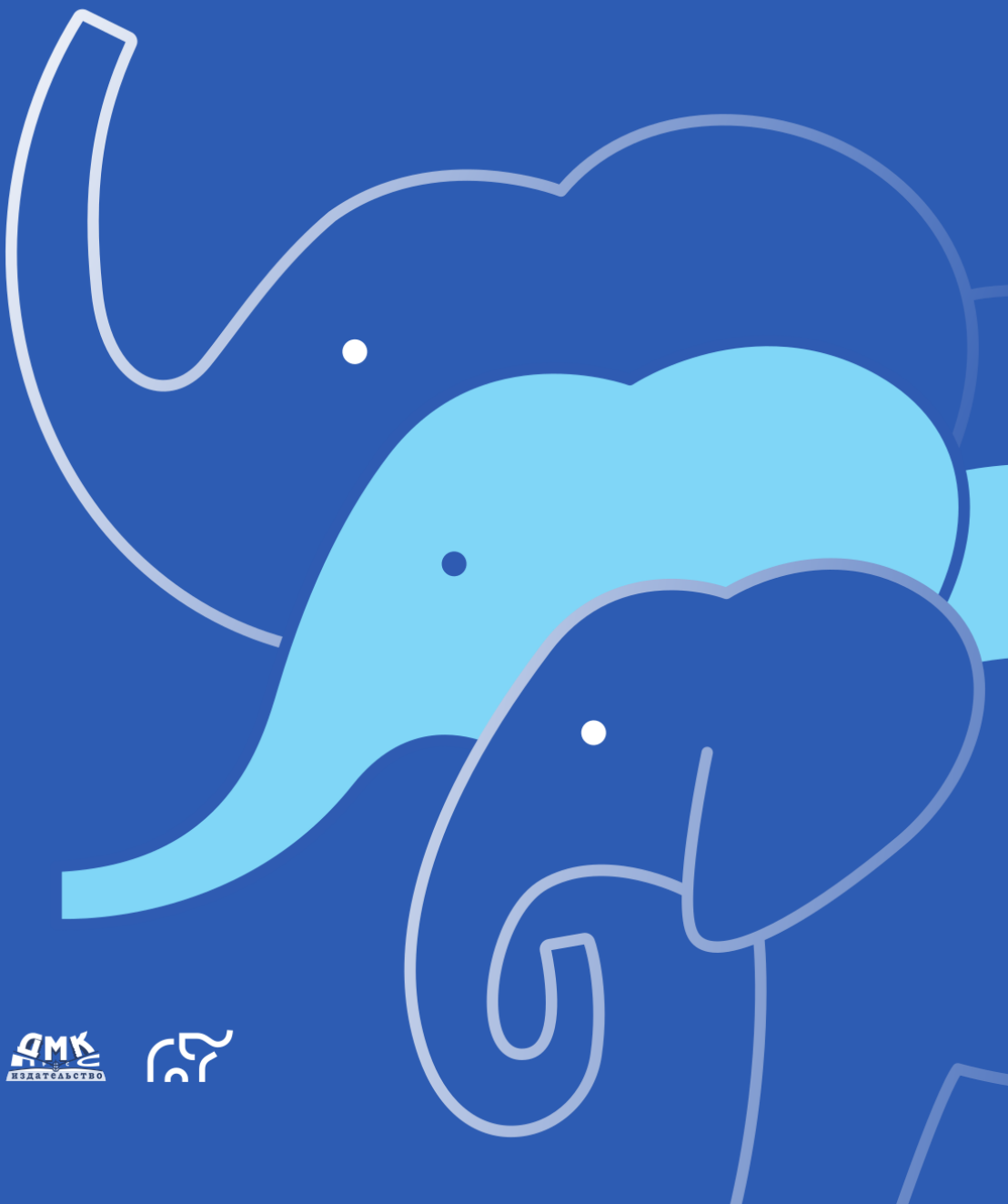
Евгений Моргунов

PostgreSQL. Профессиональный SQL



ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Компания Postgres Professional

Е. П. Моргунов

PostgreSQL. Профессиональный SQL

Учебное пособие

ДМК ПРЕСС
издательство
Москва, 2025

УДК 004.65
ББК 32.972.134
М79

Моргунов, Е. П.

М79 PostgreSQL. Профессиональный SQL : учеб. пособие / Е. П. Моргунов; под ред. Е. В. Рогова. — М. : ДМК Пресс, 2025. — 444 с.

ISBN 978-5-93700-373-7

Учебное пособие представляет собой вторую часть курса по языку SQL, созданного при поддержке российской компании Postgres Professional. В книге рассмотрены такие расширенные возможности СУБД PostgreSQL, как общие табличные выражения, агрегатные и оконные функции, подзапросы LATERAL, создание пользовательских функций и процедур. Материал сопровождается многочисленными практическими примерами, заданиями и упражнениями, дополняющими основной текст каждой главы. Пособие заинтересует читателей, желающих повысить базовый уровень владения языком SQL. Изучать материал пособия можно как под руководством преподавателя, так и самостоятельно.

Сайт книги: https://postgrespro.ru/education/books/advancedsql.

УДК 004.65
ББК 32.972.134

ISBN 978-5-93700-373-7

Оглавление

Содержание

Глава 1
Введение

1.1. О книге

Язык SQL является общепринятым средством взаимодействия с данными. Недавно этому языку исполнилось полвека, но реляционные технологии не только не собираются сдавать своих позиций, но и продолжают развиваться и распространяться все шире. Знание SQL необходимо всем, кто имеет отношение к базам данных: и разработчикам приложений, и аналитикам, и администраторам. Его поддерживают все реляционные СУБД, и PostgreSQL не является исключением. В этой системе весьма полно реализованы требования стандарта к языку SQL и представлен целый ряд «фирменных» расширений.

Несколько лет назад при поддержке компании Postgres Professional была издана книга «PostgreSQL. Основы языка SQL»[1]. В ней рассматривались основы языка и такие вопросы, как транзакции и повышение производительности. Однако современный язык SQL выходит далеко за эти рамки, предлагая множество различных команд и возможностей, зачастую сложных для освоения, но открывающих путь к эффективной работе с данными. Поэтому и возникла идея написать продолжение базового учебника.

Вот такие темы вошли в эту книгу.

Глава 2 посвящена общим табличным выражениям. Они кратко рассматривались в первой книге. Общие табличные выражения — очень полезное средство, которое позволяет, например, в одном запросе обрабатывать иерархические данные и даже графы общего вида. В качестве иллюстраций рассмотрены структура узлов и деталей самолета, штатное расписание компании, сеть маршрутов между городами.

[1] Моргунов, Е. П. PostgreSQL. Основы языка SQL. — СПб. : БХВ-Петербург, 2018. — 336 с. — ISBN 978-5-9775-4022-3.

В **главе 3** представлены аналитические возможности PostgreSQL. Рассмотрение начинается с агрегатных функций. Наиболее употребительные из них были показаны в базовом учебнике. Продемонстрировано использование основных статистических функций для вычисления дисперсии, среднеквадратического отклонения, моды, медианы. Важное место в этой главе занимают группировки данных. Конструкции GROUPING SETS, CUBE и ROLLUP позволяют свести несколько запросов с обычными группировками к одному. Большое внимание уделено оконным функциям, в частности особенностям формирования оконного кадра в разных режимах. Завершается глава обращением к гипотезирующим агрегатным функциям, то есть функциям, работающим с гипотетическими множествами строк.

Небольшая **глава 4** демонстрирует конструкцию LATERAL команды SELECT. В отличие от обычного соединения в предложении FROM, она позволяет подзапросу обращаться к полям из текущей строки другой таблицы, фактически образуя цикл в рамках одного запроса. Показаны также вызовы стандартных функций в предложении FROM.

Глава 5 посвящена программированию на стороне сервера. Рассмотрение начинается с простых функций, возвращающих одну строку, в том числе перегруженных. Затем вводятся табличные функции, а также функции с переменным числом параметров. Показано применение модификаторов IN, OUT и INOUT. Эта глава — самая большая в книге, поскольку написание и применение функций имеет множество нюансов. Функции могут зависеть от других объектов базы данных, категория изменчивости влияет на решения планировщика при оптимизации запросов и на видимость изменений, при выполнении ряда условий код функции может быть подставлен в запрос, а еще функции могут участвовать в параллельном выполнении. В качестве иллюстрации применения функций для поддержки принятия решений показан выбор наилучших альтернатив с помощью формирования множества Парето. Основной материал главы посвящен функциям, но в завершающем разделе обсуждаются и процедуры, которые во многом схожи с функциями, но имеют целый ряд отличий.

Книгу можно использовать как в процессе изучения технологий баз данных в образовательном учреждении (в бакалавриате и магистратуре), так и для самостоятельной подготовки.

Большое внимание в учебнике уделяется заданиям и упражнениям. Они различаются по объему и сложности, а также имеют разную специфику: многие предполагают написание SQL-запроса, однако есть и такие, которые заключаются в проведении экспериментов и изучении взаимосвязей между объектами базы данных. Упражнения зачастую дополняют и развивают основной текст главы, в них обсуждаются различные тонкости, частные случаи, полезные приемы и поучительные ситуации. Упражнения в конце каждой главы расположены в основном в соответствии с порядком изложения материала. Но есть и комплексные упражнения, выполнение которых требует освоения нескольких разделов или даже всей главы в целом.

Многие положения книги подкрепляются ссылками на соответствующие разделы документации, ведь именно она — первоисточник знаний. Эти разделы стоит просматривать, поскольку книга не является заменой документации и не пытается ее пересказывать.

Для учебных заведений можно предложить следующее распределение времени, исходя из того, что на изучение материала пособия будет отведено 36 академических часов:

Глава 2. Общие табличные выражения	6 часов
Глава 3. Аналитические возможности PostgreSQL	10 часов
Глава 4. Конструкция LATERAL команды SELECT	4 часа
Глава 5. Подпрограммы	16 часов

Для изучения материала учебника необходимо установить СУБД PostgreSQL (или Postgres Pro Standard) и развернуть учебную базу данных «Авиаперевозки». Скачать PostgreSQL можно с официального сайта postgresql.org/download, на котором приведена и инструкция по установке. Для использования Postgres Pro Standard в учебных заведениях нужно получить академическую лицензию, которую компания Postgres Professional предоставляет бесплатно[1]. Учебная база данных и ее полное описание доступны на сайте компании: postgrespro.ru/education/demodb. Необходима версия от 15.08.2017, вариант small.

На странице книги (postgrespro.ru/education/books/advancedsql) предлагается виртуальная машина, в которой уже установлены СУБД и учебная база данных.

[1] Для этого требуется отправить заявку на academy@postgrespro.ru.

Все SQL-запросы, приведенные в книге, были выполнены в среде PostgreSQL версии 17. Ссылки на разделы документации относятся именно к этой версии.

Большинство запросов, скорее всего, будут корректно выполняться и в более новых версиях PostgreSQL. В старых версиях результаты могут отличаться от приведенных в книге. Например, в книге используется появившийся в версии 17 перегруженный вариант функции `random`, позволяющий задать диапазон случайных значений.

Ряд запросов, представленных в книге, содержат столбцы типа `timestamptz`. Такие запросы могут возвращать результаты, отличающиеся от приведенных в книге для часового пояса Asia/Krasnoyarsk (+07), поскольку зависят от настройки часового пояса на компьютере читателя.

Команды, вводимые пользователем как в среде операционной системы, так и в среде утилиты psql, выделяются в пособии полужирным моноширинным шрифтом, а результаты их выполнения — светлым. Например:

```
postgres$ psql -d demo -U postgres
```

или

```
SELECT min( days_of_week ), max( days_of_week )
FROM routes;
 min | max
-----+-----
 {1} | {7}
(1 строка)
```

1.2. Учебная база данных

В качестве учебной базы данных используется база данных «Авиаперевозки», разработанная в компании Postgres Professional. Она представлена в трех вариантах: малом (данные о полетах за месяц), среднем (полеты за квартал) и большом (за год). Эта база данных имеет целый ряд привлекательных качеств:

- данные в ней правдоподобны (например, частоты фамилий и имен пассажиров аналогичны тем, что имеют место в России);

- данных много (даже в малом варианте базы данных есть таблицы, содержащие сотни тысяч записей, а в одной из них — больше миллиона);
- база не слишком простая и не слишком сложная (число сущностей не превышает десятка);
- проиллюстрированы разные приемы проектирования (связь «один к одному», суррогатный ключ, контролируемая избыточность и др.);
- используются разнообразные типы данных (числа, строки, даты и время, интервалы, массивы, JSON).

Легенда такова. Некая авиакомпания совершает пассажирские авиаперевозки по России. Все модели авиапарка компании собраны в представлении «Самолеты» (`aircrafts`), созданном на основе таблицы `aircrafts_data`. Коды и названия моделей соответствуют реальной практике. В таблице «Места» (`seats`) записаны схемы салонов, одинаковые для всех самолетов одной модели.

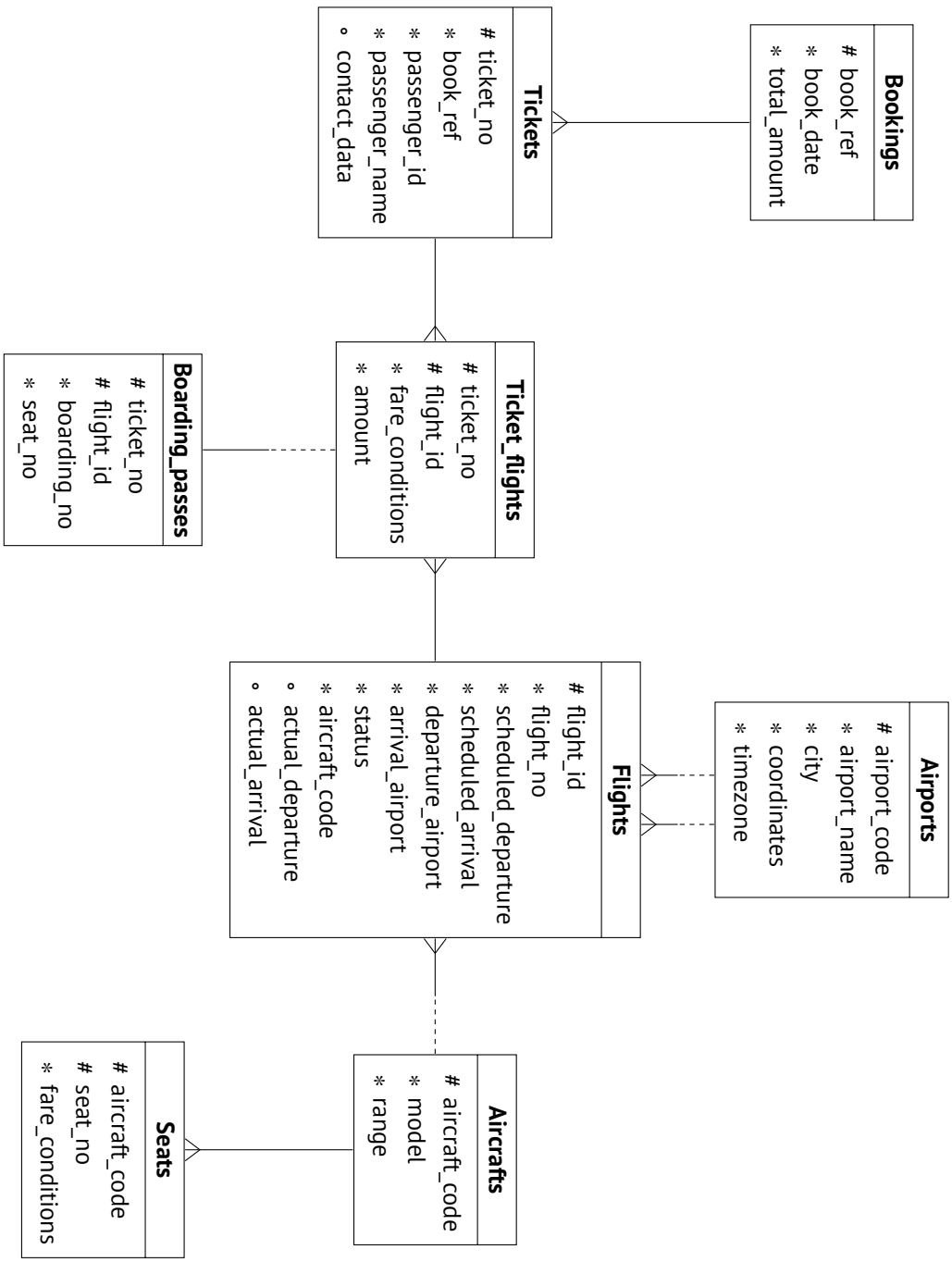
Представление «Аэропорты» (`airports`) на основе таблицы `airports_data` показывает область присутствия авиакомпании. Используются коды аэропортов, присваиваемые международной ассоциацией ИАТА. Расписание рейсов содержится в таблице «Рейсы» (`flights`). Ее первичным ключом является столбец «Идентификатор рейса» (`flight_id`). Это суррогатный ключ. Представление «Рейсы» (`flights_v`), созданное на основе этой и ряда других таблиц, выдает гораздо больше информации, например показывает местное время вылета и прибытия рейсов в соответствующих аэропортах.

Маршруты, то есть пары аэропортов, между которыми авиакомпания выполняет рейсы, можно увидеть в представлении «Маршруты» (`routes`).

Пассажиры бронируют билеты. Эти операции отражаются в таблице «Бронирования» (`bookings`). Столбец «Номер бронирования» (`book_ref`) содержит коды операций, состоящие из шести шестнадцатеричных цифр.

В каждой операции бронирования могут быть оформлены один или несколько билетов (возможно, на разных пассажиров), которые записываются в таблицу «Билеты» (`tickets`). Номера билетов представляют собой тринадцатизначные строки, состоящие из цифр.

Перелеты, входящие в оформленные билеты, записываются в таблицу «Перелеты» (`ticket_flights`). Сумма значений столбца «Стоимость перелета» (`amount`)


Bookings
book_ref
* book_date
* total_amount
Tickets
ticket_no
* book_ref
* passenger_id
* passenger_name
∘ contact_data
Ticket_flights
ticket_no
flight_id
* fare_conditions
* amount
Boarding_passes
ticket_no
flight_id
* boarding_no
* seat_no
Flights
flight_id
* flight_no
* scheduled_departure
* scheduled_arrival
* departure_airport
* arrival_airport
* status
* aircraft_code
∘ actual_departure
∘ actual_arrival
Airports
airport_code
* airport_name
* city
* coordinates
* timezone
Aircrafts
aircraft_code
* model
* range
Seats
aircraft_code
seat_no
* fare_conditions

в записях, соответствующих конкретной операции бронирования, должна быть равна значению поля «Полная сумма бронирования» (`total_amount`) в строке таблицы «Бронирования» (`bookings`), описывающей эту операцию. Это пример контролируемой избыточности.

Регистрация на рейсы отражается в таблице «Посадочные талоны» (`boarding_passes`). Она связана с таблицей «Перелеты» (`ticket_flights`) отношением «один к одному». Столбец «Статус рейса» принимает только ограниченный набор значений, например `Departed` или `Arrived`.

Все перечисленные таблицы и представления созданы в схеме `bookings`. В ней же имеется функция `now`, представляющая момент выгрузки данных, или *текущий момент*, который можно использовать в запросах вместо стандартной функции `now`.

1.3. Благодарности автора

Хотя на обложке книги указан один человек, такая работа не выполняется единолично. Многие сотрудники компании Postgres Professional помогали автору на разных этапах ее выполнения.

Огромный вклад в успешное завершение этого труда внес Егор Рогов. Он проделал большую работу в качестве редактора: предложил целый ряд структурных изменений текста книги, множество более точных формулировок ключевых понятий, улучшений SQL-запросов и модификаций упражнений, приведенных в книге. Егор выполнил верстку текста, подготовил предметный указатель, нарисовал иллюстрации, помогающие разобраться в сложных понятиях и конструкциях языка SQL.

Павел Толмачев и Павел Лузанов просматривали черновые варианты текста книги и давали полезные комментарии. Включить в книгу главу «Конструкция LATERAL команды SELECT» предложил Павел Лузанов.

В обсуждении названия книги принял участие и высказал полезные идеи Александр Мелешко.

Содержание книги оттачивалось в ходе проведения пробного курса по ее материалу со студентами Алтайского государственного технического университета

и Алтайского государственного университета. Полезные замечания высказывали Михаил Козлов, Олег Целебровский и Павел Гилев, ставшие сотрудниками компании.

И конечно, книга не состоялась бы без всесторонней поддержки и внимания со стороны руководителей компании Postgres Professional Олега Бартунова и Ивана Панченко.

Замечания и предложения можно направлять по адресу edu@postgrespro.ru.

Мы надеемся, что это учебное пособие внесет вклад в повышение уровня квалификации читателя и расширение его профессионального кругозора.

