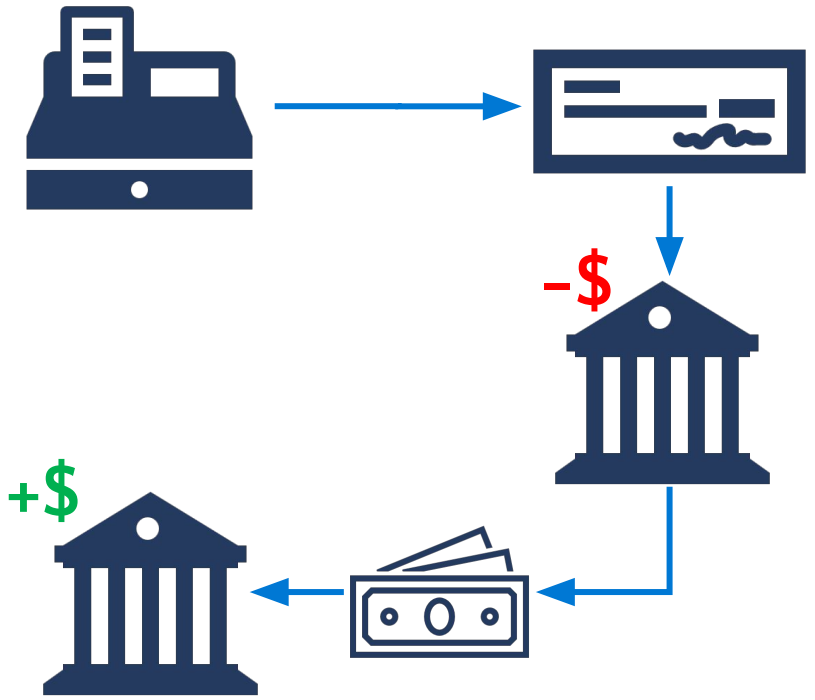
День 2

Транзакции

- Набор команд выполняемый по принципу все или ничего
- ACID
- Изоляция транзакций

ACID OLTP

- **Atomicity** – атомарность
- **Consistency** – согласованность
- **Isolation** – изоляция транзакций друг от друга
- **Durability** – сохранение результата на диске



Управление транзакцией

Открытие

- BEGIN TRANSACTION;

Или

- BEGIN WORK;

Или

- BEGIN;

Закрытие

- COMMIT TRANSACTION; COMMIT WORK; COMMIT;

Отмена

- ROLLBACK TRANSACTION; ROLLBACK WORK; ROLLBACK;

Transaction isolation level

Уровень изоляции	«Грязное» чтение	Неповторяемое чтение	Фантомное чтение	Аномалия сериализации
Read uncommited (Чтение незафиксированных данных)	Допускается, но не в PG	Возможно	Возможно	Возможно
Read committed (Чтение зафиксированных данных)	Невозможно	Возможно	Возможно	Возможно
Repeatable read (Повторяемое чтение)	Невозможно	Невозможно	Допускается, но не в PG	Возможно
Serializable (Сериализуемость)	Невозможно	Невозможно	Невозможно	Невозможно

Стандарт SQL уровней изоляции транзакций

Стандарт SQL определяет четыре уровня изоляции транзакций.

«грязное» чтение

Транзакция читает данные, записанные параллельной незавершённой транзакцией.

неповторяемое чтение

Транзакция повторно читает те же данные, что и раньше, и обнаруживает, что они были изменены другой транзакцией (которая завершилась после первого чтения).

фантомное чтение

Транзакция повторно выполняет запрос, возвращающий набор строк для некоторого условия, и обнаруживает, что набор строк, удовлетворяющих условию, изменился из-за транзакции, завершившейся за это время.

аномалия сериализации

Результат успешной фиксации группы транзакций оказывается несогласованным при всевозможных вариантах исполнения этих транзакций по очереди.

