	Имя базы данных, содержащей таблицу, для которой определён триггер (всегда текущая база)
event_object_schema sql_identifier	Имя схемы, содержащей таблицу, для которой определён триггер
event_object_table sql_identifier	Имя таблицы, для которой определён триггер
event_object_column sql_identifier	Имя столбца, для которого определён триггер

A.51. triggers

Представление triggers показывает все триггеры, определённые в текущей базе данных для таблиц и представлений, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права, кроме SELECT).

Таблица A.51. Столбцы triggers

Тип столбца	Описание
trigger_catalog sql_identifier	Имя базы данных, содержащей триггер (всегда текущая база)
trigger_schema sql_identifier	Имя схемы, содержащей триггер
trigger_name sql_identifier	Имя триггера
event_manipulation character_data	Событие, вызывающие срабатывание триггера (INSERT, UPDATE или DELETE)
event_object_catalog sql_identifier	Имя базы данных, содержащей таблицу, для которой определён триггер (всегда текущая база)
event_object_schema sql_identifier	Имя схемы, содержащей таблицу, для которой определён триггер
event_object_table sql_identifier	Имя таблицы, для которой определён триггер
action_order cardinal_number	Порядок срабатывания триггеров, имеющих одинаковые свойства event_manipulation , action_timing и action_orientation .
action_condition character_data	Условие WHEN триггера, либо NULL, если его нет (так же NULL, если таблица не принадлежит текущей активной роли)
action_statement character_data	Оператор, выполняемый триггером
action_orientation character_data	Определяет, срабатывает ли триггер для каждой обрабатываемой строки или только для каждого оператора (ROW или STATEMENT)
action_timing character_data	Момент срабатывания триггера (BEFORE (до), AFTER (после) или INSTEAD OF (вместо))
action_reference_old_table sql_identifier	Имя «старой» переходной таблицы либо NULL, если её нет
action_reference_new_table sql_identifier	Имя «новой» переходной таблицы либо NULL, если её нет

A.52. udt_privileges

Представление udt_privileges описывает права USAGE, назначенные текущей активной роли или текущей активной ролью для пользовательских типов. Оно содержит отдельную строку для каждой комбинации типа, праводателя и правообладателя.

Таблица A.52. Столбцы udt_privileges

Тип столбца	Описание
grantor sql_identifier	Имя роли, давшей право (праводатель)
grantee sql_identifier	Имя роли, которой было дано право (правообладатель)
udt_catalog sql_identifier	Имя базы данных, содержащей тип (всегда текущая база)
udt_schema sql_identifier	Имя схемы, содержащей тип
udt_name sql_identifier	Имя типа
privilege_type character_data	Всегда TYPE USAGE
is_grantable yes_or_no	YES, если право может передаваться, или NO в противном случае

A.53. usage_privileges

Представление usage_privileges описывает права USAGE, назначенные текущей активной роли или текущей активной ролью для различных типов объектов (правила сортировки, домены, обёртки сторонних данных, сторонние серверы и последовательности). Оно содержит отдельную строку для каждой комбинации объекта, праводателя и правообладателя.

Таблица A.53. Столбцы usage_privileges

Тип столбца	**Описание**
grantor sql_identifier	Имя роли, давшей право (праводатель)
grantee sql_identifier	Имя роли, которой было дано право (правообладатель)
object_catalog sql_identifier	Имя базы данных, содержащей объект (всегда текущая база)
object_schema sql_identifier	Имя схемы, содержащей объект, если это применимо, иначе пустая строка
object_name sql_identifier	Имя объекта
object_type character_data	COLLATION или DOMAIN или FOREIGN DATA WRAPPER или FOREIGN SERVER или SEQUENCE
privilege_type character_data	Всегда USAGE
is_grantable yes_or_no	YES, если право может передаваться, или NO в противном случае

A.54. user_defined_types

Представление user_defined_types показывает все составные типы, определённые в текущей базе данных. Показывает только те типы, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица A.54. Столбцы user_defined_types

Тип столбца	**Описание**
user_defined_type_catalog sql_identifier	Имя базы данных, содержащей тип (всегда текущая база)
user_defined_type_schema sql_identifier	Имя схемы, содержащей тип
user_defined_type_name sql_identifier	Имя типа
user_defined_type_category character_data	Всегда STRUCTURED

A.55. user_mapping_options

Представление user_mapping_options показывает все параметры, заданные для сопоставлений пользователей в текущей базе данных. В нём отражаются только сопоставления пользователей, установленные для сторонних серверов, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица A.55. Столбцы user_mapping_options

Тип столбца	Описание
authorization_identifier sql_identifier	Имя сопоставляемого пользователя, либо PUBLIC, если это сопоставление для всех
foreign_server_catalog sql_identifier	Имя базы данных, в которой определён сторонний сервер, задействованный в сопоставлении (всегда текущая база)
foreign_server_name sql_identifier	Имя стороннего сервера, задействованного в сопоставлении
option_name sql_identifier	Имя параметра
option_value character_data	Значение параметра. Этот столбец будет содержать не NULL, только если описывается сопоставление текущего пользователя, либо это сопоставление для PUBLIC, а текущий пользователь — владелец стороннего сервера или суперпользователь. Данное ограничение введено для защиты информации о пароле, сохранённой в параметрах сопоставления.