Глава 2
Общие табличные выражения

Общие табличные выражения — важное выразительное средство языка SQL, которое позволяет упростить сложный SQL-запрос, разбивая его на обозримые фрагменты, выполняющие отдельные подзадачи. В результате запрос становится более понятным, а зачастую и более эффективным. Не менее важно, что с помощью общих табличных выражений SQL-запрос может обрабатывать данные в цикле, что повышает вычислительные возможности языка SQL.

2.1. Запросы с несколькими общими табличными выражениями

В сложных SQL-запросах могут присутствовать подзапросы, в том числе вложенные. Они не только затрудняют понимание работы запроса, но и могут приводить к замедлению его выполнения. К счастью, в современном языке SQL есть средство, позволяющее успешно преодолевать эту проблему.

Для иллюстрации сказанного предположим, что отдел развития нашей авиакомпании хочет определить пассажиропоток (количество пассажиров в единицу времени), проходящий через аэропорты, в которые авиакомпания совершает полеты. От пассажиропотока зависит необходимая численность персонала и размеры платежей за использование наземных служб аэропортов.

Пассажиропоток идет в двух направлениях — вылет пассажиров и их прибытие, — поэтому для каждого аэропорта нам нужно получить общее количество как вылетевших пассажиров, так и прибывших. Для нас не имеет значения, каким рейсом вылетает пассажир; важно лишь то, откуда и куда он летит. Установить это можно, соединив таблицы «Рейсы» (`flights`) и «Посадочные талоны» (`boarding_passes`) по столбцу `flight_id`. Первая таблица содержит сведения об аэропортах вылета и прибытия каждого рейса, а результат соединения будет содержать сведения об аэропортах вылета и прибытия для каждого пассажира. Если теперь сгруппировать полученные строки по коду аэропорта отправления

или прибытия и подсчитать их, то мы решим поставленную задачу. Поскольку это две независимые группировки, придется выполнить два отдельных подзапроса, а в главном запросе соединить их результаты по условию совпадения аэропортов отправления и прибытия, поскольку для каждого аэропорта требуется получить два показателя: число пассажиров, вылетевших из него, и число пассажиров, прибывших в него.

Будем учитывать только прибывшие и вылетевшие рейсы, то есть рейсы со статусом Arrived и Departed. В качестве отчетного периода возьмем один месяц от текущего момента, который в нашей базе данных определяется функцией now из схемы bookings. Попадание рейса в заданный интервал будем определять по плановому времени вылета.

Кроме того, чтобы вывести не только код аэропорта, но и его название, воспользуемся таблицей «Аэропорты» (airports).

Получим такой запрос:

```
SELECT td.departure_airport AS "Код а/п",
       a.airport_name AS "Аэропорт",
       td.dep_pass_count AS "Вылетели",
       ta.arr_pass_count AS "Прибыли",
       td.dep_pass_count + ta.arr_pass_count AS "Всего"
FROM (
       SELECT f.departure_airport, count( * ) AS dep_pass_count
       FROM boarding_passes AS bp
         JOIN flights AS f ON f.flight_id = bp.flight_id
       WHERE f.status IN ( 'Arrived', 'Departed' )
         AND f.scheduled_departure >= bookings.now() - '1 mon'::interval
       GROUP BY f.departure_airport
     ) AS td -- total_departed_pass_counts
     JOIN
     (
       SELECT f.arrival_airport, count( * ) AS arr_pass_count
       FROM boarding_passes AS bp
         JOIN flights AS f ON f.flight_id = bp.flight_id
       WHERE f.status IN ( 'Arrived', 'Departed' )
         AND f.scheduled_departure >= bookings.now() - '1 mon'::interval
       GROUP BY f.arrival_airport
     ) AS ta -- total_arrived_pass_counts
     ON td.departure_airport = ta.arrival_airport
     JOIN airports AS a ON a.airport_code = td.departure_airport
ORDER BY "Всего" DESC;
```

Включив секундомер в утилите psql, определим время выполнения запроса. Для получения стабильного времени лучше выполнить запрос несколько раз.

```
\timing on
 Код а/п |        Аэропорт        | Вылетели | Прибыли | Всего
---------+-------------------------+----------+---------+--------
 SVO     | Шереметьево             |    93205 |   77347 | 170552
 DME     | Домодедово              |    82775 |   75767 | 158542
 LED     | Пулково                 |    35505 |   35575 |  71080
 VKO     | Внуково                 |    36694 |   29335 |  66029
 OVB     | Толмачёво               |    25137 |   27477 |  52614
 SVX     | Кольцово                |    19725 |   20628 |  40353
 AER     | Сочи                    |    16520 |   19182 |  35702
 PEE     | Пермь                   |    17006 |   18122 |  35128
 KHV     | Хабаровск-Новый         |    12130 |   12855 |  24985
 KZN     | Казань                  |     9656 |   10473 |  20129
 ...
 GDX     | Магадан                 |      154 |     253 |    407
 KXK     | Хурба                   |      237 |     158 |    395
 KLF     | Калуга                  |      186 |     168 |    354
 UCT     | Ухта                    |      122 |     169 |    291
 USK     | Усинск                  |       76 |     131 |    207
 CEE     | Череповец               |       78 |      80 |    158
 NYA     | Нягань                  |       51 |      82 |    133
 RGK     | Горно-Алтайск           |       35 |      51 |     86
 SWT     | Стрежевой               |        8 |       8 |     16
(94 строки)
Время: 360,589 мс
```

Попробуем ускорить выполнение этого запроса. В подзапросах сначала выполняется соединение таблиц «Посадочные талоны» (boarding_passes) и «Рейсы» (flights), а затем группировка. Число строк в этих таблицах различается на порядок, поэтому для уменьшения количества соединяемых строк можно заранее сгруппировать строки таблицы «Посадочные талоны» (boarding_passes) по идентификаторам рейсов (flight_id) во вложенном подзапросе. Для каждого рейса эта группировка будет содержать ровно одну строку с количеством пассажиров. После этого каждая строка таблицы «Рейсы» (flights), отобранная условием в предложении WHERE, будет соединяться только с одной строкой полученной группировки. Конечно, в подзапросах первого уровня мы должны будем заменить функцию count на sum, чтобы просуммировать количество пассажиров отдельных рейсов и получить итоговую численность вылетевших и прибывших.

```
 SELECT td.departure_airport AS "Код а/п",
       a.airport_name AS "Аэропорт",
       td.dep_pass_count AS "Вылетели",
       ta.arr_pass_count AS "Прибыли",
       td.dep_pass_count + ta.arr_pass_count AS "Всего"
FROM (
       SELECT f.departure_airport, sum( pass_count ) AS dep_pass_count
       FROM (
              SELECT flight_id, count( * ) AS pass_count
              FROM boarding_passes
              GROUP BY flight_id
            ) AS bp
           JOIN flights AS f ON f.flight_id = bp.flight_id
       WHERE f.status IN ( 'Arrived', 'Departed' )
         AND f.scheduled_departure >= bookings.now() - '1 mon'::interval
       GROUP BY f.departure_airport
     ) AS td -- total_departed_pass_counts
     JOIN
     (
       SELECT f.arrival_airport, sum( pass_count ) AS arr_pass_count
       FROM (
              SELECT flight_id, count( * ) AS pass_count
              FROM boarding_passes
              GROUP BY flight_id
            ) AS bp
           JOIN flights AS f ON f.flight_id = bp.flight_id
       WHERE f.status IN ( 'Arrived', 'Departed' )
         AND f.scheduled_departure >= bookings.now() - '1 mon'::interval
       GROUP BY f.arrival_airport
     ) AS ta -- total_arrived_pass_counts
     ON td.departure_airport = ta.arrival_airport
     JOIN airports AS a ON a.airport_code = td.departure_airport
ORDER BY "Всего" DESC;
```

Время выполнения заметно уменьшилось по сравнению с исходным запросом:

```
 ...
Время: 211,355 мс
```

Ускорение достигнуто, но за счет усложнения запроса. Однако можно и упростить его, не потеряв в скорости выполнения. Вложенный подзапрос, агрегирующий таблицу «Посадочные талоны» (boarding_passes), выполняется дважды. Если бы мы смогли сохранить результат первого выполнения этого подзапроса во временную таблицу, нам не пришлось бы выполнять то же самое действие повторно.

Как реализовать эту идею? Нам поможет предложение WITH, которое вводит в запрос *общее табличное выражение* — Common Table Expression (CTE). Как сказано в разделе документации 7.8 «Запросы WITH (Общие табличные выражения)», результат вычисления этого выражения можно рассматривать в качестве временной таблицы, существующей только во время выполнения всего запроса. Сразу добавим, что это не то же самое, что временная таблица, которая создается командой CREATE TEMP TABLE.

Перенесем вложенный подзапрос в конструкцию WITH, где он получит имя bp. В главном запросе можно ссылаться на это имя, как на имя обычной таблицы.

Вот как теперь будет выглядеть наш запрос:

```
WITH bp AS
( SELECT flight_id, count( * ) AS pass_count                    ┐
  FROM boarding_passes                                          ├ общее табличное выражение
  GROUP BY flight_id                                            ┘
)
SELECT td.departure_airport AS "Код а/п",
       a.airport_name AS "Аэропорт",
       td.dep_pass_count AS "Вылетели",
       ta.arr_pass_count AS "Прибыли",
       td.dep_pass_count + ta.arr_pass_count AS "Всего"
FROM (
       SELECT f.departure_airport, sum( pass_count ) AS dep_pass_count
       FROM bp
         JOIN flights AS f ON f.flight_id = bp.flight_id
       WHERE f.status IN ( 'Arrived', 'Departed' )
         AND f.scheduled_departure >= bookings.now() - '1 mon'::interval
       GROUP BY f.departure_airport
     ) AS td -- total_departed_pass_counts
     JOIN
     (
       SELECT f.arrival_airport, sum( pass_count ) AS arr_pass_count
       FROM bp
         JOIN flights AS f ON f.flight_id = bp.flight_id
       WHERE f.status IN ( 'Arrived', 'Departed' )
         AND f.scheduled_departure >= bookings.now() - '1 mon'::interval
       GROUP BY f.arrival_airport
     ) AS ta -- total_arrived_pass_counts
     ON td.departure_airport = ta.arrival_airport
     JOIN airports AS a ON a.airport_code = td.departure_airport
ORDER BY "Всего" DESC;
```

Мы избежали вложенных подзапросов, в результате наш запрос стало легче читать и, что очень важно, время его выполнения еще уменьшилось:

```
 ...
Время: 125,967 мс
```

А где можно увидеть подзапрос, помещенный в конструкцию WITH, в плане запроса? Он отображается под строкой CTE bp, относящейся к самому верхнему узлу плана:

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Sort
   Sort Key: (((sum(bp.pass_count)) + ta.arr_pass_count)) DESC
   CTE bp
     -> HashAggregate
          Group Key: boarding_passes.flight_id
          -> Seq Scan on boarding_passes
   -> Hash Join
        Hash Cond: (f.departure_airport = ml.airport_code)
        -> Merge Join
            Merge Cond: (f.departure_airport = ta.arrival_airport)
            -> Sort
                Sort Key: f.departure_airport
                -> HashAggregate
                    Group Key: f.departure_airport
                    -> Hash Join
                        Hash Cond: (bp.flight_id = f.flight_id)
                        -> CTE Scan on bp
                        -> Hash
                            -> Seq Scan on flights f
                                Filter: (((status)::text = ANY ('{Arrived,Departed}'::...
            -> Sort
                Sort Key: ta.arrival_airport
                -> Subquery Scan on ta
                    -> HashAggregate
                        Group Key: f_1.arrival_airport
                        -> Hash Join
                            Hash Cond: (bp_1.flight_id = f_1.flight_id)
                            -> CTE Scan on bp bp_1
                            -> Hash
                                -> Seq Scan on flights f_1
                                    Filter: (((status)::text = ANY ('{Arrived,Departed...
        -> Hash
            -> Seq Scan on airports_data ml
(33 строки)
```

Временная таблица bp используется в запросе дважды (узлы CTE Scan on bp) и поэтому по умолчанию материализуется. Это означает, что результат первого — и единственного — выполнения подзапроса bp сохраняется, а при повторном обращении к таблице bp берутся уже готовые материализованные данные.

Если же результат вычисления общего табличного выражения используется в запросе только один раз, то по умолчанию материализации не происходит. В таком случае в плане не будет узлов CTE Scan: подзапрос встраивается непосредственно в запрос и оптимизируется вместе с ним.

PostgreSQL позволяет потребовать материализации, указав предложение MATERIALIZED, а для ее отмены служит предложение NOT MATERIALIZED. В ряде ситуаций «ручное» управление материализацией позволяет ускорить выполнение запроса. Конкретные примеры показаны в упражнении 1 (с. 82). А в запросах, рассматриваемых сейчас, мы будем полагаться на решения, принимаемые по умолчанию.

Нельзя ли еще упростить наш запрос? Да, это возможно.

В конструкции WITH можно написать несколько подзапросов, при этом один подзапрос может ссылаться на другие, то есть обращаться к результатам их выполнения. Конечно, циклические ссылки не допускаются.

Давайте перенесем оба подзапроса из главного запроса в конструкцию WITH. Теперь все подготовительные действия по группировке данных выполняются в конструкции WITH, а в главном запросе полученные результаты используются так, как будто они хранятся в таблицах.

```
WITH bp AS
( SELECT flight_id, count( * ) AS pass_count
  FROM boarding_passes
  GROUP BY flight_id
),
total_departed_pass_counts AS
( SELECT f.departure_airport, sum( pass_count ) AS dep_pass_count
  FROM bp
    JOIN flights AS f ON f.flight_id = bp.flight_id
  WHERE f.status IN ( 'Arrived', 'Departed' )
    AND f.scheduled_departure >= bookings.now() - '1 mon'::interval
  GROUP BY f.departure_airport
),
```

```
total_arrived_pass_counts AS
( SELECT f.arrival_airport, sum( pass_count ) AS arr_pass_count
  FROM bp
    JOIN flights AS f ON f.flight_id = bp.flight_id
  WHERE f.status IN ( 'Arrived', 'Departed' )
    AND f.scheduled_departure >= bookings.now() - '1 mon'::interval
  GROUP BY f.arrival_airport
)
SELECT td.departure_airport AS "Код а/п",
       a.airport_name AS "Аэропорт",
       td.dep_pass_count AS "Вылетели",
       ta.arr_pass_count AS "Прибыли",
       td.dep_pass_count + ta.arr_pass_count AS "Всего"
FROM total_departed_pass_counts AS td
  JOIN total_arrived_pass_counts AS ta ON td.departure_airport = ta.arrival_airport
  JOIN airports AS a ON a.airport_code = td.departure_airport
ORDER BY "Всего" DESC;

 ...
Время: 128,671 мс
```

Эта модификация не ускоряет запрос, но делает его текст более понятным.

План новой версии запроса будет точно таким же, как и предыдущей версии. Но почему это так? Почему в плане не упоминаются таблицы total_departed_pass_counts и total_arrived_pass_counts? Когда результат подзапроса, помещенного в конструкцию WITH, используется в главном запросе (или в другом подзапросе внутри WITH) только один раз, этот подзапрос по умолчанию не материализуется, а просто встраивается в общий план и оптимизируется вместе с ним. А вот подзапрос с именем bp по-прежнему используется более одного раза, пусть и не в главном запросе, а в подзапросах, находящихся, как и он сам, в конструкции WITH. Поэтому он и в этой версии запроса материализуется и отражается в плане под строкой CTE bp.

В рассмотренной версии запроса остался потенциал для ускорения: соединение временной таблицы bp и таблицы «Рейсы» (flights) выполняется дважды (в подзапросах total_departed_pass_counts и total_arrived_pass_counts). Давайте вынесем эту операцию в отдельный подзапрос flights_pass_counts в конструкции WITH.

Вот что получится:

```
WITH bp AS
( SELECT flight_id, count( * ) AS pass_count
  FROM boarding_passes
  GROUP BY flight_id
),
flights_pass_counts AS
( SELECT f.departure_airport, f.arrival_airport, bp.pass_count
  FROM flights AS f
    JOIN bp ON f.flight_id = bp.flight_id
  WHERE f.status IN ( 'Arrived', 'Departed' )
    AND f.scheduled_departure >= bookings.now() - '1 mon'::interval
),
total_departed_pass_counts AS
( SELECT departure_airport, sum( pass_count ) AS dep_pass_count
  FROM flights_pass_counts
  GROUP BY departure_airport
),
total_arrived_pass_counts AS
( SELECT arrival_airport, sum( pass_count ) AS arr_pass_count
  FROM flights_pass_counts
  GROUP BY arrival_airport
)
SELECT td.departure_airport AS "Код а/п",
       a.airport_name AS "Аэропорт",
       td.dep_pass_count AS "Вылетели",
       ta.arr_pass_count AS "Прибыли",
       td.dep_pass_count + ta.arr_pass_count AS "Всего"
FROM total_departed_pass_counts AS td
  JOIN total_arrived_pass_counts AS ta ON td.departure_airport = ta.arrival_airport
  JOIN airports AS a ON a.airport_code = td.departure_airport
ORDER BY "Всего" DESC;

 ...
Время: 109,010 мс
```

Время выполнения запроса еще немного уменьшилось. Если читатель воспользуется командой EXPLAIN, то увидит, что временной таблицы bp в плане теперь нет, но появилась таблица flights_pass_counts. Это изменение согласуется с приведенным выше объяснением.

Порядок следования подзапросов в конструкции WITH имеет значение. Например, переставив в последнем запросе местами подзапросы flights_pass_counts и bp, получим ошибку:

```
ОШИБКА:  отношение "bp" не существует
СТРОКА 3:   FROM flights AS f JOIN bp ON f.flight_id = bp.flight_id
                                   ^
ПОДРОБНОСТИ:  В WITH есть элемент "bp", но на него нельзя ссылаться из этой части
запроса.
ПОДСКАЗКА:  Используйте WITH RECURSIVE или исключите ссылки вперёд, переупорядочив
элементы WITH.
```

Из сообщения об ошибке можно сделать вывод, что подзапросы могут ссылаться на другие подзапросы, размещенные в конструкции WITH после них (так называемые «ссылки вперед»), только если используется предложение RECURSIVE. Его мы рассмотрим в следующем разделе.

2.2. Рекурсивные общие табличные выражения

В этом разделе книги мы будем работать с графами, деревьями и иерархиями, поэтому сначала дадим неформальные определения этих понятий.

Граф — это множество *вершин* (узлов), соединенных линиями, которые называют *ребрами* (дугами). Каждое ребро соединяет две вершины.

По графу можно перемещаться, переходя от одной вершины к другой, смежной с ней, формируя определенную траекторию, или *путь*, как последовательность посещенных вершин и ребер. Длина пути равна числу ребер, составляющих его. Если путь начинается и заканчивается в одной и той же вершине и при этом не имеет повторяющихся ребер, он называется *циклом*.

Связным графом называют граф, в котором от любой вершины можно добраться до любой другой.

Граф может быть *ориентированным*. В этом случае его ребра имеют направление, и переходить по ребру от вершины к вершине можно только в этом направлении.

Ребрам графа могут быть приписаны числовые значения — *веса*. Их смысл определяется спецификой конкретной задачи. Если представить маршруты перелетов, выполняемых авиакомпанией, в виде графа, то весами можно считать, например, расстояния или продолжительности полетов между аэропортами.

Частным случаем графов являются *деревья*. Они, как и графы общего вида, могут быть ориентированными (направленными) и неориентированными (ненаправленными). Неориентированное дерево — это связный граф без циклов, между любыми двумя его вершинами существует только один путь (рис. 2.1):

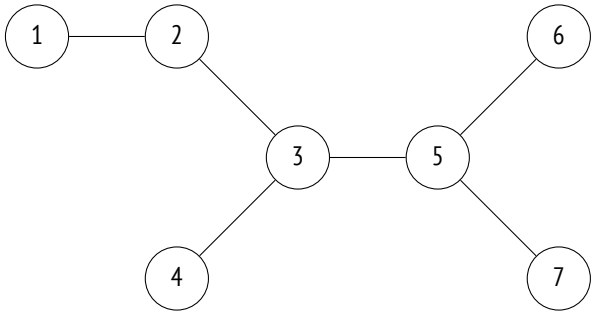


Рис. 2.1. *Пример неориентированного дерева*

Однако на практике часто удобнее иметь дело с ориентированными деревьями. Чем они примечательны? Если представить себе, что ребра графа-дерева могут вращаться относительно вершин, то можно выбрать одну из них в качестве *корня*, то есть исходной вершины, и «подвесить» граф-дерево за нее. Теперь в качестве направления перемещения по дереву можно выбрать одно из двух: либо от корня вниз, либо, наоборот, вверх к корню. В первом случае корень будет являться вершиной, в которую *не входит* ни одно ребро, а вершины, из которых ни одно ребро *не исходит*, будут называться *листьями*. При выборе второго направления перемещений, напротив, из листьев ребра будут только исходить, а в корень — только входить (рис. 2.2):

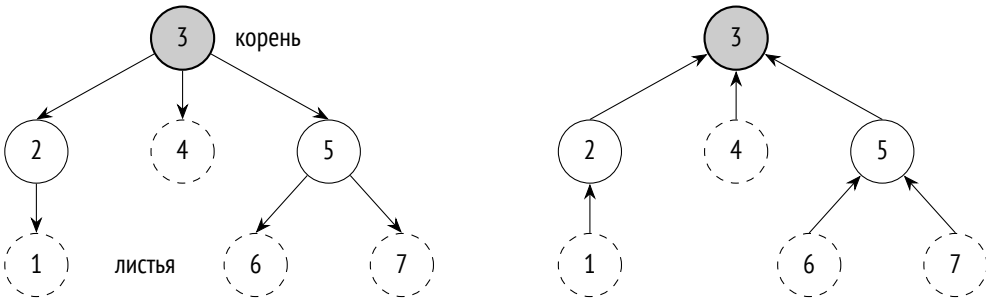


Рис. 2.2. *Пример ориентированного дерева*

Поскольку в ориентированном дереве можно двигаться лишь в одном направлении, то говорить о существовании пути между *любыми* двумя вершинами уже не приходится. Путь может быть проложен только в направлении от корня к листьям (или наоборот, в зависимости от выбранного направления).

Добавим, что вид ориентированного дерева зависит от выбора корня. Тот же граф, «подвешенный» за другую вершину, показан на рис. 2.3:

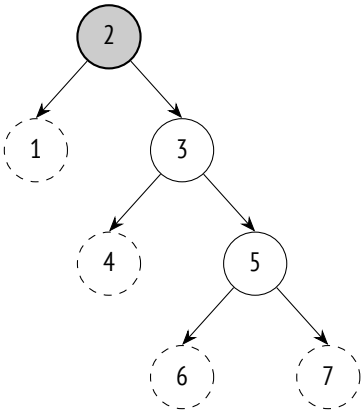


Рис. 2.3. *Другой выбор корня в ориентированном дереве*

Деревья широко используются в информатике, в частности как структура данных для индексов в СУБД. Они служат также средством моделирования иерархий. *Иерархия* — это многоуровневая структура, в которой элементы, принадлежащие соседним уровням, связаны некоторым типом взаимоотношений. В качестве примеров иерархий можно привести систему воинских формирований, административно-территориальное деление страны, классификацию биологических видов, организационные структуры компаний и учреждений, структурное устройство машин и механизмов. Отношения, имеющие место в иерархии, могут распространяться от более высоких уровней к более низким (от корня к листьям). Например, взаимоотношения между компонентами какой-либо машины можно описать формулировкой «состоит из». Также возможна и противоположная ситуация, когда отношения распространяются от нижних уровней к верхним (от листьев к корню), например «входит в состав». В каждом конкретном случае выбор в пользу одного или другого варианта делается с учетом решаемой задачи.

Сделав это элементарное введение в теорию графов, перейдем к предмету нашего обсуждения — рекурсивным общим табличным выражениям. Именно они и используются для обработки иерархических данных.

В качестве примера возьмем рекурсивный запрос, приведенный в подразделе документации 7.8.2 «Рекурсивные запросы», но рассмотрим его детально, создав необходимую таблицу и наполнив ее данными. Этот пример показывает структуру и составные части какой-либо машины или агрегата.

Далее мы модифицируем этот запрос, чтобы лучше понять механизм его выполнения. А сейчас, перед началом экспериментов, внесем в исходный вариант лишь «косметические» изменения, не затрагивающие логику его работы: переставим местами столбцы sub_part и part в конструкции WITH, изменим псевдоним подзапроса included_parts с pr на ip, добавим предложение ORDER BY, а также укажем в качестве «исходного» пункта вывода иерархии значение самолет вместо our_product:

```
WITH RECURSIVE included_parts( part, sub_part, quantity ) AS
( SELECT part, sub_part, quantity
  FROM parts
  WHERE part = 'самолет'
  UNION ALL
  SELECT p.part, p.sub_part, p.quantity * ip.quantity
  FROM included_parts ip,
       parts p
  WHERE p.part = ip.sub_part
)
SELECT sub_part, sum( quantity ) AS total_quantity
FROM included_parts
GROUP BY sub_part
ORDER BY sub_part;
```

Для проверки запроса в действии нам потребуется таблица с иерархическими данными.

```
CREATE TABLE parts
( part text,
  sub_part text,
  quantity int NOT NULL,
  PRIMARY KEY ( part, sub_part )
);
```

Оставаясь в рамках авиационной тематики, воспользуемся структурной схемой узлов самолета с отношением «состоит из».

В конструкции самолета есть узлы, представленные как в единственном экземпляре, так и в нескольких, например колесо. Если представить структурное устройство самолета в виде графа, в котором каждый физический экземпляр узлов и деталей (например, каждое колесо) является отдельной вершиной, то такой граф будет деревом, поскольку к любому узлу или любой детали на этом графе будет вести единственный путь (рис. 2.4):

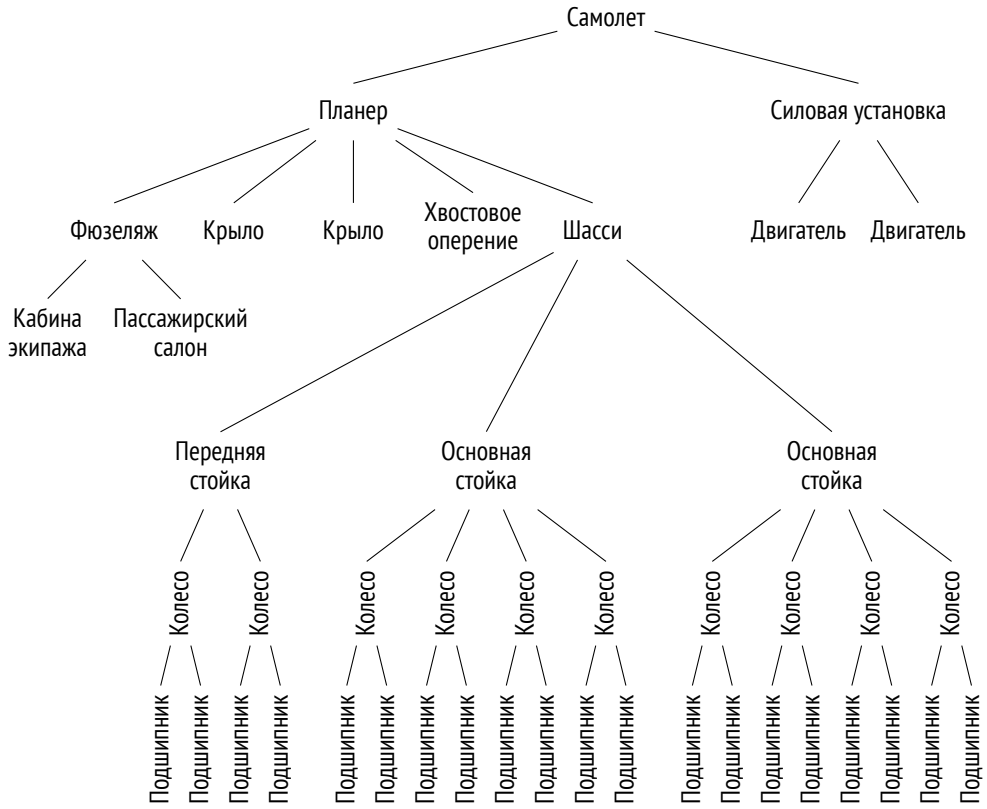


Рис. 2.4. *Структурное устройство самолета в виде дерева*

Однако в случаях, когда в состав какого-либо узла самолета входит несколько одинаковых деталей (или подузлов), мы не будем создавать для каждой из них

отдельный элемент иерархии, то есть отдельную строку в таблице, а обойдемся одним элементом с указанием количества этих однотипных деталей.

Таким образом, в нашем представлении структурного устройства самолета каждая вершина графа соответствует какому-то *виду* узлов или деталей, а вес ребра графа соответствует количеству этих деталей, входящих в состав узла более высокого уровня иерархии. В результате оказывается, что к некоторым вершинам графа ведет более одного пути, например к вершине, обозначающей колесо: ведь оно входит в состав передней и двух основных стоек (рис. 2.5):

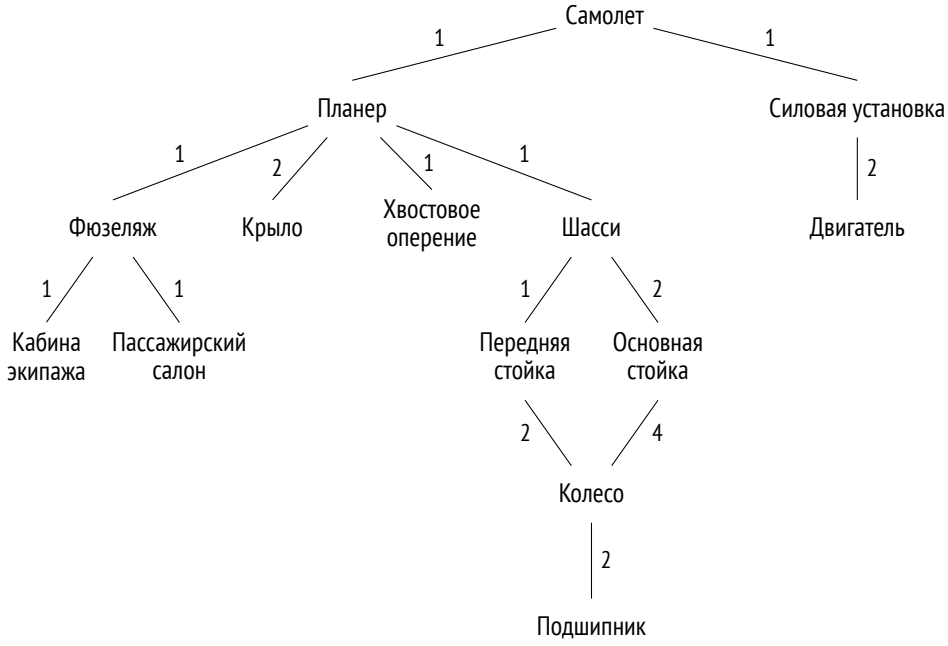


Рис. 2.5. *Структурное устройство самолета в виде графа*

Вес ребра, соединяющего вершину «передняя стойка» с вершиной «колесо», равен двум, а вес ребра, соединяющего вершину «основная стойка» с *той же* вершиной «колесо», — четырем. Таким образом, сформированный нами граф не является деревом.

Для ввода показанных на рисунках данных воспользуемся командой `COPY`.

```
COPY parts FROM STDIN WITH ( format csv );
Вводите данные для копирования, разделяя строки переводом строки.
Закончите ввод строкой '\.' или сигналом EOF.
>> самолет,планер,1
самолет,силовая установка,1
планер,фюзеляж,1
планер,крыло,2
планер,хвостовое оперение,1
планер,шасси,1
фюзеляж,кабина экипажа,1
фюзеляж,пассажирский салон,1
силовая установка,двигатель,2
шасси,передняя стойка,1
шасси,основная стойка,2
передняя стойка,колесо,2
основная стойка,колесо,4
колесо,подшипник,2
\.
COPY 14
```

Сделаем выборку с помощью представленного выше запроса:

```
      sub_part      | total_quantity
--------------------+----------------
 двигатель          |              2
 кабина экипажа     |              1
 колесо             |             10
 крыло              |              2
 основная стойка    |              2
 пассажирский салон |              1
 передняя стойка    |              1
 планер             |              1
 подшипник          |             20
 силовая установка  |              1
 фюзеляж            |              1
 хвостовое оперение |              1
 шасси              |              1
(13 строк)
```

Мы видим, что число колес равно 10, а число подшипников — 20, как это и должно быть при тех исходных данных, которые содержатся в таблице parts.

Теперь обратимся к запросу. Чтобы общее табличное выражение стало рекурсивным, мы дополнили конструкцию WITH предложением RECURSIVE. Рекурсивное общее табличное выражение состоит из двух компонентов: нерекурсивной

части и рекурсивной части, которые соединяются оператором UNION или UNION ALL. В чем заключается различие между этими операторами? Если в ходе вычислений появляются строки-дубликаты, их можно либо сохранять, либо отбрасывать. Для реализации первого подхода используется оператор UNION ALL, а для реализации второго — UNION, который требует больше ресурсов.

Вообще, как сказано в подразделе документации 7.8.2 «Рекурсивные запросы», внутренняя реализация вычислительного процесса является итерационной, а не рекурсивной, хотя в синтаксисе команды присутствует предложение RECURSIVE.

Как же проходят вычисления? Сначала вычисляется нерекурсивная часть, причем только один раз (рис. 2.6).

```
...
( SELECT part, sub_part, quantity
  FROM parts
  WHERE part = 'самолет'
...
```

Все строки (кроме, возможно, строк-дубликатов), порожденные в нерекурсивном подзапросе, включаются в формируемый итоговый результат всего рекурсивного общего табличного выражения, а также помещаются во временную *рабочую таблицу*. Именно к ней и происходит обращение в рекурсивной части при использовании имени подзапроса (в нашем примере — included_parts).

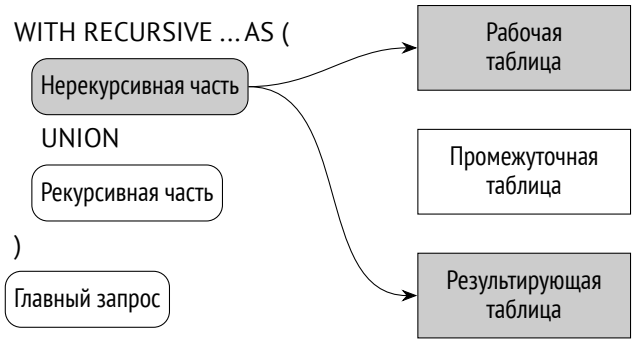


Рис. 2.6. *Вычисление нерекурсивной части*

В нашем примере в нерекурсивной части выбираются узлы верхнего уровня, входящие непосредственно в состав самолета в целом. Строки, описывающие эти узлы, будут помещены во временную рабочую таблицу и будут доступны на первой итерации выполнения рекурсивной части.

Переходим к рекурсивной части общего табличного выражения. Здесь, пока рабочая таблица не пуста, повторяются следующие действия:

1. Вычисляется подзапрос в рекурсивной части (рис. 2.7).

```
...
  SELECT p.part, p.sub_part, p.quantity * ip.quantity
  FROM included_parts ip, parts p
  WHERE p.part = ip.sub_part
...
```

   Здесь элемент `included_parts` является рекурсивной ссылкой на подзапрос. Эта ссылка обращается к текущему содержимому рабочей таблицы. При использовании `UNION ALL` строки-дубликаты сохраняются, а при использовании `UNION` — отбрасываются. Причем это касается как дублирующихся строк, сформированных на текущей итерации, так и строк, которые дублируют ранее полученные строки. Все оставшиеся строки добавляются в формируемый *итоговый результат* рекурсивного общего табличного выражения, а также помещаются во временную *промежуточную таблицу*. В отличие от рабочей таблицы, сформированной на *предыдущей* итерации, она содержит строки, порожденные *текущей* итерацией.

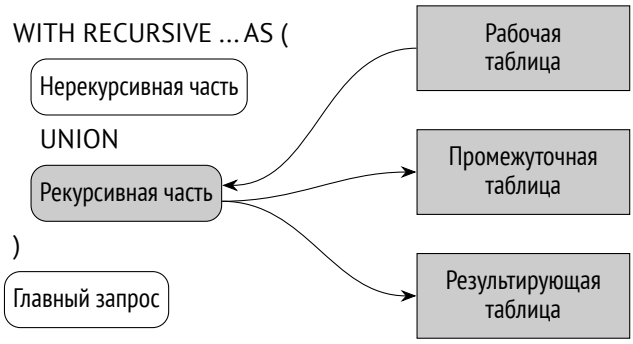


Рис. 2.7. *Вычисление рекурсивной части*

2. Содержимое рабочей таблицы заменяется содержимым промежуточной таблицы, а затем промежуточная таблица очищается (рис. 2.8).

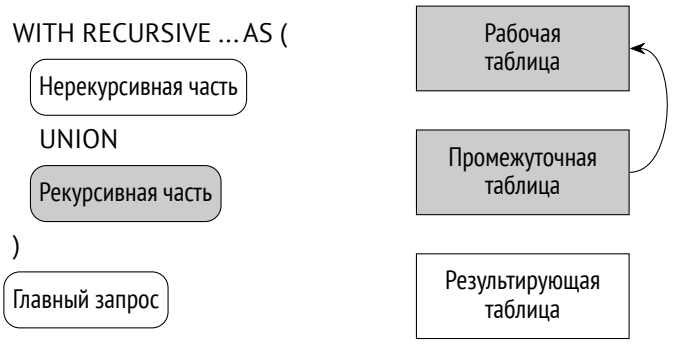


Рис. 2.8. *Замена рабочей таблицы и очистка промежуточной*

Таким образом, на первой итерации рекурсивной части в рабочей таблице included_parts находятся строки, описывающие узлы «планер» и «силовая установка». При соединении таблицы included_parts с таблицей parts будут сформированы строки, описывающие узлы следующего уровня иерархии, входящие в состав этих двух узлов. На второй итерации в таблице included_parts окажутся строки, выбранные из таблицы parts на первой итерации, и т. д.

В рассматриваемом запросе используется оператор UNION ALL, а не UNION. Поскольку в данном случае заведомо не будут формироваться строки-дубликаты, то и устранять их, затрачивая на это ресурсы СУБД, не потребуется.

Важный вопрос: почему рекурсивный (итерационный) процесс в приведенном запросе в итоге завершается? Дело в том, что рекурсивная часть общего табличного выражения в какой-то момент не вернет ни одной строки. Рано или поздно в таблице parts не найдется строк, удовлетворяющих условию p.part = ip.sub_part, поскольку в составе конструкции самолета есть элементарные детали, не имеющие составных частей (в нашем примере это крыло, подшипник и др.). Поэтому и результат соединения таблиц included_parts и parts *будет пустым*. Конечно, в реальном мире крыло самолета является сложной конструкцией, но в нашем примере ситуация намеренно упрощена. Более детально изучить этот вопрос можно с помощью упражнений 4 (с. 89), 5 (с. 89), 6 (с. 90) и 7 (с. 91).

После завершения вычислений общего табличного выражения выполняется главный запрос (рис. 2.9):

```
...
SELECT sub_part, sum( quantity ) AS total_quantity
FROM included_parts
GROUP BY sub_part
ORDER BY sub_part;
```

Здесь по имени included_parts запрос получает доступ уже не к текущему содержимому временной рабочей таблицы, а к итоговому результату вычисления общего табличного выражения.

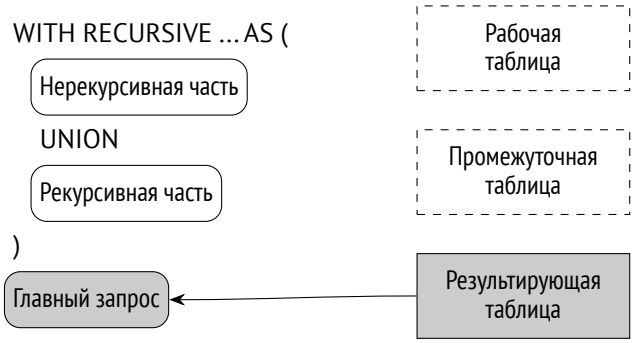


Рис. 2.9. *Выполнение главного запроса*

За счет чего же нам удается получить правильное общее количество тех узлов и деталей, которые входят в состав нескольких более крупных узлов? В качестве примеров можно привести подшипники, являющиеся составной частью колес, и колеса, входящие в переднюю и основные стойки. Объяснение этому можно найти в рекурсивной части общего табличного выражения. В строке

```
SELECT p.part, p.sub_part, p.quantity * ip.quantity
```

сомножитель p.quantity означает число деталей в узле, а ip.quantity — число узлов этого вида. Это следует из того, что, «разбирая» самолет на части, мы идем сверху вниз: от более крупных узлов к более мелким составным частям. Поэтому временная рабочая таблица included_parts соответствует текущему уровню иерархии конструктивных частей самолета, а из таблицы parts выбираются *составные части* для узлов, представленных на текущем уровне иерархии.

Для того чтобы лучше разобраться в механизме выполнения рекурсивного запроса, давайте модифицируем его. Уберем группировку и добавим в вывод номер итерации и значение, сформированное на основе сведений об узлах, находящихся на двух смежных уровнях иерархии. При этом подузел текущего уровня иерархии выступает в роли узла на следующем, более низком, уровне, поэтому в комбинированном столбце path_to_sub_part будет выводиться завершающий фрагмент пути к узлу, показанному в столбце sub_part. Сведения о текущем уровне иерархии выбираются из временной таблицы included_parts, а сведения о следующем, более низком, уровне — из таблицы parts. Воспользуемся внешним соединением (LEFT OUTER JOIN), чтобы включить в вывод те узлы, которые в нашем примере не имеют составных частей.