MVCC

Реализация транзакций в СУБД PostgreSQL основана на многоверсионной модели (Multiversion Concurrency Control, MVCC).
Эта модель предполагает, что каждый SQL-оператор видит так называемый **снимок данных (snapshot),** т. е. то согласованное состояние (версию) базы данных, которое она имела на определенный момент времени.
При этом параллельно исполняемые транзакции, даже вносящие изменения в базу данных, не нарушают согласованности данных этого снимка.
Такой результат в PostgreSQL достигается за счет того, что когда параллельные транзакции изменяют одни и те же строки таблиц, тогда **создаются отдельные версии** этих строк, доступные соответствующим транзакциям. Это позволяет ускорить работу с базой данных, однако требует больше дискового пространства и оперативной памяти.
И еще одно важное следствие применения MVCC — **операции чтения никогда не блокируются операциями записи, а операции записи никогда не блокируются операциями чтения**.

READ COMMITTED

Read Committed — уровень изоляции транзакции, выбираемый в PostgreSQL по умолчанию. В транзакции, работающей на этом уровне, запрос SELECT (без предложения FOR UPDATE/SHARE) видит только те данные, которые были зафиксированы до начала запроса; он никогда не увидит незафиксированных данных или изменений, внесённых в процессе выполнения запроса параллельными транзакциями.

По сути запрос SELECT видит снимок базы данных в момент начала выполнения запроса. Однако SELECT видит результаты изменений, внесённых ранее в этой же транзакции, даже если они ещё не зафиксированы. Также заметьте, что два последовательных оператора SELECT могут видеть разные данные даже в рамках одной транзакции, если какие-то другие транзакции зафиксируют изменения после запуска первого SELECT, но до запуска второго.

REPEATABLE READ

В режиме Repeatable Read видны только те данные, которые были зафиксированы до начала транзакции, но не видны незафиксированные данные и изменения, произведённые другими транзакциями в процессе выполнения данной транзакции. (Однако запрос будет видеть эффекты предыдущих изменений в своей транзакции, несмотря на то, что они не зафиксированы.) Это самое строгое требование, которое стандарт SQL вводит для этого уровня изоляции, и при его выполнении предотвращаются все явления, описанные в Таблице 13.1, за исключением аномалий сериализации. Как было сказано выше, это не противоречит стандарту, так как он определяет только минимальную защиту, которая должна обеспечиваться на каждом уровне изоляции.

Этот уровень отличается от Read Committed тем, что запрос в транзакции данного уровня видит снимок данных на момент начала первого оператора в транзакции (не считая команд управления транзакциями), а не начала текущего оператора. Таким образом, последовательные команды SELECT в одной транзакции видят одни и те же данные; они не видят изменений, внесённых и зафиксированных другими транзакциями после начала их текущей транзакции.

Приложения, использующие этот уровень, должны быть готовы повторить транзакции в случае сбоев сериализации.

SERIALIZABLE

Уровень Serializable обеспечивает самую строгую изоляцию транзакций. На этом уровне моделируется последовательное выполнение всех зафиксированных транзакций, как если бы транзакции выполнялись одна за другой, последовательно, а не параллельно. Однако, как и на уровне Repeatable Read, на этом уровне приложения должны быть готовы повторять транзакции из-за сбоев сериализации. Фактически этот режим изоляции работает так же, как и Repeatable Read, только он дополнительно отслеживает условия, при которых результат параллельно выполняемых сериализуемых транзакций может не согласовываться с результатом этих же транзакций, выполняемых по очереди. Это отслеживание не привносит дополнительных препятствий для выполнения, кроме тех, что присущи режиму Repeatable Read, но тем не менее создаёт некоторую добавочную нагрузку, а при выявлении исключительных условий регистрируется аномалия сериализации и происходит сбой сериализации.

https://postgrespro.ru/docs/postgresql/15/transaction-iso

AUTOCOMMIT - сервер

- Сервер работает ВСЕГДА в режиме AUTOCOMMIT
- Изменить это на стороне сервера нельзя
- В версии 7.3 ввели параметр autocommit, но в версии 7.4 от него отказались