A.56. user_mappings

Представление user_mappings показывает все сопоставления пользователей, определённые в текущей базе данных. В нём показываются только сопоставления, установленные для сторонних серверов, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица A.56. Столбцы user_mappings

Тип столбца	Описание
authorization_identifier sql_identifier	Имя сопоставляемого пользователя, либо PUBLIC, если это сопоставление для всех
foreign_server_catalog sql_identifier	Имя базы данных, в которой определён сторонний сервер, задействованный в сопоставлении (всегда текущая база)
foreign_server_name sql_identifier	Имя стороннего сервера, задействованного в сопоставлении

A.57. view_column_usage

Представление view_column_usage описывает все столбцы, задействованные в выражении запроса представления (операторе SELECT, определяющем представление). Столбец выводится в этом списке, только если содержащая его таблица принадлежит текущей активной роли.

Таблица A.57. Столбцы view_column_usage

Тип столбца	Описание
view_catalog sql_identifier	Имя базы данных, содержащей представление (всегда текущая база)
view_schema sql_identifier	Имя схемы, содержащей представление
view_name sql_identifier	Имя представления

Тип столбца	Описание
table_catalog sql_identifier	Имя базы данных, содержащей таблицу со столбцом, задействованным в представлении (всегда текущая база)
table_schema sql_identifier	Имя схемы, содержащей таблицу со столбцом, задействованным в представлении
table_name sql_identifier	Имя таблицы со столбцом, задействованным в представлении
column_name sql_identifier	Имя столбца, задействованного в представлении

A.58. view_routine_usage

Представление view_routine_usage описывает все подпрограммы (функции и процедуры), используемые в выражении запроса представления (операторе SELECT, определяющем представление). Подпрограмма выводится в этом списке, только если она принадлежит текущей активной роли.

Таблица A.58. Столбцы view_routine_usage

Тип столбца	Описание
table_catalog sql_identifier	Имя базы данных, содержащей представление (всегда текущая база)
table_schema sql_identifier	Имя схемы, содержащей представление
table_name sql_identifier	Имя представления
specific_catalog sql_identifier	Имя базы данных, содержащей функцию (всегда текущая база)
specific_schema sql_identifier	Имя схемы, содержащей функцию
specific_name sql_identifier	«Однозначное имя» функции

A.59. view_table_usage

Представление view_table_usage описывает все таблицы, задействованные в выражении запроса представления (операторе SELECT, определяющем представление). В этом представлении показываются только таблицы, принадлежащие текущей активной роли.

Таблица A.59. Столбцы view_table_usage

Тип столбца	Описание
view_catalog sql_identifier	Имя базы данных, содержащей представление (всегда текущая база)
view_schema sql_identifier	Имя схемы, содержащей представление
view_name sql_identifier	Имя представления
table_catalog sql_identifier	Имя базы данных, содержащей таблицу, задействованную в представлении (всегда текущая база)
table_schema sql_identifier	Имя схемы, содержащей таблицу, задействованную в представлении
table_name sql_identifier	Имя таблицы, задействованной в представлении

A.60. views

Представление views показывает все представления, определённые в текущей базе данных. В нём показываются только представления, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица A.60. Столбцы views

Тип столбца	**Описание**
table_catalog sql_identifier	Имя базы данных, содержащей представление (всегда текущая база)
table_schema sql_identifier	Имя схемы, содержащей представление
table_name sql_identifier	Имя представления
view_definition character_data	Выражение запроса, определяющее представление (NULL, если представление не принадлежит текущей активной роли)
check_option character_data	CASCADED или LOCAL, если для представления определена характеристика CHECK OPTION, и NONE в противном случае
is_updatable yes_or_no	YES, если представление допускает изменение (команды UPDATE и DELETE), или NO в противном случае
is_insertable_into yes_or_no	YES, если представление допускает добавление данных (команду INSERT), или NO в противном случае
is_trigger_updatable yes_or_no	YES, если для представления определён триггер INSTEAD OF UPDATE, или NO в противном случае
is_trigger_deletable yes_or_no	YES, если для представления определён триггер INSTEAD OF DELETE, или NO в противном случае
is_trigger_insertable_into yes_or_no	YES, если для представления определён триггер INSTEAD OF INSERT, или NO в противном случае

Лист регистрации изменений									
Изм.	Номера листов (страниц)				Всего листов (страниц) в докум.	№ докум.	Входящий № сопроводительного докум. и дата	Подп.	Дата
	измененных	замененных	новых	аннулированных					