```
WITH RECURSIVE included_parts( iteration, part, sub_part, quantity, path_to_sub_part ) AS
( SELECT 1, part, sub_part, quantity,
         part || ' -> ' || sub_part || ' (x' || quantity || ')'
    FROM parts WHERE part = 'самолет'
  UNION ALL
  SELECT iteration + 1, p.part, p.sub_part, p.quantity * ip.quantity,
         -- комбинированное значение
         ip.part || ' -> ' || ip.sub_part || ' (x' || ip.quantity || ')' ||
           CASE
             WHEN p.part IS NOT NULL
               THEN ' -> ' || p.sub_part || ' (x' || p.quantity || ')'
             ELSE ' -> составных частей нет'
           END
  FROM included_parts ip
    LEFT OUTER JOIN parts p ON p.part = ip.sub_part
  WHERE ip.part IS NOT NULL
)
SELECT iteration, part, sub_part, quantity, path_to_sub_part
FROM included_parts
ORDER BY iteration \gx
```

Дополнительно в предложение WHERE рекурсивной части пришлось включить условие ip.part IS NOT NULL, поскольку без него рекурсивный процесс зациклится. Ведь рано или поздно дело дойдет до неделимых узлов (деталей), и значения столбцов p.part и p.sub_part в списке SELECT рекурсивной части станут равными NULL (сработает внешнее соединение). Значит, на следующей итерации эти значения NULL будут подставлены уже в столбцы ip.part и ip.sub_part. Говоря другими словами, строка, содержащая значения NULL, будет записана во временную промежуточную таблицу и на следующей итерации попадет уже

во временную рабочую таблицу. Поскольку мы используем внешнее соединение, то при отсутствии условия `ip.part IS NOT NULL` снова будет сформирована комбинированная строка и т. д.

Теперь мы можем увидеть развертывание рекурсивного (точнее говоря, итерационного) процесса во времени, обращая внимание на номер итерации. Покажем не весь результат выполнения запроса, а только наиболее информативные строки, причем сделаем это в расширенном режиме вывода.

Строки, выведенные на первой итерации, формируются нерекурсивной частью общего табличного выражения, которая обращается только к таблице parts. В скобках приводится количество составных частей. Символ × (знак умножения) означает, что это количество входит в состав *каждого* узла верхнего уровня.

```
-[ RECORD 1 ]----+-------------------------------------------------------------
iteration        | 1
part             | самолет
sub_part         | планер
quantity         | 1
path_to_sub_part | самолет -> планер (x1)
-[ RECORD 2 ]----+-------------------------------------------------------------
iteration        | 1
part             | самолет
sub_part         | силовая установка
quantity         | 1
path_to_sub_part | самолет -> силовая установка (x1)
 ...
```

Давайте проследим процесс подсчета количества колес и подшипников. Колеса входят в состав передней стойки и основных стоек, а подшипники — в состав колес, установленных на этих стойках.

Запись номер 16 демонстрирует, что в состав шасси входят две основные стойки, а каждая из них включает в себя по четыре колеса. Поэтому в составе основных стоек всего восемь колес, что показано в столбце `quantity`. Это значение равно произведению двух чисел, взятых из столбца `path_to_sub_part`. Аналогично запись номер 15 говорит, что в состав шасси входит одна передняя стойка, включающая два колеса. Общее число колес в составе передней стойки, таким образом, равно двум.

```
 ...
-[ RECORD 15 ]---+-------------------------------------------------------------
iteration        | 4
part             | передняя стойка
sub_part         | колесо
quantity         | 2
path_to_sub_part | шасси -> передняя стойка (x1) -> колесо (x2)
-[ RECORD 16 ]---+-------------------------------------------------------------
iteration        | 4
part             | основная стойка
sub_part         | колесо
quantity         | 8
path_to_sub_part | шасси -> основная стойка (x2) -> колесо (x4)
 ...
```

Теперь перейдем к подшипникам. Запись номер 19 демонстрирует, что в состав основных стоек входят восемь колес, а каждое из них включает в себя по два подшипника. Поэтому в составе колес, установленных на основные стойки, всего 16 подшипников, что отражено в столбце `quantity`. Запись номер 20 говорит, что в состав колес, установленных на передней стойке, входят всего четыре подшипника.

```
 ...
-[ RECORD 19 ]---+-------------------------------------------------------------
iteration        | 5
part             | колесо
sub_part         | подшипник
quantity         | 16
path_to_sub_part | основная стойка -> колесо (x8) -> подшипник (x2)
-[ RECORD 20 ]---+-------------------------------------------------------------
iteration        | 5
part             | колесо
sub_part         | подшипник
quantity         | 4
path_to_sub_part | передняя стойка -> колесо (x2) -> подшипник (x2)
 ...
```

Обратите внимание, что в столбце `path_to_sub_part` записи номер 16 сведения об основной стойке взяты из таблицы `parts`, поэтому дано число колес для *одной* основной стойки. А в столбце `path_to_sub_part` записи номер 19 сведения об основной стойке взяты из таблицы `included_parts`, поэтому число колес уже подсчитано для *обеих* основных стоек. Аналогичные рассуждения на основе рассмотрения записей номер 15 и 20 можно привести для передней стойки и колес, входящих в ее состав.

Для тех деталей, которые не имеют составных частей, в поле path_to_sub_part приводится общее количество этих деталей в составе более крупных узлов. В качестве примера служит подшипник. Эта деталь в количестве четырех штук входит в состав колес, установленных на передней стойке, и в количестве шестнадцати штук входит в состав колес, установленных на основных стойках, что в сумме дает 20.

Обратите внимание, что сведения в столбце path_to_sub_part в записях номер 21 и 22 взяты только из таблицы included_parts, поскольку в таблице parts нет строк, в которых значение столбца part было бы равно «подшипник», и продолжить процесс детализации уже невозможно. Поэтому в этих строках приводится только общее количество подшипников в составе двух групп более крупных узлов (вспомните, что значение столбца quantity таблицы included_parts вычисляется как p.quantity * ip.quantity). Сведения о самих этих узлах уже были показаны ранее на предыдущих итерациях.

В записях номер 21 и 22 значения столбцов part и sub_part равны NULL, поэтому во временную рабочую таблицу included_parts на этой итерации попадут строки, содержащие только значения NULL, на следующей итерации сработает условие ip.part IS NOT NULL, и процесс завершится, не зацикливаясь.

```
 ...
-[ RECORD 21 ]---+-------------------------------------------------------------
iteration        | 6
part             |
sub_part         |
quantity         |
path_to_sub_part | колесо -> подшипник (x4) -> составных частей нет
-[ RECORD 22 ]---+-------------------------------------------------------------
iteration        | 6
part             |
sub_part         |
quantity         |
path_to_sub_part | колесо -> подшипник (x16) -> составных частей нет
 ...
```

Надо добавить, что номер итерации, на которой завершается процесс детализации того или иного узла, зависит от степени вложенности конкретных узлов и деталей, представленных в таблице parts. Это может случиться, например, на третьей итерации:

```
 ...
-[ RECORD 14 ]---+-------------------------------------------------------------
iteration        | 3
part             |
sub_part         |
quantity         |
path_to_sub_part | планер -> крыло (x2) -> составных частей нет
 ...
```

Обратите внимание, что когда процесс детализации завершается, поля part, sub_part и quantity остаются пустыми. Поля sub_part и quantity в этом случае заполнить невозможно, поскольку более мелких частей данный узел не имеет, однако его название можно поместить в поле part. Предлагаем читателю выполнить данную модификацию запроса самостоятельно. Для этого можно воспользоваться функцией coalesce вместо буквального вывода столбца p.part, а в предложении WHERE заменить столбец ip.part на другой. Посмотрите и на условие соединения таблиц included_parts и parts: p.part = ip.sub_part.

```
...
  UNION ALL
  SELECT iteration + 1, coalesce( ... ), p.sub_part,
         p.quantity * ip.quantity,
...
       ON p.part = ip.sub_part
  WHERE ... IS NOT NULL
...
```

В качестве еще одного примера применения рекурсивных общих табличных выражений рассмотрим штатное расписание авиакомпании. Если бы мы проектировали настоящую базу данных, то в ней, по всей видимости, понадобились бы такие таблицы:

- «Персонал» — сведения о работниках авиакомпании (табельный номер или другой уникальный идентификатор, фамилия, имя, отчество и другие данные);
- «Справочник должностей» — перечень должностей, предусмотренных в нашей авиакомпании: от президента до бортпроводника;
- «Штатное расписание» — сведения о штатных позициях, предусмотренных в организации, с указанием их подчиненности;
- «Назначения» — сведения о назначениях сотрудников на конкретные позиции, предусмотренные штатным расписанием.

Таблица «Назначения» необходима, поскольку один сотрудник может занимать более одной позиции (это называется совместительством), и, с другой стороны, одну позицию могут разделять несколько сотрудников, занимая по доле ставки. Могут также оставаться незанятые (вакантные) позиции.

Все должности были бы представлены в «Справочнике должностей» только один раз. При этом некоторые из них могли бы встречаться в таблице «Штатное расписание» более одного раза, например «командир корабля», «бортпроводник», «программист» и др. Назначение «Справочника должностей» — в обеспечении единообразного именования одних и тех же должностей.

Но поскольку этот учебник предназначен не для изучения искусства проектирования баз данных, мы обойдемся одной денормализованной таблицей — «Штатное расписание»:

```
CREATE TABLE staff
( position_id integer PRIMARY KEY,  -- ID позиции
  position_title text NOT NULL,     -- наименование должности
  person_name text NOT NULL UNIQUE, -- имя работника
  boss_position_id integer,         -- ID начальника
  FOREIGN KEY ( boss_position_id ) REFERENCES staff ( position_id )
);
```

Столбец position_title может содержать неуникальные значения, так как названия должностей могут повторяться. Например, может быть несколько позиций для должности «второй пилот».

Наличия вакансий не предусматриваем — все позиции должны быть заняты работниками, поэтому для столбца person_name добавим ограничение NOT NULL.

У нас нет отдельной таблицы «Персонал» и имя работника служит его идентификатором, поэтому столбец person_name сделаем уникальным. Конечно, это большое упрощение.

Значение NULL в столбце boss_position_id означает, что эта строка описывает самого главного начальника в компании.

Удалять руководителя, у которого еще есть подчиненные, не позволим. Поэтому в определении внешнего ключа нет предложения ON DELETE CASCADE или ON DELETE SET NULL. Изменять идентификатор позиции мы также не планируем, поэтому не включаем и предложение ON UPDATE CASCADE.

Заполним таблицу «Штатное расписание» (staff) данными (рис. 2.10).

```
COPY staff FROM STDIN WITH ( format csv );
Вводите данные для копирования, разделяя строки переводом строки.
Закончите ввод строкой '\.' или сигналом EOF.
>> 1,Президент,Олег,
2,Вице-президент 1,Иван,1
3,Вице-президент 2,Федор,1
4,Программист,Павел,2
5,Программист,Егор,2
6,Командир корабля,Марат,3
7,Второй пилот,Игорь,6
8,Старший бортпроводник,Дарья,6
9,Бортпроводник,Ксения,8
10,Командир корабля,Артем,3
11,Второй пилот,Мария,10
12,Старший бортпроводник,Алиса,10
13,Бортпроводник,Алина,12
\.
COPY 13
```

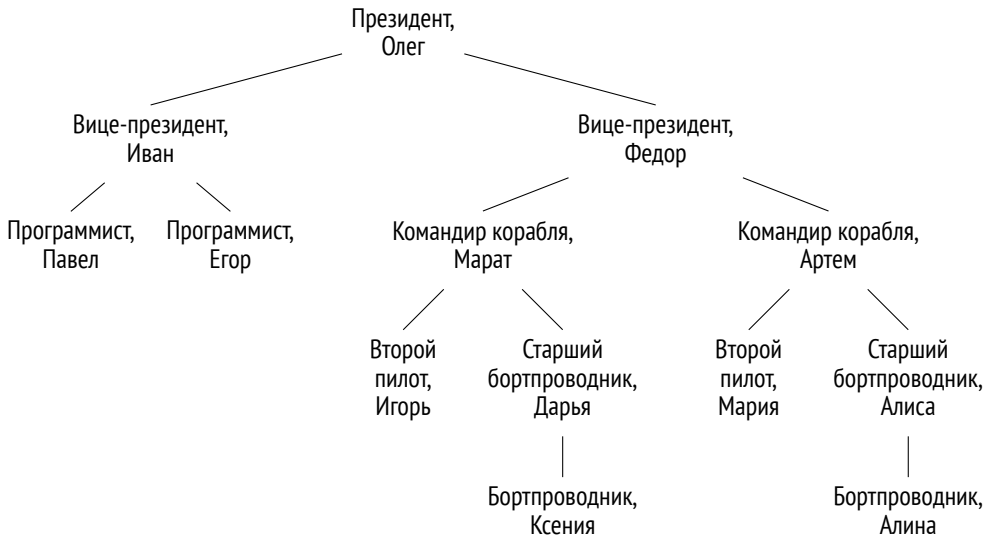


Рис. 2.10. *Штатное расписание авиакомпании*

У президента компании нет руководителя, поэтому в его строке нет значения для столбца boss_position_id. В нашей реализации штатного расписания мы для каждой позиции приводим *вышестоящую* должность. Образно говоря, мы формируем иерархию должностей *снизу вверх*.

Предположим, что нам потребовалось вывести руководителей конкретного работника на всех уровнях иерархии. Запрос может быть таким:

```
WITH RECURSIVE search_staff
( position_title, person_name, position_id, boss_position_id, level
) AS
( SELECT s.position_title, s.person_name, s.position_id, s.boss_position_id, 1
  FROM staff s
  WHERE s.person_name = 'Павел'
  UNION ALL
  SELECT s.position_title, s.person_name, s.position_id, s.boss_position_id, ss.level + 1
  FROM search_staff ss,
       staff s
  WHERE s.position_id = ss.boss_position_id
)
SELECT position_title, person_name, position_id, boss_position_id,
       ( SELECT max( level ) FROM search_staff ) - level + 1 AS level
FROM search_staff
ORDER BY level DESC;
```

В нерекурсивной части конструкции WITH этого запроса выбирается та строка из таблицы «Штатное расписание» (staff), которая соответствует интересующему нас работнику. Эта строка помещается во временную рабочую таблицу. В рекурсивной части конструкции WITH эта временная таблица выступает под именем search_staff. Мы соединяем ее с таблицей staff. Поскольку мы движемся по иерархии *снизу вверх*, то условие соединения сопоставляет такие строки, в которых позиция *работника* на следующем уровне иерархии из таблицы staff совпадает с позицией *руководителя* работника на текущем уровне иерархии из таблицы search_staff. Проще говоря, мы ищем *руководителя* текущего работника (из столбца boss_position_id) в списке *работников* (в столбце position_id).

Этот процесс повторяется до тех пор, пока не дойдет до строки, соответствующей президенту компании. Когда «президентская» строка будет выбрана и помещена в рабочую таблицу search_staff, для нее не будет найдена парная строка в таблице staff, поскольку поле boss_position_id для президента компании имеет значение NULL, и соединение таблиц search_staff и staff не вернет ни одной строки. На этом рекурсивный процесс завершится, а все накопленные строки станут доступны внешнему (главному) запросу под именем search_staff.

В нашем общем табличном выражении не возникает строк-дубликатов (ведь у каждого работника есть только один непосредственный начальник, имеющий

уникальное значение поля position_id), поэтому следует использовать оператор UNION ALL, чтобы не увеличивать нагрузку на сервер избыточным отсеиванием строк-дубликатов.

В главном запросе мы выводим и значение уровня иерархии для каждого руководителя. Поскольку мы выполняли обход иерархии снизу вверх, то самый нижний уровень получит номер 1, а самый верхний — уровень президента компании — получит наибольшее значение, что представляется нелогичным. Для того чтобы вывести номера уровней иерархии в правильном порядке, мы воспользовались подзапросом:

```
...
      ( SELECT max( level ) FROM search_staff ) - level + 1
...
```

Небольшое дополнение: псевдоним s для таблицы staff в нерекурсивной части конструкции WITH используется лишь для единообразия с рекурсивной частью.

```
  position_title  | person_name | position_id | boss_position_id | level
------------------+-------------+-------------+------------------+-------
 Программист      | Павел       |           4 |                2 |     3
 Вице-президент 1 | Иван        |           2 |                1 |     2
 Президент        | Олег        |           1 |                  |     1
(3 строки)
```

Теперь давайте решим противоположную задачу: выберем всех подчиненных указанного работника. Запрос может быть таким:

```
WITH RECURSIVE search_staff
( position_title, person_name, position_id, boss_position_id, level
) AS
( SELECT s.position_title, s.person_name, s.position_id, s.boss_position_id, 1
  FROM staff s
  WHERE s.person_name = 'Федор'
  UNION ALL
  SELECT s.position_title, s.person_name, s.position_id, s.boss_position_id, ss.level + 1
  FROM search_staff ss,
       staff s
  WHERE s.boss_position_id = ss.position_id
)
SELECT position_title, person_name, position_id, boss_position_id, level
FROM search_staff
ORDER BY level, boss_position_id, position_id;
```

Этот запрос очень походит на предыдущий. Общее табличное выражение в нем отличается только тем, что в его рекурсивной части в условии WHERE переставлены местами имена столбцов position_id и boss_position_id.

Было так:

```
...
  FROM search_staff ss,
       staff s
  WHERE s.position_id = ss.boss_position_id
...
```

Стало так:

```
...
  FROM search_staff ss,
       staff s
  WHERE s.boss_position_id = ss.position_id
...
```

В результате направление обхода иерархии изменилось: было снизу вверх, а стало сверху вниз. В рекурсивной части общего табличного выражения мы выбираем *всех подчиненных* тех работников, которые были выбраны на предыдущей итерации, а в предыдущей задаче мы выбирали *одного непосредственного начальника* работника, выбранного на предыдущей итерации. Теперь строки временной рабочей таблицы search_staff на каждой итерации описывают начальников текущего уровня иерархии, а строки постоянной таблицы «Штатное расписание» (staff) соответствуют подчиненным этих начальников.

Вот такой получается результат:

```
    position_title     | person_name | position_id | boss_position_id | level
-----------------------+-------------+-------------+------------------+-------
 Вице-президент 2      | Федор       |           3 |                1 |     1
 Командир корабля      | Марат       |           6 |                3 |     2
 Командир корабля      | Артем       |          10 |                3 |     2
 Второй пилот          | Игорь       |           7 |                6 |     3
 Старший бортпроводник | Дарья       |           8 |                6 |     3
 Второй пилот          | Мария       |          11 |               10 |     3
 Старший бортпроводник | Алиса       |          12 |               10 |     3
 Бортпроводник         | Ксения      |           9 |                8 |     4
 Бортпроводник         | Алина       |          13 |               12 |     4
(9 строк)
```

В качестве точки отсчета для нумерации уровней иерархии выбран уровень того руководителя, для которого выполняется запрос. В обоих рассмотренных запросах вычисляется уровень иерархии level, который позволяет упорядочить *вывод* уже сформированных строк в соответствии с принципом «сначала в ширину» (breadth-first). Однако управлять порядком *обхода* дерева (порядком выбора строк из таблицы) предложение ORDER BY не позволяет: этот порядок зависит от реализации SQL в среде СУБД, как сказано в подразделе документации 7.8.2.1 «Порядок поиска».

2.3. Массивы в общих табличных выражениях

При обработке данных, представленных в виде графов, массивы помогают решать такие задачи, как поиск пути между вершинами или выявление циклов. Для демонстрации сначала обратимся к абстрактному примеру иерархически организованных данных, а в конце раздела рассмотрим правдоподобную ситуацию, в которой используется граф общего вида.

2.3.1. Выявление циклов

При работе с иерархическими данными важно убедиться в их корректности: в графе между любыми двумя вершинами должен существовать только один путь и не должно быть циклов.

Итак, давайте создадим таблицу «Иерархия» (hier — от слова hierarchy) и заполним ее данными (рис. 2.11). Ребра будут направлены от вышестоящих вершин к вершинам более низкого уровня. Столбец data нашей таблицы будет содержать числовые значения, отражающие веса ребер. В зависимости от предметной области эти веса могут иметь разный смысл.

Конечно, абстрактный пример не позволит содержательно интерпретировать полученные результаты, однако он представляет обобщенную модель, которую можно применить, пусть и с некоторыми поправками, к различным реальным ситуациям.