AUTOCOMMIT - клиент

Можно выключить на клиенте

- psql “\set AUTOCOMMIT off”
- JDBC, call java.sql.Connection.setAutoCommit(boolean)
- psycopg2, call connection.set_session(autocommit=True)
- pgAdmin 4, нажать стрелку “down”
- DBeaver autocommit icon в SQL editor для отключения

В этом режиме psql перед первой командой после завершенной транзакции неявно (за сценой) выполняет команду BEGIN;

ON_ERROR_ROLLBACK в psql

- \set
- ON_ERROR_ROLLBACK = 'off’
  - В этом режиме любая ошибка в транзакции приводит к ее отмене
    - ERROR: current transaction is aborted, commands ignored until end of transaction block
- ON_ERROR_ROLLBACK = ‘on’
  - В этом режиме psql создает savepoint перед каждой вводимой командой и при ошибке откатывается только до этого savepoint
  - Если ошибки нет psql выполняет RELEASE savepoint (за сценой)

Ввод пользователя

- BEGIN; ## entered by the user
- INSERT INTO somi VALUES ('Mr. Roboto', '3gNc841Rmy+a', 'Triton');
- INSERT INTO somi VALUES ('A Mountain We Will Climb', 'O2DMZfqnfj8Tle', 'Tethys');
- INSERT INTO somi BALUES ('Samba de Janeiro', 'W2rQpGU0MfIrm', 'Dione’);

Что отправляет psql

- BEGIN; ## entered by the user
- SAVEPOINT myapp_temporary_savepoint ## entered by the application
- INSERT INTO somi VALUES ('Mr. Roboto', '3gNc841Rmy+a', 'Triton');
- RELEASE myapp_temporary_savepoint

- SAVEPOINT myapp_temporary_savepoint
- INSERT INTO somi VALUES ('A Mountain We Will Climb', 'O2DMZfqnfj8Tle', 'Tethys');
- RELEASE myapp_temporary_savepoint

- SAVEPOINT myapp_temporary_savepoint
- INSERT INTO somi BALUES ('Samba de Janeiro', 'W2rQpGU0MfIrm', 'Dione');
- ROLLBACK TO myapp_temporary_savepoint

Особенности транзакций

- DDL команды откатываются
  - Кроме DROP DATABASE, CREATE TABLESPACE, DROP TABLESPACE и еще нескольких команд
- Нет автономных транзакций
- Точки сохранения – savepoint
  - После возврата к точке сохранения транзакция может продолжаться
  - Количество точек сохранения практически неограничено
- Управление транзакцией внутри процедуры
- Нельзя вызвать процедуру, содержащую транзакцию, во внешней явной транзакции или при AUTOCOMMIT off – это вызовет ошибку

Блокировка таблицы

- \h lock
- Команда: LOCK
- Описание: заблокировать таблицу
- Синтаксис:
- LOCK [TABLE] [ONLY] имя [*] [, ...] [IN режим_блокировки MODE] [NOWAIT]

- где допустимый режим_блокировки:

- ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
- | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE

- https://www.postgresql.org/docs/13/sql-lock.html

Блокировки на уровне строк

Запрашиваемый режим блокировки	Текущий режим FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

FOR UPDATE

```
BEGIN;
SELECT * FROM invoice WHERE processed = false FOR UPDATE;
** здесь приложение что-то делает **
UPDATE invoice SET processed = true ...
COMMIT;
```

- SELECT FOR UPDATE блокирует строки так же, как это сделала бы команда UPDATE.
- конкурентно произвести изменения нельзя.
- После COMMIT все блокировки освобождаются.

SELECT FOR UPDATE NOWAIT

Если одна команда `SELECT FOR UPDATE` ждет завершения другой такой же команды, то она будет «висеть», пока первая начатая команда не завершится (в результате `COMMIT` или `ROLLBACK`).

Если первая транзакция по какой-то причине не
хочет завершаться, то вторая будет ждать бесконечно.

Чтобы предотвратить
такое развитие ситуации, можно воспользоваться командой `SELECT FOR UPDATE NOWAIT`.
Или

```
SET lock_timeout TO 5000;
```

SELECT FOR UPDATE SKIP LOCKED

- SELECT ... FROM flight LIMIT 1 FOR UPDATE;
  - Блокирует всю таблицу, конкурентные запросы ждут
- SELECT * FROM t_flight LIMIT 2 FOR UPDATE SKIP LOCKED;
  - Выдает свободные записи, пропуская заблокированные

Deadlock

ОШИБКА: обнаружена взаимоблокировка

ПОДРОБНОСТИ: Процесс 91521 ожидает в режиме ShareLock блокировку "транзакция 903";

заблокирован процессом 77185.

Процесс 77185 ожидает в режиме ShareLock блокировку "транзакция 905";

заблокирован процессом 91521.

ПОДСКАЗКА: подробности запроса смотрите в протоколе сервера.

КОНТЕКСТ: при изменении кортежа (0,1) в отношении "t_deadlock"

Select for …

- SELECT FOR UPDATE
  - В режиме FOR UPDATE строки, выданные оператором SELECT, блокируются как для изменения. При этом они защищаются от блокировки, изменения и удаления другими транзакциями до завершения текущей.
- SELECT FOR NO KEY UPDATE
  - Действует подобно FOR UPDATE, но запрашиваемая в этом режиме блокировка слабее: она не будет блокировать команды SELECT FOR KEY SHARE, пытающиеся получить блокировку тех же строк.
- SELECT FOR SHARE
  - Действует подобно FOR NO KEY UPDATE, за исключением того, что для каждой из полученных строк запрашивается разделяемая, а не исключительная блокировка.
- SELECT FOR KEY SHARE
  - Действует подобно FOR SHARE, но устанавливает более слабую блокировку: блокируется SELECT FOR UPDATE, но не SELECT FOR NO KEY UPDATE. Блокировка разделяемого ключа не позволяет другим транзакциям выполнять команды DELETE и UPDATE, только если они меняют значение ключа (но не другие UPDATE), и при этом допускает выполнение команд SELECT FOR NO KEY UPDATE, SELECT FOR SHARE и SELECT FOR KEY SHARE.

Insert

INSERT INTO table_name(column1, column2, …)VALUES (value1, value2, …);

INSERT INTO table_name(column1, column2, …)VALUES (value1, value2, …)

RETURNING *;

Добавление нескольких строк одной командой

```
INSERT INTO table_name (column_list)
VALUES    (value_list_1),    (value_list_2),    ...    (value_list_n);

INSERT INTO table_name (column_list)
VALUES    (value_list_1),    (value_list_2),    ...    (value_list_n)
RETURNING * | output_expression;
```

Update

UPDATE table_name SET column1 = value1, column2 = value2, ...WHERE condition;

UPDATE table_name SET column1 = value1, column2 = value2, ...WHERE condition

RETURNING * | output_expression AS output_name;

UPDATE t1 SET t1.c1 = new_value

FROM t2

WHERE t1.c2 = t2.c2;

Delete

DELETE FROM table_name WHERE condition;

DELETE FROM table_name WHERE condition

RETURNING (select_list | *)

DELETE FROM table_name1 USING table_expression WHERE condition
RETURNING returning_columns;

Upsert

Или добавляет или обновляет (или не обновляет) запись – merge

INSERT INTO table_name(column_list) VALUES(value_list)
ON CONFLICT target action;
target:
    (column_name)
    ON CONSTRAINT constraint_name – имя UNIQUE constraint.
    WHERE

action:

    DO NOTHING – ничего не делать с существующей записью в таблице
    DO UPDATE SET column_1 = value_1, .. WHERE condition – обновить поля

Truncate Table

Быстрая очистка всей таблицы

TRUNCATE TABLE invoices;

TRUNCATE TABLE invoices RESTART IDENTITY;

TRUNCATE TABLE invoices, customers; -- очистка нескольких таблиц одной командой

TRUNCATE TABLE invoices CASCADE; --FK

Может быть в теле транзакции, откатывается

CTE

```
WITH cte_name (column_list)
AS (    CTE_query_definition )
statement;
```