```
CREATE TABLE hier
( vertex_from integer,
  vertex_to integer,
  data numeric,
  PRIMARY KEY ( vertex_from, vertex_to )
);
CREATE TABLE
COPY hier FROM STDIN WITH ( format csv );
Вводите данные для копирования, разделяя строки переводом строки.
Закончите ввод строкой '\.' или сигналом EOF.
>> 1,2,4.7
1,3,5.6
2,4,6.3
2,5,1.9
3,6,3.5
3,7,2.8
3,8,4.1
5,9,3.3
5,10,4.5
6,11,2.7
6,12,1.3
9,13,2.1
\.
COPY 12
```

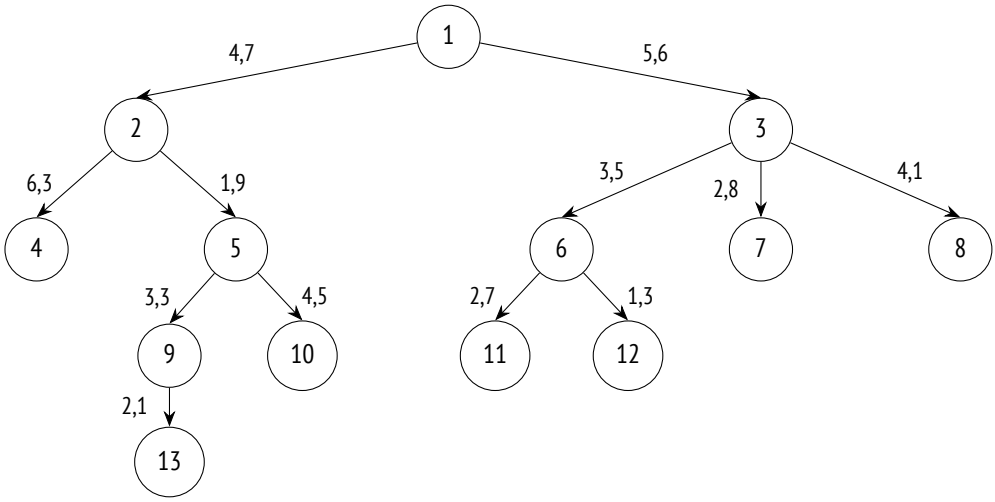


Рис. 2.11. *Иерархические данные*

Подобная таблица может использоваться и при проектировании новой иерархически организованной системы, и при исследовании уже существующей

иерархии. Конечно, столбцы реальной таблицы будут иметь не абстрактные, а конкретные наименования, отражающие специфику предметной области. В таблице могут присутствовать и дополнительные столбцы.

В первом случае, видимо, имеет смысл предусмотреть в таблице ограничения целостности, которые будут препятствовать отклонениям от строгой иерархии, таким как циклы и наличие более одного пути от ее начала до любой другой вершины. Например, уникальный ключ по столбцу `vertex_to` сделает невозможным возникновение множественных путей, для которых необходимо наличие хотя бы одной вершины, в которую входит более одного ребра.

Во втором случае, когда система уже существует, в ней могут обнаружиться вышеназванные «дефекты» иерархии. Например, в организационной структуре компании может иметь место двойное подчинение отдельных работников. Однако для того чтобы выявить имеющиеся отклонения от правильной иерархической структуры, мы должны иметь возможность ввести в таблицу все фактически существующие связи-ребра.

Давайте для нашего примера выберем второй вариант задачи, то есть исследование существующей иерархической системы. Поэтому не будем накладывать на таблицу никаких дополнительных ограничений целостности. Ограничимся лишь уже созданным первичным ключом по столбцам `vertex_from` и `vertex_to`. Его наличие не позволит появиться ребрам-дубликатам.

Для выявления множественных путей нужно сначала научиться получать пути, ведущие из начала иерархии ко всем другим вершинам. В нашем примере начало иерархии — вершина с номером 1. Но если бы этот номер не был заранее известен, его можно было бы определить, вспомнив, что в вершину, являющуюся начальной точкой иерархии, не входит ни одно ребро, то есть она не фигурирует в столбце `vertex_to` ни в одной строке таблицы «Иерархия» (`hier`). Одним из решений могло бы быть такое:

```
SELECT DISTINCT( vertex_from ) FROM hier h1
WHERE NOT EXISTS (
  SELECT 1 FROM hier h2 WHERE h2.vertex_to = h1.vertex_from
);
 vertex_from
-------------
           1
(1 строка)
```

Здесь в списке SELECT подзапроса присутствует константа 1. В подразделе документации 9.24.1 «EXISTS» рекомендуется поступать именно таким образом в случаях, когда нас интересует только факт наличия хотя бы одной строки, удовлетворяющей условию, а значения ее полей в запросе не используются. В этом списке SELECT можно было написать любую константу, и даже просто NULL (или вообще оставить этот список пустым).

Решение этой же задачи на основе внешнего соединения таблицы «Иерархия» (hier) с самой собой выглядит так:

```
SELECT DISTINCT h1.vertex_from
FROM hier h1
  LEFT OUTER JOIN hier h2 ON h2.vertex_to = h1.vertex_from
WHERE h2.vertex_to IS NULL;
```

Этот же результат можно получить и с помощью совсем простого запроса (он выполняется немного быстрее двух предыдущих):

```
SELECT vertex_from FROM hier
EXCEPT
SELECT vertex_to FROM hier;
```

Конечно, в реальности иерархия может распасться на несколько фрагментов, не связанных друг с другом. Например, если из дерева удалить одно ребро, то это дерево распадется на два дерева. При этом каждая из двух вершин удаленного ребра будет принадлежать разным вновь полученным деревьям. Такой случай рассмотрен в упражнении 10 (с. 95).

Теперь приведем запрос, который решает основную задачу: получает пути, ведущие из начала иерархии ко всем другим вершинам.

```
WITH RECURSIVE search_hier( vertex_from, vertex_to, data, depth ) AS
( SELECT h.vertex_from, h.vertex_to, h.data, 1
  FROM hier h
  WHERE vertex_from = 1
  UNION ALL
  SELECT h.vertex_from, h.vertex_to, h.data, sh.depth + 1
  FROM search_hier sh,
       hier h
  WHERE h.vertex_from = sh.vertex_to
)
SELECT * FROM search_hier
ORDER BY depth, vertex_from, vertex_to;
```

В нерекурсивной части запроса выбираются ребра, ведущие из вершины иерархии в ближайшие вершины. Если бы здесь не было предложения WHERE, то формирование путей началось бы со всех уровней иерархии. Столбец depth означает «глубину» текущего уровня иерархии, то есть число ребер, отсчитанных от вершины.

Затем вступает в работу рекурсивная часть конструкции WITH. Из основной таблицы «Иерархия» (hier) выбираются строки (ребра), у которых номер исходной вершины vertex_from равен номеру целевой вершины vertex_to из строки временной рабочей таблицы search_hier, хранящей результаты прошлой итерации. Тем самым мы ищем ребро, которое является продолжением уже сформированного пути. Этот процесс продолжается до тех пор, пока он не дойдет до вершин-листьев, из которых не берут свое начало другие ребра.

Вот что выдает запрос:

```
 vertex_from | vertex_to | data | depth
-------------+-----------+------+-------
           1 |         2 |  4.7 |     1
           1 |         3 |  5.6 |     1
           2 |         4 |  6.3 |     2
           2 |         5 |  1.9 |     2
           3 |         6 |  3.5 |     2
           3 |         7 |  2.8 |     2
           3 |         8 |  4.1 |     2
           5 |         9 |  3.3 |     3
           5 |        10 |  4.5 |     3
           6 |        11 |  2.7 |     3
           6 |        12 |  1.3 |     3
           9 |        13 |  2.1 |     4
(12 строк)
```

Этот результат позволяет проследить ход процесса, но полные пути от какой-либо вершины до других вершин можно только мысленно реконструировать, переходя от одной строки выведенной таблицы к другой в соответствии с совпадающими значениями в столбцах vertex_to и vertex_from. Таким образом, чтобы «увидеть» полный путь от одной вершины к другой, надо в общем случае просмотреть более одной строки, каждая из которых содержит *только одно ребро* какого-либо пути.

Тем не менее существует возможность представить полные пути в компактной форме. Для этого можно воспользоваться массивами.

```
WITH RECURSIVE search_hier( vertex_from, vertex_to, data, depth, path ) AS
( SELECT h.vertex_from, h.vertex_to, h.data, 1,
         ARRAY[ h.vertex_from, h.vertex_to ]
  FROM hier h
  WHERE h.vertex_from = 1
  UNION ALL
  SELECT h.vertex_from, h.vertex_to, h.data, sh.depth + 1,
         sh.path || h.vertex_to
  FROM search_hier sh, hier h
  WHERE h.vertex_from = sh.vertex_to
)
SELECT * FROM search_hier
ORDER BY depth, vertex_from, vertex_to;
```

В нерекурсивной части общего табличного выражения начинается формирование пути в виде массива пройденных вершин, а в рекурсивной части к нему с помощью оператора || добавляется конечная вершина следующего ребра. Таким образом, путь формируется итеративно, ребро за ребром.

Нужно учитывать, что мы получили пути не только к листьям, но и ко всем промежуточным вершинам иерархии. Обратите внимание, что в каждой строке в полях vertex_from и vertex_to содержатся номера вершин последнего ребра в том пути, который представлен в поле path этой же строки.

```
 vertex_from | vertex_to | data | depth |     path
-------------+-----------+------+-------+--------------
           1 |         2 |  4.7 |     1 | {1,2}
           1 |         3 |  5.6 |     1 | {1,3}
           2 |         4 |  6.3 |     2 | {1,2,4}
           2 |         5 |  1.9 |     2 | {1,2,5}
           3 |         6 |  3.5 |     2 | {1,3,6}
           3 |         7 |  2.8 |     2 | {1,3,7}
           3 |         8 |  4.1 |     2 | {1,3,8}
           5 |         9 |  3.3 |     3 | {1,2,5,9}
           5 |        10 |  4.5 |     3 | {1,2,5,10}
           6 |        11 |  2.7 |     3 | {1,3,6,11}
           6 |        12 |  1.3 |     3 | {1,3,6,12}
           9 |        13 |  2.1 |     4 | {1,2,5,9,13}
(12 строк)
```

Если же нас интересуют пути только к вершинам-листьям, необходимо сначала сформировать все пути, как мы только что сделали, а затем отобрать лишь

те из них, которые ведут к листьям. Для этого необходимо уметь определять, является ли листом вершина из поля vertex_to конкретной строки. В качестве одного из способов можно каким-то образом заранее получить список всех вершин-листьев, а затем на завершающей стадии процесса, в главном запросе, воспользоваться этим списком для фильтрации результатов.

Напомним, что листом является такая вершина, из которой не берет начало никакое другое ребро. Если вершина представлена в столбце vertex_from, то она заведомо листом не является, поскольку дает начало как минимум одному ребру. Следовательно, нужно выбирать значения из столбца vertex_to, причем такие, которых нет в столбце vertex_from.

Выше мы уже рассматривали три варианта решения аналогичной задачи — определения вершины иерархии. Поэтому сейчас покажем только одно из возможных решений, а остальные предлагаем читателю реализовать самостоятельно.

```
SELECT vertex_to FROM hier
EXCEPT
SELECT vertex_from FROM hier
ORDER BY 1;
 vertex_to
-----------
         4
         7
         8
        10
        11
        12
        13
(7 строк)
```

Но куда можно поместить такой подзапрос, формирующий список вершин-листьев? Напрашивается ответ: в конструкцию WITH в виде общего табличного выражения.

Полученные в запросе строки отсортируем по значениям столбца path, что будет соответствовать порядку вывода «сначала в глубину» (depth-first).

Получаем в итоге следующий запрос:

```
WITH RECURSIVE search_hier( vertex_from, vertex_to, data, depth, path ) AS
( SELECT h.vertex_from, h.vertex_to, h.data, 1,
         ARRAY[ h.vertex_from, h.vertex_to ]
  FROM hier h
  WHERE h.vertex_from = 1
  UNION ALL
  SELECT h.vertex_from, h.vertex_to, h.data, sh.depth + 1,
         path || h.vertex_to
  FROM search_hier sh,
       hier h
  WHERE h.vertex_from = sh.vertex_to
),
leaves( leaf ) AS
( SELECT vertex_to FROM hier
  EXCEPT
  SELECT vertex_from FROM hier
)
SELECT sh.vertex_to AS leaf, sh.path
FROM search_hier AS sh
  JOIN leaves AS l ON l.leaf = sh.vertex_to
ORDER BY path;
```

```
 leaf |     path
------+--------------
    4 | {1,2,4}
   13 | {1,2,5,9,13}
   10 | {1,2,5,10}
   11 | {1,3,6,11}
   12 | {1,3,6,12}
    7 | {1,3,7}
    8 | {1,3,8}
(7 строк)
```

Для получения сортировки в порядке «сначала в ширину» (breadth-first) нужно в предложении ORDER BY указать столбец depth (глубина уровня иерархии). Поскольку на одном уровне иерархии может в общем случае оказаться несколько вершин-листьев, а порядок строк в рамках одного уровня по умолчанию не определен, то столбец path стоит сохранить, но задать его в качестве второго критерия сортировки.

```
...
SELECT sh.vertex_to AS leaf, sh.path
FROM search_hier AS sh JOIN leaves AS l
       ON l.leaf = sh.vertex_to
ORDER BY depth, path;
```

```
 leaf |     path
------+--------------
    4 | {1,2,4}
    7 | {1,3,7}
    8 | {1,3,8}
   10 | {1,2,5,10}
   11 | {1,3,6,11}
   12 | {1,3,6,12}
   13 | {1,2,5,9,13}
(7 строк)
```

Конечно, мы не обязаны ограничиваться выбором только одного из двух порядков сортировки — «сначала в глубину» или «сначала в ширину». Например, можно отсортировать строки по значениям столбца leaf. Правда, при выбранной нумерации вершин графа мы получили бы такой же порядок строк, что и в предыдущем случае. Чтобы результат отличался от только что полученного, давайте изменим номер одной из вершин, например 4, на другой, скажем 44:

```
UPDATE hier SET vertex_to = 44 WHERE vertex_to = 4;
UPDATE 1
...
SELECT sh.vertex_to AS leaf, sh.path
FROM search_hier AS sh
  JOIN leaves AS l ON l.leaf = sh.vertex_to
ORDER BY leaf;
 leaf |     path
------+--------------
    7 | {1,3,7}
    8 | {1,3,8}
   10 | {1,2,5,10}
   11 | {1,3,6,11}
   12 | {1,3,6,12}
   13 | {1,2,5,9,13}
   44 | {1,2,44}
(7 строк)
```

Детально вопрос обхода деревьев рассмотрен в подразделе документации 7.8.2.1 «Порядок поиска». Здесь сказано и о том, что PostgreSQL предлагает встроенные синтаксические конструкции для упорядочивания результатов запроса по принципу «сначала в глубину» или «сначала в ширину» — SEARCH DEPTH FIRST и SEARCH BREADTH FIRST соответственно. Обе эти конструкции рассмотрены в упражнении 8 (с. 94).

Поскольку эксперименты с иерархией еще не завершены, давайте вернем вершине 44 ее исходный номер.

```
UPDATE hier SET vertex_to = 4 WHERE vertex_to = 44;
UPDATE 1
```

Теперь мы готовы к тому, чтобы научиться выявлять нарушения структуры иерархии: присутствие циклов и наличие более одного пути от начала иерархии (корня дерева) к другим вершинам.

Для продолжения экспериментов сделаем в нашей иерархии два цикла: короткий и длинный. Назовем коротким такой цикл, в котором участвуют всего две вершины: например, в графе есть ребро (2, 5), а мы создадим ребро (5, 2). В длинном цикле участвуют более двух вершин. Для создания такого цикла добавим ребро (11, 3). В роли третьих полей обеих вставляемых строк выступают веса ребер, их значения выберем произвольно (рис. 2.12).

```
INSERT INTO hier VALUES ( 5, 2, 3.8 ), ( 11, 3, 7.4 );
INSERT 0 2
```

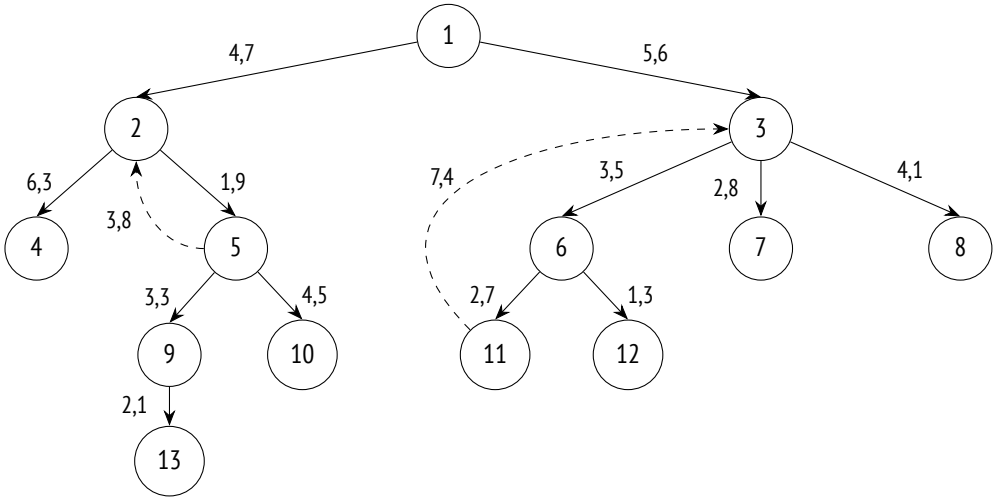


Рис. 2.12. *Иерархия с добавленными циклами*

Следующий запрос определяет, имеет ли место зацикливание путей, проверяя наличие целевой вершины очередного ребра иерархии в массиве уже пройденных вершин. В нерекурсивной части общего табличного выражения, как

и прежде, есть ограничение h.vertex_from = 1, благодаря которому все пути исходят из начала иерархии.

```
WITH RECURSIVE search_hier( vertex_from, vertex_to, data, depth, path, cycle ) AS
( SELECT h.vertex_from, h.vertex_to, h.data, 1,
         ARRAY[ h.vertex_from, h.vertex_to ],  -- path
         false                                 -- cycle
  FROM hier h
  WHERE h.vertex_from = 1
  UNION ALL
  SELECT h.vertex_from, h.vertex_to, h.data, sh.depth + 1,
         sh.path || h.vertex_to,
         h.vertex_to = ANY( sh.path )
  FROM search_hier sh, hier h
  WHERE h.vertex_from = sh.vertex_to
    AND NOT sh.cycle
)
SELECT * FROM search_hier
ORDER BY vertex_from, vertex_to;
```

```
 vertex_from | vertex_to | data | depth |     path     | cycle
-------------+-----------+------+-------+--------------+-------
           1 |         2 |  4.7 |     1 | {1,2}        | f
           1 |         3 |  5.6 |     1 | {1,3}        | f
           2 |         4 |  6.3 |     2 | {1,2,4}      | f
           2 |         5 |  1.9 |     2 | {1,2,5}      | f
           3 |         6 |  3.5 |     2 | {1,3,6}      | f
           3 |         7 |  2.8 |     2 | {1,3,7}      | f
           3 |         8 |  4.1 |     2 | {1,3,8}      | f
           5 |         2 |  3.8 |     3 | {1,2,5,2}    | t
           5 |         9 |  3.3 |     3 | {1,2,5,9}    | f
           5 |        10 |  4.5 |     3 | {1,2,5,10}   | f
           6 |        11 |  2.7 |     3 | {1,3,6,11}   | f
           6 |        12 |  1.3 |     3 | {1,3,6,12}   | f
           9 |        13 |  2.1 |     4 | {1,2,5,9,13} | f
          11 |         3 |  7.4 |     4 | {1,3,6,11,3} | t
(14 строк)
```

Как и предполагалось, искусственно созданные нами циклы были успешно выявлены.

Обратите внимание, что значение {1,2,5,2} представляет собой короткий цикл, а значение {1,3,6,11,3} — длинный. Важно, что в выборке нет строк, содержащих значения вида {1,2,5,2,5,2,5,2,...} или {1,3,6,11,3,6,11,3,6,11,...}, то есть процесс не зацикливается.

В предложении SELECT рекурсивной части общего табличного выражения столбец path фигурирует дважды. В первом случае к его значению добавляется вершина h.vertex_to для формирования нового значения пути, а во втором проверяется вхождение вершины h.vertex_to в его исходное значение:

```
sh.path || h.vertex_to,
h.vertex_to = ANY( sh.path )
```

Истинность условия h.vertex_to = ANY(sh.path) означает наличие цикла, однако текущая строка, сформированная в предложении SELECT и содержащая поле cycle, равное true, попадает как в следующую «версию» рабочей таблицы search_hier, так и в результирующую выборку. И лишь затем, уже на следующей итерации, такая строка отсеивается условием NOT sh.cycle в предложении WHERE. В результате мы получаем строки, соответствующие путям с циклами, но при этом в массиве path нет *многократного* повторения вершин, образующих цикл.

Детально вопрос выявления циклов при обходе деревьев рассмотрен в подразделе документации 7.8.2.2 «Выявление циклов». Здесь сказано и о том, что для решения данной задачи PostgreSQL предлагает синтаксическую конструкцию CYCLE ... SET ... USING .... Она рассмотрена в упражнении 9 (с. 95).

2.3.2. Выявление множественных путей

Теперь рассмотрим запрос, выявляющий еще один дефект иерархии — наличие более одного пути от начала иерархии до какой-либо ее вершины.

Перед тем как продолжить эксперименты, нужно восстановить исходные данные в таблице «Иерархия» (hier), чтобы избавиться от циклов, а затем добавить ребро, чтобы из вершины 2 в вершину 9 возник еще один путь. Обратите внимание на использование конструкции ROW в команде DELETE. Это конструктор табличной строки, или составного значения. Если в составном значении более одного элемента (как в нашем примере), то можно опустить ключевое слово ROW, оставив только скобки. Более подробно об этом написано в подразделе документации 4.2.13 «Конструкторы табличных строк».