- Бывает удобно для сложных запросов
- Рекурсия
- Вместе с оконными фукнциями

Пример рекурсии CTE

```
WITH RECURSIVE cte_name AS(    CTE_query_definition  --nonrecursiveterm
UNION [ALL]
CTE_query definion  -- recursive term)
SELECT * FROM cte_name;
```

SQL курсоры

- Declare – открытие курсора
- Fetch – получение нужной записи
- Move – перемещение на указанную запись без ее выдачи
- Close – закрытие
- Курсор требует транзакции, кроме варианта открытия с with hold – иначе он живет только одну команду и смысла не имеет
- https://postgrespro.ru/docs/postgresql/15/sql-declare
- PL/pgsql использует свои курсоры с более широкими возможностями

Встроенные функции

- Агрегатные
- Оконные
- Дата время
- Строковые
- Математические
- Прочие…

Введение в оконные функции

- Window functions – оконные функции
- Нужны для того, чтобы вычислить некоторый результат по выборке вертикально по колонкам и расположить его горизонтально как отдельную колонку для каждой записи выборки
- Пример – расчет нарастающего итога в бухгалтерии
- Используют специальные функции, похожие на агрегатные (например, оконная функция SUM)
- Также к оконным функция относятся функции ранжировки и смещения

Что дает оконная фукнция?

```
SELECT city, SUM(order_amount) total_order_amount
FROM [dbo].[Orders] GROUP BY city
```

Results | Messages

	city	total_order_amount
1	Arington	37000.00
2	GuildFord	50500.00
3	Shalford	13000.00

```
SELECT order_id, order_date, customer_name, city, order_amount
 ,SUM(order_amount) OVER(PARTITION BY city) as grand_total
FROM [dbo].[Orders]
```

Results | Messages

	order_id	order_date	customer_name	city	order_amount	grand_total
1	1002	2017-04-02	David Jones	Arington	20000.00	37000.00
2	1007	2017-04-10	Andrew Smith	Arington	15000.00	37000.00
3	1008	2017-04-11	David Brown	Arington	2000.00	37000.00
4	1001	2017-04-01	David Smith	GuildFord	10000.00	50500.00
5	1006	2017-04-06	Paum Smith	GuildFord	25000.00	50500.00
6	1004	2017-04-04	Michael Smith	GuildFord	15000.00	50500.00
7	1010	2017-04-25	Peter Smith	GuildFord	500.00	50500.00
8	1005	2017-04-05	David Williams	Shalford	7000.00	13000.00
9	1003	2017-04-03	John Smith	Shalford	5000.00	13000.00
10	1009	2017-04-20	Robert Smith	Shalford	1000.00	13000.00

Window function

window_function(arg1, arg2,..)

OVER ([PARTITION BY partition_expression]

[ORDER BY sort_expression [ASC | DESC] [NULLS {FIRST | LAST }])

Пользовательские функции

В PostgreSQL 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.

Проще всего определить функции на языке SQL, поэтому мы рассмотрим их.

SQL функции

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. (Помните, что 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.

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.

Пример SQL функции

```
CREATE FUNCTION one() RETURNS integer AS $$
    SELECT 1 AS result;
$$ LANGUAGE SQL;
```

- Двойной символ доллара для обозначения строки
- LANGUAGE указывает язык функции
- Строка интерпретируется при каждом вызове функции

Параметры в функциях

```
CREATE FUNCTION hello(name text) -- формальный параметр
RETURNS text AS $$
SELECT 'Hello, ' || name || '!';
$$ LANGUAGE sql;

--при вызове передаем параметр нужного типа
SELECT hello('Student’);
```

Особенности параметров

CREATE FUNCTION hello(text) – параметр только по типу, без имени
RETURNS text AS $$
SELECT 'Hello, ' || $1 || '!'; -- номер вместо имени
$$ LANGUAGE sql;

CREATE FUNCTION hello(IN name text, IN title text DEFAULT 'Mr’) -- второй параметр задан дефолтом, его можно пропустить при вызове функции