```
DELETE FROM hier WHERE ROW( vertex_from, vertex_to ) IN ( ROW( 5, 2 ), ROW( 11, 3 ) );
DELETE 2
```

```
INSERT INTO hier VALUES ( 4, 9, 5.7 );
INSERT 0 1
```

Снова выполним предыдущий запрос. Циклов в нашей иерархии теперь нет, но мы *увидим*, что между вершинами 2 и 9 есть два разных пути: {2,4,9} и {2,5,9}. Поскольку формируются не только пути от начала иерархии до вершин-листьев, но и все промежуточные пути, исходящие из ее начала, то каждая из этих комбинаций вершин встречается более одного раза.

```
 vertex_from | vertex_to | data | depth |     path     | cycle
-------------+-----------+------+-------+--------------+-------
           1 |         2 |  4.7 |     1 | {1,2}        | f
           1 |         3 |  5.6 |     1 | {1,3}        | f
           2 |         4 |  6.3 |     2 | {1,2,4}      | f
           2 |         5 |  1.9 |     2 | {1,2,5}      | f
           3 |         6 |  3.5 |     2 | {1,3,6}      | f
           3 |         7 |  2.8 |     2 | {1,3,7}      | f
           3 |         8 |  4.1 |     2 | {1,3,8}      | f
           4 |         9 |  5.7 |     3 | {1,2,4,9}    | f
           5 |         9 |  3.3 |     3 | {1,2,5,9}    | f
           5 |        10 |  4.5 |     3 | {1,2,5,10}   | f
           6 |        11 |  2.7 |     3 | {1,3,6,11}   | f
           6 |        12 |  1.3 |     3 | {1,3,6,12}   | f
           9 |        13 |  2.1 |     4 | {1,2,4,9,13} | f
           9 |        13 |  2.1 |     4 | {1,2,5,9,13} | f
(14 строк)
```

Решение можно построить на основе ранее рассмотренного запроса, который формирует все пути, исходящие из начала иерархии:

```
WITH RECURSIVE search_hier( vertex_from, vertex_to, data, depth, path ) AS
( SELECT h.vertex_from, h.vertex_to, h.data, 1,
         ARRAY[ h.vertex_from, h.vertex_to ]
  FROM hier h
  WHERE h.vertex_from = 1
  UNION ALL
  SELECT h.vertex_from, h.vertex_to, h.data, sh.depth + 1,
         sh.path || h.vertex_to
  FROM search_hier sh,
       hier h
  WHERE h.vertex_from = sh.vertex_to
)
SELECT * FROM search_hier
ORDER BY depth, vertex_from, vertex_to;
```

В этом запросе в общем табличном выражении search_hier значение столбца vertex_to содержит номер последней вершины того пути, который записан в виде массива в столбце path. Для выявления неуникальных путей достаточно учитывать лишь эту конечную вершину, поскольку начальные вершины всех путей совпадают.

Порядок действий предлагается следующий.

1. Сформируем все пути, исходящие из начала иерархии.
2. Отберем те вершины, в которых заканчивается более одного пути. Для этого сгруппируем строки по значениям столбца vertex_to и воспользуемся предложением HAVING.
3. Из общего множества путей выберем лишь те, которые завершаются в этих вершинах.

Первый шаг будет выполняться в общем табличном выражении, как в запросе-прототипе. Для реализации второго шага надо включить в запрос еще одно общее табличное выражение. Завершающий шаг реализуем в главном запросе:

```
WITH RECURSIVE search_hier( vertex_from, vertex_to, data, depth, path ) AS
( SELECT h.vertex_from, h.vertex_to, h.data, 1,
         ARRAY[ h.vertex_from, h.vertex_to ]
  FROM hier h
  WHERE h.vertex_from = 1
  UNION ALL
  SELECT h.vertex_from, h.vertex_to, h.data, sh.depth + 1,
         sh.path || h.vertex_to
  FROM search_hier sh,
       hier h
  WHERE h.vertex_from = sh.vertex_to
),
nonunique_paths ( vertex_to ) AS
( SELECT vertex_to
  FROM search_hier
  GROUP BY vertex_to
  HAVING count( * ) > 1
)
SELECT nup.vertex_to, sh.path
FROM nonunique_paths nup
  JOIN search_hier sh ON sh.vertex_to = nup.vertex_to
ORDER BY nup.vertex_to, sh.path;
```

Рекурсивный подзапрос в конструкции WITH не изменился. В подзапросе nonunique_paths мы отбираем те вершины, в которые из начала иерархии существует более одного пути. Обратите внимание, что использование функции count в предложении HAVING не требует включения этой функции в список SELECT. В главном запросе выполняется эквисоединение двух временных таблиц, чтобы получить для каждой из отобранных вершин все варианты полных путей из начала иерархии.

В этом запросе выбираются все пути, исходящие из начала иерархии, включая промежуточные, поэтому возможны повторы комбинаций вершин в более длинных путях.

```
 vertex_to |     path
-----------+--------------
         9 | {1,2,4,9}
         9 | {1,2,5,9}
        13 | {1,2,4,9,13}
        13 | {1,2,5,9,13}
(4 строки)
```

До сих пор мы никак не использовали значения из столбца data, хотя он фигурировал в каждом запросе. Давайте сейчас воспользуемся им для выбора одного из нескольких найденных путей. Будем рассматривать значения столбца data в качестве условной стоимости каждого ребра иерархии. Таким образом, сумма стоимостей всех ребер, образующих путь, определяет стоимость этого пути.

Для расчета стоимости достаточно в подзапрос search_hier добавить столбец total_value, в котором будут накапливаться значения поля data для каждого ребра, составляющего путь.

Приступая к работе с иерархиями, мы исходили из того, что имеем дело с существующей системой, которая может иметь ряд отклонений от правильной иерархической структуры. Одним из них, как мы видели, могут быть неуникальные пути от вершины иерархии к другим вершинам. Вычислив самый «правильный» путь (наиболее дорогой или, наоборот, дешевый), мы можем устранить остальные неуникальные пути, приблизив тем самым структуру системы к строгой иерархии.

```
WITH RECURSIVE search_hier( vertex_from, vertex_to, data, depth, path, total_value ) AS
( SELECT h.vertex_from, h.vertex_to, h.data, 1,
         ARRAY[ h.vertex_from, h.vertex_to ], h.data
  FROM hier h
  WHERE h.vertex_from = 1
  UNION ALL
  SELECT h.vertex_from, h.vertex_to, h.data, sh.depth + 1,
         sh.path || h.vertex_to, sh.total_value + h.data
  FROM search_hier sh,
       hier h
  WHERE h.vertex_from = sh.vertex_to
),
nonunique_paths ( vertex_to ) AS
( SELECT vertex_to
  FROM search_hier
  GROUP BY vertex_to
  HAVING count( * ) > 1
)
SELECT nup.vertex_to, sh.path, sh.total_value
FROM nonunique_paths nup
  JOIN search_hier sh ON sh.vertex_to = nup.vertex_to
ORDER BY nup.vertex_to, sh.path;
```

Получаем такой результат:

```
 vertex_to |     path     | total_value
-----------+--------------+-------------
         9 | {1,2,4,9}    |        16.7
         9 | {1,2,5,9}    |         9.9
        13 | {1,2,4,9,13} |        18.8
        13 | {1,2,5,9,13} |        12.0
(4 строки)
```

Конкретный путь выбираем, исходя из критерия выбора: самый «дорогой» или самый «дешевый».

2.3.3. Поиск маршрута между двумя городами

В завершение этого раздела отойдем от деревьев и иерархий и рассмотрим работу с графом более общего вида. В качестве примера возьмем поиск маршрута между двумя городами, между которыми, возможно, нет прямого сообщения. Но даже если оно существует, у пассажира может возникнуть необходимость

в маршруте с пересадками, если, например, на более удобные рейсы закончились билеты.

Обратите внимание, что для каждой пары городов, между которыми установлено сообщение, в таблице (точнее говоря, представлении) «Маршруты» (`routes`) находятся две строки: для рейсов в прямом и в обратном направлениях. Таким образом, мы имеем дело с ориентированным графом, причем для каждой пары вершин a и b существует два ребра: (a, b) и (b, a).

Важно, что мы ищем только маршрут, то есть определяем принципиальную возможность перемещения между городами, а не подбираем конкретные рейсы с учетом их стыковок по времени вылета.

Представим стратегию поиска, которая позволит выбрать не только самый короткий (по числу перелетов) маршрут, но и более длинные маршруты.

1. Сформировать маршруты из исходного города в те города, в которые из него выполняются прямые рейсы.

2. Удлинить формируемые маршруты на один перелет, добавив к текущему конечному пункту те города, в которые есть прямые рейсы из него.

   Если текущий конечный пункт какого-либо маршрута (до его удлинения) является целевым городом, то прокладывать дальнейший маршрут из него уже не нужно, однако необходимо записать полученный маршрут в результирующий список маршрутов.

   В составе каждого формируемого маршрута не должно быть повторяющихся городов — циклы недопустимы.

   Процесс удлинения продолжается, пока не достигнута заданная предельная длина маршрута (число перелетов между городами).

3. Выбрать из сформированных маршрутов только те, в которых конечной точкой является целевой город.

Давайте проложим маршрут из Хабаровска в Сочи. При этом будем ограничивать предельное число перелетов, скажем, четырьмя. Алгоритм не изменится, если число перелетов будет и бóльшим, но негуманно заставлять пассажира совершать так много пересадок.

Запрос будет таким:

```
WITH RECURSIVE search_route( city_from, city_to, transfers, route ) AS
( SELECT DISTINCT ON ( arrival_city )
         departure_city, arrival_city, 1,
         ARRAY[ departure_city, arrival_city ]
  FROM routes
  WHERE departure_city = 'Хабаровск'
  UNION ALL
  SELECT DISTINCT ON ( sr.route || r.arrival_city )
         r.departure_city, r.arrival_city, transfers + 1,
         sr.route || r.arrival_city
  FROM search_route AS sr
    JOIN routes AS r ON r.departure_city = sr.city_to
  WHERE sr.city_to <> 'Сочи'
    AND sr.transfers <= 3
    AND r.arrival_city <> ALL( sr.route )
)
SELECT transfers AS "Число перелетов", array_to_string( route, ' - ' ) AS "Маршрут"
FROM search_route
WHERE city_to = 'Сочи'
ORDER BY transfers, route;
```

В нем столбец `city_from` означает город отправления, столбец `city_to` — город назначения, столбец `transfers` показывает число перелетов, а столбец `route` — формируемый маршрут, представленный в виде массива названий городов.

Первый шаг алгоритма реализуется в нерекурсивной части общего табличного выражения, второй — в его рекурсивной части, а последний — в главном запросе.

В процессе вычисления рекурсивного общего табличного выражения рабочая таблица — она имеет имя `search_route` — содержит строки, то есть маршруты, доступные на текущей итерации. После завершения его вычисления под этим же именем доступна результирующая таблица. Она используется уже в главном запросе.

Для удлинения маршрутов рабочая таблица `search_route` соединяется с представлением «Маршруты» (routes) по условию `r.departure_city = sr.city_to`. Это означает, что город отправления на текущей итерации будет совпадать с городом прибытия предыдущей итерации.

В подзапросах используется предложение `DISTINCT ON`, поскольку по целому ряду направлений выполняется более одного регулярного рейса. Например,

по направлениям Сочи — Москва и Новый Уренгой — Москва выполняется по два рейса. Если в каком-то маршруте окажутся все названные города, то при отсутствии предложения `DISTINCT ON` такой маршрут будет сформирован не один раз, а четыре. Чтобы увидеть влияние предложения `DISTINCT ON` на результаты выполнения запроса, можно удалить его и повторить запрос. Обратите внимание, что в данном случае не используется предложение `ORDER BY`. При совпадении в нескольких строках значений выражений, включенных в предложение `DISTINCT ON`, эти строки будут полностью идентичными. Следовательно, можно оставить в выборке любую из них, поэтому сортировка и не нужна.

Строки-маршруты, сформированные на текущей итерации на основе рабочей таблицы, поступают в промежуточную и результирующую таблицы. Затем содержимое рабочей таблицы заменяется содержимым промежуточной таблицы для новой итерации, а промежуточная таблица очищается. Процесс повторяется до тех пор, пока рабочая таблица перед выполнением очередной итерации не окажется пустой. Это в нашем запросе рано или поздно произойдет. Критерием завершения итерационного процесса является достижение заданного предельного числа перелетов, за что отвечает условие `sr.transfers <= 3` в предложении `WHERE` рекурсивной части общего табличного выражения. Когда на очередной итерации значение столбца `sr.transfers` станет равным четырем, в выборке не окажется ни одной строки, поэтому промежуточная таблица останется пустой и, как следствие, станет пустой рабочая таблица — на этом процесс и завершится.

Обратите внимание, что мы планировали ограничивать предельное число перелетов четырьмя, но условие записано как `sr.transfers <= 3`, поскольку в списке `SELECT` число перелетов увеличивается на единицу: `transfers + 1`. Конечно, некоторые строки будут выбывать из состава рабочей таблицы и до срабатывания этого условия. Это строки с маршрутами, в которых текущим конечным пунктом является целевой город Сочи. За это отвечает условие `sr.city_to <> 'Сочи'`. Ведь если в процессе формирования какого-то маршрута мы уже добрались до целевого города, то продлевать маршрут дальше нет смысла.

Условие `r.arrival_city <> ALL( sr.route )` отвечает за отсутствие циклов: из городов, в которые можно улететь из текущей конечной точки, исключаются города, уже включенные в формируемый маршрут.

В процессе вычисления общего табличного выражения в результирующей таблице оказываются маршруты, ведущие не только в город назначения, но и в другие города. Поэтому в предложении WHERE главного запроса необходимо условие city_to = 'Сочи', позволяющее отобрать только нужные маршруты.

Обратите внимание, что в главном запросе в предложении ORDER BY на втором месте указан массив городов маршрута. Что в этом особенного? Во-первых, это *входной* столбец: он является столбцом таблицы search_route из предложения FROM, но не присутствует в списке SELECT в таком же виде: там он фигурирует в качестве параметра функции, преобразующей массив в строку (различия между входными и выходными столбцами показаны в документации в описаниях предложений GROUP BY и ORDER BY команды SELECT). Во-вторых, сортировка по столбцу-массиву выполняется корректно, давая требуемый порядок строк.

Выполнение запроса показывает, что минимальное число перелетов равно двум:

```
 Число перелетов |                          Маршрут
-----------------+-------------------------------------------------------------
               2 | Хабаровск - Москва - Сочи
               3 | Хабаровск - Анадырь - Москва - Сочи
 ...
               3 | Хабаровск - Москва - Белгород - Сочи
               3 | Хабаровск - Москва - Красноярск - Сочи
 ...
               3 | Хабаровск - Усть-Илимск - Красноярск - Сочи
 ...
               4 | Хабаровск - Анадырь - Москва - Красноярск - Сочи
 ...
               4 | Хабаровск - Санкт-Петербург - Оренбург - Москва - Сочи
 ...
               4 | Хабаровск - Усть-Илимск - Красноярск - Новокузнецк - Сочи
 ...
(91 строка)
```

Поскольку в этом запросе не предусмотрено никаких ограничений на выбор смежных городов, то в выборку попадают в том числе не самые рациональные маршруты с перелетами в направлениях, очень далеких от направления на город назначения. Это, например, перелеты Хабаровск — Анадырь, Санкт-Петербург — Оренбург и Москва — Красноярск. Для исправления ситуации и уменьшения числа вариантов можно исключать некоторые города из списка допустимых.

Этот пример можно развить, назначая конкретные числовые оценки всем отобранным маршрутам, например, на основе суммарной продолжительности полетов или с учетом времени ожидания между перелетами. Оставляем реализацию этой и других идей читателю.

Также можно значительно ускорить выполнение запроса. Один из возможных способов рассмотрен в упражнении 11 (с. 98).

2.4. Модификация данных в общем табличном выражении

До сих пор в общих табличных выражениях мы ограничивались только командами SELECT. Однако в эти выражения можно включать и команды изменения данных: вставку, обновление и удаление строк.

Давайте в качестве примера рассмотрим периодическое перемещение части записей из оперативных таблиц в архивные. Это позволяет уменьшить размер данных, с которыми происходит активная работа, и ускорить запросы к ним. Конечно, число записей в базе данных «Авиаперевозки» не исчисляется миллиардами, тем не менее решение такой задачи можно показать и на ее примере. Самые большие по числу записей таблицы — это «Перелеты» (ticket_flights), «Посадочные талоны» (boarding_passes), «Билеты» (tickets), «Бронирования» (bookings) и «Рейсы» (flights).

Можно выбирать различные стратегии перемещения данных из оперативных таблиц в архивные, учитывая специфику предметной области, размеры таблиц и скорость приращения числа записей. В нашем случае в рамках одной операции бронирования может быть оформлено более одного билета, а в одном билете может быть записано более одного перелета, которые могут выполняться в разные даты. Мы изберем такую стратегию:

- Из оперативных таблиц «Перелеты» (ticket_flights), «Посадочные талоны» (boarding_passes) и «Рейсы» (flights) будем переносить в архивные таблицы записи, относящиеся к рейсам, «возраст» которых не менее тридцати дней. Очевидно, что речь идет о рейсах, которые уже завершены или отменены. Важно, что мы не допустим ситуацию, когда из оперативной таблицы

«Посадочные талоны» (boarding_passes) в архивную перенесена лишь часть записей, относящихся к конкретному рейсу. Однако при этом возможна ситуация, когда часть записей о перелетах, относящихся к одному и тому же билету, останется в оперативной таблице «Перелеты» (ticket_flights), а другая часть будет перенесена в архивную таблицу.

- Из оперативной таблицы «Билеты» (tickets) будем переносить в архивную таблицу те строки, у которых нет связанных строк в таблице «Перелеты» (ticket_flights). Проще говоря, это билеты, оставшиеся без перелетов. С таблицей «Бронирования» (bookings) поступим аналогично: будем переносить в архивную таблицу те строки, у которых нет связанных строк в таблице «Билеты» (tickets), то есть бронирования, оставшиеся без билетов.

Описанные операции с таблицами можно повторять, скажем, один раз в день.

При выбранной стратегии сохраняется возможность получить полную стоимость даже тех бронирований, часть записей о перелетах которых уже перенесена в архив. Это возможно потому, что в таблице «Бронирования» (bookings) есть столбец total_amount, содержащий полную стоимость бронирования, которая складывается из стоимостей всех входящих в него перелетов. Это пример контролируемой избыточности в базе данных.

Прежде всего создадим архивные таблицы. Заполнять данными их, конечно, не будем. Создавать первичные и внешние ключи также не станем, поскольку данные, которые будут вводиться в эти таблицы, не будут согласованными: связанные данные могут разрываться между оперативными и архивными таблицами. Например, в архив может быть перенесена лишь часть записей о перелетах, оформленных в одном билете.

```
CREATE TABLE flights_arch AS
SELECT * FROM flights WITH NO DATA;
```

```
CREATE TABLE AS
```

```
CREATE TABLE bookings_arch AS
SELECT * FROM bookings WITH NO DATA;
```

```
CREATE TABLE AS
```

```
CREATE TABLE tickets_arch AS
SELECT * FROM tickets WITH NO DATA;
```

```
CREATE TABLE AS
```

```
CREATE TABLE ticket_flights_arch AS
SELECT * FROM ticket_flights WITH NO DATA;
```

```
CREATE TABLE AS
```

```
CREATE TABLE boarding_passes_arch AS
SELECT * FROM boarding_passes WITH NO DATA;
```

```
CREATE TABLE AS
```

Теперь перейдем к реализации предложенной стратегии. Сначала решим задачу с помощью отдельных SQL-команд для вставки строк в архивные таблицы и удаления из оперативных таблиц. Принимая во внимание наличие внешних ключей в таблицах, нам придется сначала выбрать строки из оперативных таблиц «Рейсы» (flights), «Перелеты» (ticket_flights) и «Посадочные талоны» (boarding_passes) и вставить их в архивные таблицы flights_arch, ticket_flights_arch и boarding_passes_arch, а затем удалить эти же строки из оперативных таблиц «Посадочные талоны» (boarding_passes), «Перелеты» (ticket_flights) и «Рейсы» (flights) — в таком порядке.

Все операции выполним в рамках одной транзакции, чтобы можно было, отменяя ее в самом конце работы, провести эксперимент несколько раз.

Переносим в архив историю рейсов, совершенных не менее тридцати дней назад:

```
BEGIN;
```

```
BEGIN
```

```
INSERT INTO flights_arch
SELECT * FROM flights
WHERE bookings.now()::date - scheduled_departure::date >= 30;
```

```
INSERT 0 550
```

```
INSERT INTO ticket_flights_arch
SELECT tf.*
FROM ticket_flights AS tf, flights AS f
WHERE bookings.now()::date - f.scheduled_departure::date >= 30
  AND tf.flight_id = f.flight_id;
```

```
INSERT 0 8163
```

```
INSERT INTO boarding_passes_arch
SELECT bp.*
FROM boarding_passes AS bp, flights AS f
WHERE bookings.now()::date - f.scheduled_departure::date >= 30
  AND bp.flight_id = f.flight_id;
```

```
INSERT 0 8163
```

В следующих далее командах для отбора удаляемых строк используется предложение USING. Как сказано в описании команды DELETE, приведенном в документации (см. раздел «Замечания»), при этом, по сути, целевая таблица соединяется с таблицами, перечисленными в предложении USING; для удаления отбираются строки целевой таблицы, вошедшие в соединение.

В предложении USING команды удаления строк из таблицы «Посадочные талоны» (boarding_passes) указана таблица «Рейсы» (flights). Хотя в таблице boarding_passes составной внешний ключ (ticket_no, flight_id) ссылается на таблицу «Перелеты» (ticket_flights), но внешний ключ flight_id этой таблицы, в свою очередь, ссылается на первичный ключ flight_id таблицы «Рейсы» (flights). Таким образом, мы исключили ненужное соединение с таблицей «Перелеты» (ticket_flights).

В командах используется функция bookings.now, возвращающая момент, который считается «текущим» в ретроспективе. По сути, он является константой, поэтому при выполнении ряда экспериментов это значение можно каждый раз наращивать на один день, имитируя ход времени.

```
DELETE FROM boarding_passes AS bp
USING flights AS f
WHERE bookings.now()::date - f.scheduled_departure::date >= 30
  AND bp.flight_id = f.flight_id;
```

```
DELETE 8163
```

```
DELETE FROM ticket_flights AS tf
USING flights AS f
WHERE bookings.now()::date - f.scheduled_departure::date >= 30
  AND tf.flight_id = f.flight_id;
```

```
DELETE 8163
```

А эта команда выполняется значительно дольше предыдущих.

```
DELETE FROM flights
WHERE bookings.now()::date - scheduled_departure::date >= 30;
```

```
DELETE 550
```

```
ROLLBACK;
```

```
ROLLBACK
```

Общие табличные выражения значительно упрощают весь процесс: все приведенные операции объединяются в один SQL-запрос. Работу выполним также в рамках транзакции.

```
BEGIN;
BEGIN
WITH deleted_f AS
( DELETE FROM flights
  WHERE bookings.now()::date - scheduled_departure::date >= 30
  RETURNING *
),
deleted_tf AS
( DELETE FROM ticket_flights AS tf
  USING deleted_f AS df
  WHERE tf.flight_id = df.flight_id
  RETURNING tf.*    -- попробуйте оставить здесь только *
),
deleted_bp AS
( DELETE FROM boarding_passes AS bp
  USING deleted_f AS df
  WHERE bp.flight_id = df.flight_id
  RETURNING bp.*    -- попробуйте оставить здесь только *
),
inserted_bp AS
( INSERT INTO boarding_passes_arch
  SELECT * FROM deleted_bp
),
inserted_tf AS
( INSERT INTO ticket_flights_arch
  SELECT * FROM deleted_tf
)
INSERT INTO flights_arch
SELECT * FROM deleted_f;
INSERT 0 550
```

Как мы уже знаем, сообщение, выведенное в результате выполнения запроса, содержащего общие табличные выражения, относится к главному запросу. В нашем примере это запрос, который вставляет строки в таблицу flights_arch.

Давайте посмотрим, сколько строк было вставлено в таблицы boarding_passes_arch и ticket_flights_arch, а потом завершим транзакцию.

```
SELECT count( * ) FROM boarding_passes_arch;
 count
-------
  8163
(1 строка)
```

```
SELECT count( * ) FROM ticket_flights_arch;
 count
-------
  8163
(1 строка)
ROLLBACK;
ROLLBACK
```

Эти результаты совпадают с теми, что были получены в первом решении. Можно сделать и другие выборки как из архивных, так и из основных таблиц, чтобы убедиться, что запрос работает корректно. Но он выполняется значительно дольше, чем можно было ожидать. Один из способов ускорения рассматривается в упражнении 18 (с. 106).

Возникает вопрос: каким образом строки, удаленные из основных таблиц, оказываются в архивных, если в запросе нет операторов, выбирающих строки из основных таблиц? В запросе не видно фрагментов, подобных следующему:

```
INSERT INTO flights_arch
SELECT * FROM flights WHERE ...;
```

Дело в предложении RETURNING, которое добавляется к операторам DELETE в общих табличных выражениях. Обычно оператор DELETE только удаляет строки, но с предложением RETURNING он также возвращает — как SELECT — строки, сформированные на основе удаленных. Как сказано в подразделе документации 7.8.4 «Изменение данных в WITH», из результата RETURNING создается временная таблица с именем общего табличного выражения. Обращаясь к ней, другие общие табличные выражения и главный запрос могут получить доступ к результатам модификации данных. Однако они не могут этого сделать, обращаясь непосредственно к самой модифицированной таблице.

Сразу скажем, что эта временная таблица создается только на время выполнения запроса и (при небольшом размере) располагается в оперативной памяти сервера. Этим она отличается от «настоящей» временной таблицы, которая создается командой CREATE TEMP TABLE.

Для оператора INSERT ... RETURNING временная таблица будет сформирована на основе вставленных строк, а для оператора UPDATE ... RETURNING — на основе обновленных. Конечно, возможны ситуации, когда формально правильная команда вставки, обновления или удаления фактически не вставит (не обновит

или не удалит) ни одной строки из-за того, что они не удовлетворяют условию. В этом случае временная таблица будет пустой. Напомним, что даже в команде INSERT вставляемые строки можно подготовить с помощью подзапроса, который в общем случае может не вернуть ни одной строки.

В предложении RETURNING можно задать конкретные имена столбцов или выражения, сформированные на их основе. Если же нужны все столбцы в их исходном виде, просто пишется символ *.

В подзапросах deleted_tf и deleted_bp, содержащих команды удаления, для отбора удаляемых строк используется предложение USING. В нем задана не постоянная таблица, а временная — deleted_f, сформированная одноименным подзапросом. Она содержит строки, только что удаленные из таблицы «Рейсы» (flights). Надо учитывать, что в данном случае при использовании предложения USING приходится в предложении RETURNING дополнять символ * псевдонимом временной таблицы. Если этого не сделать, мы получим в результате столбцы обеих таблиц, целевой и вспомогательной, что приведет к ошибке:

```
...
deleted_tf AS
 ( DELETE FROM ticket_flights AS tf
  USING deleted_f AS df
  WHERE tf.flight_id = df.flight_id
  RETURNING *    -- обратите внимание!
),
...
inserted_tf AS
(
  INSERT INTO ticket_flights_arc
  SELECT * FROM deleted_tf    -- ошибка здесь
)
...
ОШИБКА:  INSERT содержит больше выражений, чем целевых столбцов
СТРОКА 24:   SELECT * FROM deleted_tf
                    ^
```

А что касается подзапроса deleted_f, то, поскольку нас интересуют только значения столбца flight_id, в нем можно было написать RETURNING flight_id.

Обратите внимание: в командах INSERT подзапросов inserted_bp и inserted_tf нет предложения RETURNING, так как дальнейшая обработка строк, только что вставленных в архивные таблицы boarding_passes_arch и ticket_flights_arch, не предполагается.

Порядок следования подзапросов deleted_f, deleted_tf и deleted_bp, казалось бы, вступает в противоречие с внешними ключами таблиц «Посадочные талоны» (boarding_passes) и «Перелеты» (ticket_flights). Однако запрос работает корректно, поскольку ограничения внешних ключей проверяются после выполнения *всей команды*, а не после вычисления каждого общего табличного выражения или удаления (вставки, обновления) каждой строки (см. описание команды CREATE TABLE в документации). Таким образом, в процессе выполнения рассматриваемого запроса возможна ситуация, когда, например, уже удалены строки из таблицы «Рейсы» (flights), а ссылающиеся на них строки таблицы «Перелеты» (ticket_flights) еще не удалены. Заметим также, что если бы внешние ключи таблиц, задействованных в нашем эксперименте, имели предложение ON DELETE CASCADE, то запрос не решал бы поставленную задачу, пришлось бы его модифицировать.

Быстродействие двух реализаций примерно одинаково. Может возникнуть закономерный вопрос: в чем тогда заключается преимущество запроса с общими табличными выражениями по сравнению с выполнением нескольких команд? На уровне изоляции Read Committed использование нескольких команд может привести к переносу в архив несогласованных данных, поскольку на этом уровне изоляции снимок создается перед выполнением каждого запроса транзакции, а параллельные транзакции могут обновить эти же данные. В результате, например, в архив будут перенесены данные из таблицы «Перелеты» (ticket_flights) в том состоянии, которое они имели до изменения, а из таблицы «Посадочные талоны» (boarding_passes) — уже после него. Конечно, в нашем случае риск возникновения такой ситуации невелик, поскольку вероятность изменения записей, относящихся к уже выполненным рейсам, мала, но исключать его полностью все же нельзя.

Итак, мы реализовали первую часть стратегии архивирования ряда таблиц. Вторую часть — архивирование таблиц «Билеты» (tickets) и «Бронирования» (bookings) — обсудим в упражнении 19 (с. 107).

Особенность рассмотренной ситуации была в том, что все операции выполнялись над разными таблицами. А какие сложности могут возникнуть, если таблица будет одна и та же?

Давайте предположим, что с целью повышения качества обслуживания пассажиров рассматривается возможность изменения расположения кресел в салоне

самолета «Сухой Суперджет-100». Предлагается увеличить число мест бизнес-класса за счет сокращения числа мест эконом-класса. В настоящее время компоновка салона такова:

```
SELECT * FROM seats
WHERE aircraft_code = 'SU9';
 aircraft_code | seat_no | fare_conditions
---------------+---------+-----------------
 ...
 SU9           | 2F      | Business
 SU9           | 3A      | Business
 SU9           | 3C      | Business
 SU9           | 3D      | Business
 SU9           | 3F      | Business
 SU9           | 4A      | Economy
 SU9           | 4C      | Economy
 SU9           | 4D      | Economy
 SU9           | 4E      | Economy
 SU9           | 4F      | Economy
 SU9           | 5A      | Economy
 ...
 SU9           | 20F     | Economy
(97 строк)
```

Поскольку нам предстоит провести несколько экспериментов, будем выполнять все операции в рамках транзакции, чтобы таблица «Места» (seats) всегда оставалась в исходном состоянии.

Итак, поместим в конструкцию WITH операторы для удаления и вставки строк. Удалим два первых ряда кресел в салоне эконом-класса, а после последнего ряда кресел в салоне бизнес-класса добавим еще один ряд. С целью упрощения SQL-запроса зададим номера этих рядов явным образом, просто посмотрев на результат предыдущей выборки из таблицы «Места» (seats). Конечно, можно получить номера рядов с помощью подзапросов, чтобы сделать решение универсальным, пригодным для работы с другими моделями самолетов, но мы оставим это читателю.

Одному ряду кресел соответствует несколько строк в таблице «Места» (seats), поэтому в подзапросе inserted_seats сформируем новый ряд кресел, выбрав из этой же таблицы строки последнего ряда в салоне бизнес-класса. С помощью функции overlay заменим для добавленных кресел номер ряда с 3-го на 4-й.

```
BEGIN;
BEGIN
WITH deleted_seats AS
( DELETE FROM seats
  WHERE aircraft_code = 'SU9'
    AND left( seat_no, 1 ) IN ( '4', '5' )
),
inserted_seats AS
( INSERT INTO seats
  SELECT aircraft_code, overlay( seat_no PLACING '4' FROM 1 FOR 1 ), fare_conditions
  FROM seats
  WHERE aircraft_code = 'SU9'
    AND left( seat_no, 1 ) = '3'
)
SELECT count( * )
FROM seats
WHERE aircraft_code = 'SU9';
ОШИБКА:  повторяющееся значение ключа нарушает ограничение уникальности "seats_pkey"
ПОДРОБНОСТИ:  Ключ "(aircraft_code, seat_no)=(SU9, 4A)" уже существует.
ROLLBACK;
ROLLBACK
```

Запрос завершился ошибкой дублирования значения первичного ключа. Попробуем добавить к номерам новых кресел пометку +, оставив номер ряда без изменений:

```
BEGIN;
BEGIN
WITH deleted_seats AS
( DELETE FROM seats
  WHERE aircraft_code = 'SU9'
    AND left( seat_no, 1 ) IN ( '4', '5' )
),
inserted_seats AS
( INSERT INTO seats
  SELECT aircraft_code, seat_no || '+', fare_conditions
  FROM seats
  WHERE aircraft_code = 'SU9'
    AND left( seat_no, 1 ) = '3'
)
SELECT count( * )
FROM seats
WHERE aircraft_code = 'SU9';
```

Запрос выдает старое значение числа кресел:

```
 count
-------
    97
(1 строка)
```

А если в этой же транзакции выполнить еще один запрос к таблице «Места (seats)», чтобы все-таки выяснить, произошли изменения в таблице или нет? Видим, что изменения в таблицу успешно внесены:

```
 aircraft_code | seat_no | fare_conditions
---------------+---------+-----------------
 ...
 SU9           | 3A      | Business
 SU9           | 3A+     | Business
 SU9           | 3C      | Business
 SU9           | 3C+     | Business
 SU9           | 3D      | Business
 SU9           | 3D+     | Business
 SU9           | 3F      | Business
 SU9           | 3F+     | Business
 SU9           | 6A      | Economy
 SU9           | 6C      | Economy
 SU9           | 6D      | Economy
 SU9           | 6E      | Economy
 SU9           | 6F      | Economy
 ...
(91 строка)
ROLLBACK;
ROLLBACK
```

Но почему же предыдущий запрос не показал новое число строк в таблице? В подразделе документации 7.8.4 «Изменение данных в WITH» сказано, что все операторы в конструкции WITH и главный запрос выполняются параллельно, при этом все они используют один и тот же снимок данных. Это понятие широко используется при обсуждении механизмов выполнения транзакций на различных уровнях изоляции (подробно эти вопросы рассматриваются в главе документации 13 «Управление конкурентным доступом»). Использование одного и того же снимка данных приводит к тому, что операторы в конструкции WITH и главный запрос не видят изменения, выполненные друг другом. Поэтому запрос и показывал исходное число строк, такое же, какое было в таблице до удаления и вставки, выполненных этим же запросом.

Важно также и то, что порядок, в котором операторы в конструкции WITH будут фактически вносить изменения в данные, непредсказуем. Им нельзя управлять, изменяя порядок следования этих операторов в тексте конструкции WITH.

Мы уже выяснили ранее, что немедленно получить результаты выполнения операторов, модифицирующих базу данных, можно с помощью предложения RETURNING. Давайте добавим предложение RETURNING в оба оператора, модифицирующие таблицу «Места» (seats), а в главном запросе воспользуемся результатами этих операторов:

```
BEGIN;
BEGIN
WITH deleted_seats AS
( DELETE FROM seats
  WHERE aircraft_code = 'SU9'
    AND left( seat_no, 1 ) IN ( '4', '5' )
  RETURNING *
),
inserted_seats AS
( INSERT INTO seats
  SELECT aircraft_code, seat_no || '+', fare_conditions
  FROM seats
  WHERE aircraft_code = 'SU9'
    AND left( seat_no, 1 ) = '3'
  RETURNING *
)
SELECT count( * ) AS "Было",
       ( SELECT count( * ) FROM deleted_seats ) AS "Удалено",
       ( SELECT count( * ) FROM inserted_seats ) AS "Добавлено",
       count( * ) -
         ( SELECT count( * ) FROM deleted_seats ) +
         ( SELECT count( * ) FROM inserted_seats ) AS "Стало"
FROM seats
WHERE aircraft_code = 'SU9';
```

Вот теперь мы сразу увидим результаты изменения таблицы «Места» (seats). Но нужно учитывать, что результат в столбце «Стало» получен путем вычислений, а не прямым подсчетом строк в таблице.

```
 Было | Удалено | Добавлено | Стало
------+---------+-----------+-------
   97 |      10 |         4 |    91
(1 строка)
```

```
ROLLBACK;
ROLLBACK
```

Возникает вопрос: можно ли увидеть в плане запроса какие-либо признаки того, что операторы в конструкции WITH и главный запрос не видят непосредственно результаты работы друг друга? Давайте получим план предыдущего запроса, но воспользуемся рядом параметров команды EXPLAIN для его упрощения:

```
BEGIN;
BEGIN
EXPLAIN ( analyze, costs off, timing off, summary off )
...
                                QUERY PLAN
---------------------------------------------------------------------------
 Aggregate (actual rows=1 loops=1)
   CTE deleted_seats
     -> Delete on seats seats_1 (actual rows=10 loops=1)
           -> Bitmap Index Scan on seats_pkey (actual rows=97 loops=1)
               Index Cond: (aircraft_code = 'SU9'::bpchar)
   CTE inserted_seats
     -> Insert on seats seats_2 (actual rows=4 loops=1)
           -> Bitmap Index Scan on seats_pkey (actual rows=97 loops=1)
               Index Cond: (aircraft_code = 'SU9'::bpchar)
   InitPlan 3
     -> Aggregate (actual rows=1 loops=1)
          -> CTE Scan on deleted_seats (actual rows=10 loops=1)
   InitPlan 4
     -> Aggregate (actual rows=1 loops=1)
          -> CTE Scan on inserted_seats (actual rows=4 loops=1)
   InitPlan 5
     -> Aggregate (actual rows=1 loops=1)
          -> CTE Scan on deleted_seats deleted_seats_1 (actual rows=10 loops=1)
   InitPlan 6
     -> Aggregate (actual rows=1 loops=1)
          -> CTE Scan on inserted_seats inserted_seats_1 (actual rows=4 loops=1)
   -> Index Only Scan using seats_pkey on seats (actual rows=97 loops=1)
        Index Cond: (aircraft_code = 'SU9'::bpchar)
 Trigger for constraint seats_aircraft_code_fkey on seats: calls=4
 ...
ROLLBACK;
ROLLBACK
```

Обратите внимание, что число строк, отобранных обоими операторами в конструкции WITH и главным запросом, одинаковое — 97. Получается, что все они работали с исходной таблицей, в которой содержится 97 строк, удовлетворяющих условию отбора.

Попутно заметим, что строка плана «Trigger for constraint ...» говорит о том, что при вставке строк проверялось их соответствие ограничению внешнего ключа.

Таким образом, если данные изменяются в общих табличных выражениях, то доступ к результатам этих изменений в процессе выполнения команды можно получить только через предложение RETURNING. Это нужно учитывать, особенно если в одной команде присутствуют несколько общих табличных выражений, модифицирующих одну и ту же таблицу.

2.5. Контрольные вопросы и задания

1 Материализация общего табличного выражения может влиять на скорость выполнения всего запроса

Проиллюстрируем влияние материализации общего табличного выражения (конструкция WITH) на скорость выполнения всего запроса.

Предположим, что с целью определения степени покрытия страны маршрутной сетью нам нужно подсчитать число возможных прямых маршрутов между аэропортами, приведенными в таблице «Аэропорты» (airports). Это можно сделать с помощью следующего запроса (конечно, город вылета и город прибытия должны быть различными):

```
WITH aps AS ( SELECT * FROM airports )
SELECT count( * )
FROM aps AS a1
  JOIN aps AS a2 ON a1.city <> a2.city;
 count
-------
 10704
(1 строка)
```

Прежде чем перейти к дальнейшим рассуждениям, напомним, что объект airports является представлением. Вот его код:

```
\sv airports
CREATE OR REPLACE VIEW bookings.airports AS
 SELECT airport_code,
    airport_name ->> lang() AS airport_name,
    city ->> lang() AS city,
    coordinates,
    timezone
   FROM airports_data ml
```

Столбцы airport_name и city имеют тип данных jsonb. При выборке данных будут выводиться не полные значения JSON-объектов, содержащихся в этих столбцах, а только наименования аэропорта и города на языке, который определяется функцией lang. Оператор ->> возвращает значение поля из JSON-объекта в виде символьной строки.