RETURNS text AS $$
SELECT 'Hello, ' || title || ' ' || name || '!';
$$ LANGUAGE sql;

Порядок передачи параметров

- По порядку как заданы в теле функции
- По имени
  - SELECT hello(title => 'Mr', name => 'Bond’);
- Комбинированно
  - SELECT hello(‘Bond', title => 'Mr');

Функции SQL на базовых типах

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT balance FROM bank WHERE accountno = tf1.accountno;
$$ LANGUAGE SQL;
```

или то же самое одной командой

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno
    RETURNING balance;
$$ LANGUAGE SQL;
```

Функции SQL на составных типах

- В функциях с аргументами составных типов мы должны указывать не только, какой аргумент, но и какой атрибут (поле) этого аргумента нам нужен.
- Составной тип или создан явно командой Create Type или создается неявно для каждой таблицы

Что такое составной тип

- набор именованных атрибутов (полей)
- то же, что табличная строка, но без ограничений целостности
- Полный аналог типа structure в языках C

Создание

- явное объявление нового типа
- неявно при создании таблицы создается ОДНОИМЕННЫЙ составной тип
- неопределенный составной тип record

Использование

- скаляр
- можно сравнивать, проверять на NULL, использовать с подзапросами

таблица.столбец vs столбец(таблица)

для доступа к столбцу таблицы можно использовать не только стандартное обращение таблица.столбец, но и функциональное: столбец(таблица).

Это позволяет создавать вычисляемые поля, через функцию, принимающую на вход составной тип.

Функции SQL с выходными параметрами

Альтернативный способ описать результаты функции — определить её с выходными параметрами

```
CREATE FUNCTION add_em (IN x int, IN y int, OUT sum int)
AS 'SELECT x + y'
LANGUAGE SQL;
```

Действительная ценность выходных параметров в том, что они позволяют удобным способом определить функции, возвращающие несколько столбцов. Фактически здесь мы определили анонимный составной тип для результата функции.

Имена, назначаемые выходным параметрам, не просто декоративные, а определяют имена столбцов анонимного составного типа.

```
CREATE FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int)
AS 'SELECT x + y, x * y'
LANGUAGE SQL;
```

Выходные параметры

выходные параметры не включаются в список аргументов при вызове такой функции из SQL. Это объясняется тем, что PostgreSQL определяет сигнатуру вызова функции, рассматривая только входные параметры. Это также значит, что при таких операциях, как удаление функции, в ссылках на функцию учитываются только типы входных параметров. Таким образом, удалить эту конкретную функцию можно любой из этих команд:

```
DROP FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int);
DROP FUNCTION sum_n_product (int, int);
```

Параметры функции могут быть объявлены как IN (по умолчанию), OUT, INOUT или VARIADIC. Параметр INOUT действует как входной (является частью списка аргументов при вызове) и как выходной (часть типа записи результата). Параметры VARIADIC являются входными, но обрабатывается специальным образом

Функции SQL с переменным количеством параметров

- VARIADIC
- Функции SQL могут быть объявлены как принимающие переменное число аргументов, с условием, что все «необязательные» аргументы имеют один тип данных. Необязательные аргументы будут переданы такой функции в виде массива. Для этого в объявлении функции последний параметр помечается как VARIADIC; при этом он должен иметь тип массива.

Функции SQL со значением параметров по умолчанию

Функции могут быть объявлены со значениями по умолчанию для некоторых или всех входных аргументов. Значения по умолчанию подставляются, когда функция вызывается с недостаточным количеством фактических аргументов. Так как аргументы можно опускать только с конца списка фактических аргументов, все параметры после параметра со значением по умолчанию также получат значения по умолчанию.

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL
AS $$
    SELECT $1 + $2 + $3;
$$;
```

Функции SQL в качестве табличных данных

Все функции SQL можно использовать в предложении FROM запросов, но наиболее полезно это для функций, возвращающих составные типы. Если функция объявлена как возвращающая базовый тип, она возвращает таблицу с одним столбцом. Если же функция объявлена как возвращающая составной тип, она возвращает таблицу со столбцами для каждого атрибута составного типа.

Возвраается только одна строка (функция определена без SETOF)

```
SELECT *, upper(fooname) FROM getfoo(1) AS t1;

 fooid | foosubid | fooname | upper
-------+----------+---------+-------
     1 |        1 | Joe     | JOE
(1 row)
```

Функции SQL, возвращающие набор строк

Когда SQL-функция объявляется как возвращающая SETOF некий_тип, конечный запрос функции выполняется до завершения и каждая строка выводится как элемент результирующего множества.

Это обычно используется, когда функция вызывается в предложении FROM. В этом случае каждая строка, возвращаемая функцией, становится строкой таблицы, появляющейся в запросе.

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;
```

RETURNS SETOF record

Также возможно выдать несколько строк со столбцами, определяемыми выходными параметрами

Здесь ключевая особенность заключается в записи RETURNS SETOF record, показывающей, что функция возвращает множество строк вместо одной. Если существует только один выходной параметр, укажите тип этого параметра вместо record.

```
CREATE FUNCTION sum_n_product_with_tab (x int, OUT sum int, OUT product int)
RETURNS SETOF record
AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;

SELECT * FROM sum_n_product_with_tab(10);
```

Функции SQL, возвращающие таблицы (TABLE)

Есть ещё один способ объявить функцию, возвращающую множества, — использовать синтаксис RETURNS TABLE(столбцы). Это равнозначно использованию одного или нескольких параметров OUT с объявлением функции как возвращающей SETOF record (или SETOF тип единственного параметра, если это применимо). Этот синтаксис описан в последних версиях стандарта SQL, так что этот вариант может быть более портируемым, чем SETOF.

```
CREATE FUNCTION sum_n_product_with_tab (x int)
RETURNS TABLE(sum int, product int) AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;
```

Запись RETURNS TABLE не позволяет явно указывать OUT и INOUT для параметров — все выходные столбцы необходимо записать в списке TABLE.

Полиморфные функции SQL

Функции SQL могут быть объявлены как принимающие и возвращающие полиморфные типы

```
CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;
```

```
intarray | textarray
----------+-----------
 {1,2}    | {a,b}
(1 row)
```

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 (см. Раздел 8.7)
anyrange	Простое	Указывает, что функция принимает любой диапазонный тип (см. Раздел 8.17)
anymultirange	Простое	Указывает, что функция принимает любой мультидиапазонный тип (см. Раздел 8.17)
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
anycompatiblemultirange	Общее	Указывает, что функция принимает любой мультидиапазонный тип данных и может автоматически приводить различные аргументы к общему типу данных

Полиморфизм выходных аргументов

Для результата типа anyelement требуется минимум один аргумент типа anyelement, anyarray, anynonarray, anyenum или anyrange

```
CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE SQL;

SELECT * FROM dup(22);
 f2 |   f3
----+---------
 22 | {22,22}
(1 row)
```

Перегрузка функций

Вы можете определить несколько функций с одним именем SQL, если эти функции будут принимать разные аргументы. Другими словами, имена функций можно перегружать

- сигнатура функции — ее имя и типы входных параметров
- подпрограммы различаются типами входных параметров;
- тип возвращаемого значения и выходные параметры игнорируются
- подходящая подпрограмма выбирается во время выполнения в зависимости от фактически заданных параметров при вызове

Команда CREATE OR REPLACE

- при несовпадении типов входных параметров создаст новую перегруженную функцию
- при совпадении — изменит существующую, но поменять возвращаемый тип и типы выходных параметров нельзя

Волатильность 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

VOLATILE

- 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.

STABLE

- 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.

IMMUTABLE

- 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.

Волатильность и оптимизатор

Для наилучших результатов оптимизации, функции следует назначать самую строгую характеристику изменчивости, которой она соответствует.

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.

Волатильность и MVCC – видимость изменений

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.

Рекомендации по волатильности

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.

PostgreSQL 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. Однако PostgreSQL не принуждает вас явно отказаться от этого.