```
SELECT lang();
 lang
------
 ru
(1 строка)
```

Вот так выглядит план запроса:

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Aggregate  (cost=329.82..329.83 rows=1 width=8)
   (actual time=3.793..3.794 rows=1 loops=1)
   CTE aps
     -> Seq Scan on airports_data ml  (cost=0.00..56.56 rows=104 width=99)
         (actual time=0.038..0.755 rows=104 loops=1)
   -> Nested Loop  (cost=0.00..246.48 rows=10712 width=0)
       (actual time=0.048..3.226 rows=10704 loops=1)
       Join Filter: (a1.city <> a2.city)
       Rows Removed by Join Filter: 112
       -> CTE Scan on aps a1  (cost=0.00..2.08 rows=104 width=32)
           (actual time=0.041..0.049 rows=104 loops=1)
       -> CTE Scan on aps a2  (cost=0.00..2.08 rows=104 width=32)
           (actual time=0.000..0.017 rows=104 loops=104)
 Planning Time: 0.094 ms
 Execution Time: 3.833 ms
(15 строк)
```

Поскольку результат выполнения подзапроса, представленного в конструкции WITH, используется в главном запросе более одного раза, то по умолчанию этот подзапрос материализуется — в плане присутствуют узлы CTE Scan on aps. При выполнении подзапроса aps создается временная таблица с таким же именем. Для соединения двух наборов строк используется метод вложенного цикла — узел Nested Loop. (Заметим попутно, что здесь другие методы использоваться не могут, поскольку условием соединения является неравенство значений столбцов.) В строке с меткой Join Filter этого узла показано условие соединения строк. Важно, что в данном случае как столбец a1.city, так и столбец a2.city представляют собой скалярные текстовые значения, а не объекты типа JSON, поскольку при материализации подзапроса для каждой строки были вычислены выражения с оператором ->>, предусмотренные представлением.

Давайте проведем другой эксперимент — отменим материализацию подзапроса в конструкции WITH.

```
WITH aps AS NOT MATERIALIZED ( SELECT * FROM airports )
SELECT count( * )
FROM aps AS a1
  JOIN aps AS a2 ON a1.city <> a2.city;
```

План запроса изменился:

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=5659.44..5659.45 rows=1 width=8)
   (actual time=24.145..24.147 rows=1 loops=1)
   -> Nested Loop  (cost=0.00..5632.66 rows=10712 width=0)
       (actual time=0.048..23.550 rows=10704 loops=1)
       Join Filter: ((ml.city ->> lang()) <> (ml_1.city ->> lang()))
       Rows Removed by Join Filter: 112
       -> Seq Scan on airports_data ml  (cost=0.00..4.04 rows=104 width=49)
           (actual time=0.010..0.026 rows=104 loops=1)
       -> Materialize  (cost=0.00..4.56 rows=104 width=49)
           (actual time=0.000..0.006 rows=104 loops=104)
           -> Seq Scan on airports_data ml_1  (cost=0.00..4.04 rows=104 width=49)
               (actual time=0.003..0.017 rows=104 loops=1)
 Planning Time: 0.109 ms
 Execution Time: 24.179 ms
(14 строк)
```

Теперь условие соединения Join Filter обращается к полям JSON-объектов, имена ключей которых формируются для каждой строки вызовами функции lang. Это приводит к значительному замедлению выполнения запроса.

В приведенном плане присутствует узел Materialize. Такая материализация работает так же, как и материализация общих табличных выражений. В разделе документации 14.1 «Использование EXPLAIN» сказано, что узел Materialize сохраняет считанные данные в памяти, а затем, когда эти данные потребуются вновь, он выдает их из памяти, а не обращается повторно к источнику данных. Как показывает приведенный план, при материализации базовой таблицы airports_data в памяти сохраняются JSON-объекты полностью, а не конкретные значения их полей, как это происходило при материализации представления airports в предыдущем эксперименте. В результате вызовы функции lang и получение поля JSON-объекта по ключу выполняются позднее — при проверке условия соединения для *каждой* пары строк. Чтобы получить более полное представление о материализации, добавьте в команду EXPLAIN параметр VERBOSE и посмотрите, что выводится в строках с меткой Output.

Задание 1. В основе представления «Аэропорты» (airports) лежит таблица airports_data. Повторите описанные эксперименты, заменив в запросах представление на эту таблицу. Сравните время выполнения запроса в новых экспериментах с теми результатами, которые были получены ранее, и дайте необходимые пояснения.

Указание. Учтите, что теперь в процессе соединения строк будут сравниваться не скалярные значения одного поля, взятые из JSON-объектов, а сами JSON-объекты.

Задание 2. Проверьте справедливость следующей гипотезы: если в запросе используются два совершенно одинаковых подзапроса, то автоматически будет сформировано общее табличное выражение, и оно будет материализовано. Для эксперимента можно воспользоваться запросом, приведенным в начале упражнения, заменив в нем общее табличное выражение двумя подзапросами:

```
SELECT count( * )
FROM ( SELECT * FROM airports ) AS a1
  JOIN ( SELECT * FROM airports ) AS a2 ON a1.city <> a2.city;
```

Указание. Учтите пояснения насчет узла Materialize, приведенные выше в этом упражнении.

2 А если главный запрос использует не все строки, порождаемые конструкцией WITH?

Давайте рассмотрим запрос, который не имеет практической пользы, но позволяет проиллюстрировать один интересный и важный эффект.

```
EXPLAIN ANALYZE
WITH g AS
( SELECT * FROM generate_series( 1, 100 ) AS gs( num )
)
SELECT g1.num, g2.num
FROM g AS g1
  JOIN g AS g2 ON g1.num <> g2.num
/* ORDER BY 1, 2 */
LIMIT 10;
```

Изучите план этого запроса. Обратите внимание на предполагаемое и фактическое число выбираемых строк во всех узлах плана, сравните их со значением, заданным в предложении LIMIT, и с числом строк, порождаемых функцией generate_series.

```
                                    QUERY PLAN
-------------------------------------------------------------------------------------------
 Limit  (cost=1.00..1.23 rows=10 width=8) (actual time=0.015..0.019 rows=10 loops=1)
   CTE g
     -> Function Scan on generate_series gs  (cost=0.00..1.00 rows=100 width=4)
          (actual time=0.009..0.010 rows=11 loops=1)
   -> Nested Loop  (cost=0.00..228.00 rows=9900 width=8)
        (actual time=0.014..0.017 rows=10 loops=1)
        Join Filter: (g1.num <> g2.num)
        Rows Removed by Join Filter: 1
        -> CTE Scan on g g1  (cost=0.00..2.00 rows=100 width=4)
             (actual time=0.011..0.011 rows=1 loops=1)
        -> CTE Scan on g g2  (cost=0.00..2.00 rows=100 width=4)
             (actual time=0.000..0.002 rows=11 loops=1)
 Planning Time: 0.069 ms
 Execution Time: 0.034 ms
(14 строк)
```

Задание 1. Раскомментируйте в приведенном запросе предложение ORDER BY, выполните запрос и посмотрите, что стало с этими же показателями плана. Попробуйте найти в разделе документации 7.8 «Запросы WITH (Общие табличные выражения)» объяснение обнаруженному эффекту.

Указание. Обратите внимание на подразделы документации 7.8.2.2 «Выявление циклов» и 7.8.3 «Материализация общих табличных выражений».

Задание 2. Предложите более полезный запрос в базе данных «Авиаперевозки», также иллюстрирующий продемонстрированный эффект.

Указание. Одним из вариантов может быть запрос для формирования всех возможных маршрутов между городами, скажем, одного часового пояса.

3 Изменчивая функция random: неожиданные эффекты

В подразделе документации 7.8.3 «Материализация общих табличных выражений» сказано, что если подзапрос, помещенный в конструкцию WITH, является нерекурсивным и свободным от побочных эффектов (то есть если это SELECT, не вызывающий изменчивых функций), он может быть встроен в главный запрос. Общее рассмотрение вопроса о том, что такое изменчивость функций, мы проведем в главе 5 «Подпрограммы» (с. 263), а сейчас скажем только, что изменчивая функция в рамках одного SQL-запроса может возвращать *различные результаты*, если будет вызвана несколько раз с *одинаковыми аргументами* (или без аргументов, если она их не принимает). Конечно, если аргументы будут различными, то и возвращаемое значение вовсе не обязано быть одним и тем же. Значение, полученное при выполнении предыдущего вызова изменчивой функции для *другой строки*, при обработке текущей строки использоваться не будет.

В качестве примера изменчивой функции приведем функцию random и посмотрим, как будет решаться вопрос материализации в случае ее использования в конструкции WITH.

Поставим задачу: сформировать два набора случайных значений и проверить, будут ли эти наборы одинаковыми при материализации и без нее.

```
WITH r AS
( SELECT g, random() AS rand
  FROM generate_series( 1, 5 ) AS g
)
SELECT r1.g AS g1, r1.rand AS rand1, r2.g AS g2, r2.rand AS rand2
FROM r AS r1
  JOIN r AS r2 ON r1.g = r2.g;
```

```
 g1 |        rand1        | g2 |        rand2
----+---------------------+----+---------------------
  1 | 0.22815992065731816 |  1 | 0.22815992065731816
  2 | 0.11832240506576386 |  2 | 0.11832240506576386
  3 |  0.4837381024562195 |  3 |  0.4837381024562195
  4 |  0.7350704222861746 |  4 |  0.7350704222861746
  5 |  0.3557016765076011 |  5 |  0.3557016765076011
(5 строк)
```

В плане запроса не будет ничего неожиданного: подзапрос материализуется, как это и должно быть по умолчанию, если он используется в главном запросе более одного раза. Материализация подзапроса означает, что результаты его выполнения сохраняются во временной таблице, и повторная выборка производится уже из нее, поэтому наборы случайных значений будут одинаковыми:

```
EXPLAIN ( costs off )
...
                QUERY PLAN
-----------------------------------------
 Hash Join
   Hash Cond: (r1.g = r2.g)
   CTE r
     -> Function Scan on generate_series g
   -> CTE Scan on r r1
   -> Hash
       -> CTE Scan on r r2
(7 строк)
```

Давайте потребуем, чтобы СУБД не выполняла материализацию подзапроса.

```
WITH r AS NOT MATERIALIZED
...
```

Увы, это не срабатывает: мешает изменчивая функция `random`. План запроса не изменится, и мы опять получим два одинаковых набора случайных значений:

```
 g1 |        rand1        | g2 |        rand2
----+---------------------+----+---------------------
  1 |  0.4074326907403574 |  1 |  0.4074326907403574
  2 | 0.04418083458685218 |  2 | 0.04418083458685218
  3 |   0.831435286636717 |  3 |   0.831435286636717
  4 |  0.6064335131813559 |  4 |  0.6064335131813559
  5 | 0.08002064876445303 |  5 | 0.08002064876445303
(5 строк)
```

Задание 1. Модифицируйте запрос таким образом, чтобы наборы случайных значений оказались различными.

Указание. Перенесите подзапрос из конструкции WITH в главный запрос.

Задание 2. Предложите свой — более реалистичный — пример использования функции random в общем табличном выражении и проведите аналогичные эксперименты.

4 Не является ли промежуточная таблица лишним звеном?

В разделе 2.2 «Рекурсивные общие табличные выражения» (с. 28) при рассмотрении алгоритма вычисления рекурсивного общего табличного выражения было сказано, что на каждой итерации содержимое рабочей таблицы заменяется содержимым промежуточной таблицы, а затем промежуточная таблица очищается.

Как вы думаете, нельзя ли было в этом процессе обойтись вообще без промежуточной таблицы? Ведь она очищается на каждой итерации.

5 Имя таблицы одно и то же, а совпадают ли наборы строк?

В разделе 2.2 «Рекурсивные общие табличные выражения» (с. 28) был приведен запрос:

```
WITH RECURSIVE included_parts( part, sub_part, quantity ) AS
( SELECT part, sub_part, quantity
  FROM parts
  WHERE part = 'самолет'
  UNION ALL
  SELECT p.part, p.sub_part, p.quantity * ip.quantity
  FROM included_parts ip,
       parts p
  WHERE p.part = ip.sub_part
)
SELECT sub_part, sum( quantity ) AS total_quantity
FROM included_parts
GROUP BY sub_part
ORDER BY sub_part;
```

В этом запросе имя временной таблицы included_parts дважды фигурирует в предложениях FROM: один раз — внутри общего табличного выражения, а второй — в главном запросе. Как вы думаете, в общем случае эти два обращения к таблице included_parts имеют дело с одним и тем же набором строк или нет? Объясните свой ответ.

Указание. Вспомните о понятии рабочей таблицы.

6 Почему рекурсивный запрос может зациклиться и что с этим делать?

Правильно написанный рекурсивный запрос должен рано или поздно завершиться. Но если не предусмотреть условие завершения, запрос может зациклиться.

Давайте смоделируем такую ситуацию, удалив из запроса, который был рассмотрен в разделе 2.2 «Рекурсивные общие табличные выражения» (с. 28), предложение WHERE общего табличного выражения.

```
WITH RECURSIVE included_parts( iteration, part, sub_part, quantity, path_to_sub_part ) AS
( SELECT 1, part, sub_part, quantity,
         part || ' -> ' || sub_part || ' (x' || quantity || ')'
    FROM parts WHERE part = 'самолет'
  UNION ALL
  SELECT iteration + 1, p.part, p.sub_part, p.quantity * ip.quantity,
         -- комбинированное значение
         ip.part || ' -> ' || ip.sub_part || ' (x' || ip.quantity || ')' ||
           CASE
             WHEN p.part IS NOT NULL
               THEN ' -> ' || p.sub_part || ' (x' || p.quantity || ')'
             ELSE ' -> составных частей нет'
           END
  FROM included_parts ip
    LEFT OUTER JOIN parts p ON p.part = ip.sub_part
--  WHERE ip.part IS NOT NULL  -- условие удалено
)
SELECT iteration, part, sub_part, quantity, path_to_sub_part
FROM included_parts
ORDER BY iteration;
```

Выполните запрос. Оказывается, теперь он не завершается. Прервите его с помощью клавиш **Ctrl**+**C**.

Задание 1. Для того чтобы увидеть, что происходит, вновь добавьте предложение WHERE, но вместо прежнего условия введите такое:

```
...
  WHERE iteration < 7
...
```

Теперь будет видно, что в выборку в том числе попадают строки, в которых все поля, кроме iteration, пустые. Почему такие строки появляются, причем уже на четвертой итерации, а не только на более поздних?

Задание 2. Вновь удалите предложение WHERE из общего табличного выражения, но добавьте в главный запрос предложение LIMIT с параметром, скажем, 30. Будет ли запрос зацикливаться теперь? Удалите предложение ORDER BY из главного запроса. Выполняется ли запрос успешно? Найдите объяснение этим результатам в подразделе документации 7.8.2.2 «Выявление циклов».

Задание 3. Посмотрите план запроса. Есть ли в нем какие-то признаки того, что запрос может зациклиться? Попробуйте в предложении LIMIT задать число строк значительно большее, чем та оценка этого числа, которую выдает планировщик. Выполните запрос, добавив в команду EXPLAIN параметр ANALYZE. Будет ли выбрано столько строк, сколько вы задали в предложении LIMIT?

7 Разница между UNION и UNION ALL в рекурсивном запросе

В подразделе документации 7.8.2 «Рекурсивные запросы» объясняется механизм их работы, и вводятся понятия рабочей таблицы (working table) и промежуточной таблицы (intermediate table). В этом же разделе приведен простой рекурсивный запрос:

```
WITH RECURSIVE t( n ) AS
( SELECT 1
  UNION ALL
  SELECT n + 1 FROM t
  WHERE n < 100
)
SELECT sum( n ) FROM t;
```

Задание 1. В документации говорится, что в этом запросе рабочая таблица содержит только одну строку на каждом шаге. Покажите, как можно проверить это утверждение.

Указание 1. Выполните запрос с командой EXPLAIN ANALYZE и изучите ее вывод. Выясните, какие строки плана отвечают за вычисление общего табличного выражения, в том числе его нерекурсивной и рекурсивной частей. Найдите узел WorkTable Scan. Обратите внимание на фактические значения rows и loops выполненного запроса. В частности, в узле Result вы увидите rows=1 и loops=1, в узле WorkTable Scan — rows=1 и loops=100.

Указание 2. Замените в условии предложения WHERE число 100 на число 2 и посмотрите, как это отразится на фактическом значении rows в узле WorkTable Scan в плане запроса. Учтите, что это целочисленное среднее, вычисленное по всем «прогонам» данного узла, число которых равно значению loops.

Задание 2. Измените запрос таким образом, чтобы в его нерекурсивной части формировалась не одна строка, а три, и уберите вычисление агрегатной функции sum из главного запроса.

```
WITH RECURSIVE t( n ) AS
( VALUES ( 1 ), ( 2 ), ( 3 )
  UNION ALL
  SELECT n + 1 FROM t
  WHERE n < 100
)
SELECT n FROM t;
```

```
  n
-----
   1     ⎫
   2     ⎬ 1
   3     ⎭
   2     ⎫
   3     ⎬ 2
   4     ⎭
 ...
  98     ⎫
  99     ⎬ 98
 100     ⎭
  99     ⎫
 100     ⎬ 99
 100
(297 строк)
```

Теперь выполните этот запрос с командой EXPLAIN ANALYZE.

```
                                QUERY PLAN
-------------------------------------------------------------------------
 CTE Scan on t  (cost=8.07..10.13 rows=103 width=4)
   (actual time=0.004..0.158 rows=297 loops=1)
   CTE t
     -> Recursive Union  (cost=0.00..8.07 rows=103 width=4)
         (actual time=0.002..0.113 rows=297 loops=1)
         -> Values Scan on "*VALUES*"  (cost=0.00..0.04 rows=3 width=4)
             (actual time=0.001..0.002 rows=3 loops=1)
         -> WorkTable Scan on t t_1  (cost=0.00..0.70 rows=10 width=4)
             (actual time=0.000..0.001 rows=3 loops=100)
             Filter: (n < 100)
             Rows Removed by Filter: 0
 Planning Time: 0.053 ms
 Execution Time: 0.181 ms
(13 строк)
```

Можно заметить, что нерекурсивная часть запроса выполняется один раз, возвращая три строки. Это ожидаемый и понятный результат. А рекурсивная часть запроса выполняется 100 раз, возвращая каждый раз по 3 (в среднем) строки, поскольку мы в запросе использовали конструкцию UNION ALL, которая не отбрасывает строки-дубликаты. Обратите внимание, что на предпоследней итерации возвращается всего две строки (99 и 100), а на последней итерации — одна строка (100).

Уберите ключевое слово ALL, оставив только UNION, и снова выполните запрос с командой EXPLAIN ANALYZE.

```
                                QUERY PLAN
-------------------------------------------------------------------------
 ...
         -> WorkTable Scan on t t_1  (cost=0.00..0.70 rows=10 width=4)
             (actual time=0.000..0.000 rows=1 loops=98)
 ...
```

Почему теперь значение rows стало равным 1 вместо 3, а значение loops теперь равно 98, а не 100? Опишите механизм формирования строк в рекурсивной части запроса детально, по шагам, обратив особое внимание на самые последние итерации.

8 Порядком обхода иерархии управлять нельзя, а порядком вывода — можно

Стратегией *обхода* иерархической структуры в процессе выполнения запроса мы управлять не можем: это определяется особенностями реализации СУБД. Однако *выводить* полученную иерархическую структуру можно в соответствии с определенным порядком. Традиционно используют порядки «сначала в ширину» или «сначала в глубину», хотя можно выбрать и другой порядок, наилучшим образом подходящий в конкретной ситуации.

Для реализации этих порядков вывода в запрос добавляют предложение `ORDER BY`, в котором используют столбец, отражающий уровень иерархии (в наших запросах это были столбцы `level` или `depth`), или столбец, содержащий путь от исходной вершины к целевой (у нас это был столбец `path`).

PostgreSQL предлагает встроенный синтаксис для реализации двух традиционных порядков вывода иерархий. При этом в общем табличном выражении не нужно задавать специальные столбцы и изменять их значения, как мы это делали в тексте главы. Теперь достаточно воспользоваться предложениями `SEARCH DEPTH FIRST` и `SEARCH BREADTH FIRST`, представленными в подразделе документации 7.8.2.1 «Порядок поиска».

Задание 1. Модифицируйте, например, запрос, выводящий руководителей конкретного работника на всех уровнях иерархии (см. раздел 2.2 «Рекурсивные общие табличные выражения», с. 28). Удалите из него столбец `level` и добавьте конструкцию `SEARCH BREADTH FIRST BY position_id SET ordercol`. Включите в список `SELECT` главного запроса дополнительный столбец `ordercol`. В предложении `ORDER BY` также укажите этот столбец.

Посмотрите, какие значения он будет содержать. Это будут не скалярные значения.

Задание 2. Модифицируйте, например, запрос, формирующий пути в иерархии к вершинам-листьям (см. раздел 2.3 «Массивы в общих табличных выражениях», с. 49). Удалите из него столбцы `depth` и `path` и добавьте конструкцию `SEARCH DEPTH FIRST BY vertex_to SET ordercol`. Включите в список `SELECT` главного запроса дополнительный столбец `ordercol`. В предложении `ORDER BY` также укажите этот столбец.

Посмотрите, какие пути будут сформированы. Затем замените в этой конструкции столбец vertex_to на vertex_from и повторите запрос. В сформированных путях отсутствует либо начальная вершина, либо конечная. Как можно исправить этот недостаток?

9 Для поиска циклов в графе есть встроенный синтаксис

В графах могут образовываться циклы. В тексте главы для их выявления мы использовали массивы. Однако PostgreSQL предлагает встроенный синтаксис для решения этой задачи: в запрос нужно добавить предложение CYCLE, представленное в подразделе документации 7.8.2.2 «Выявление циклов».

Задание. Модифицируйте приведенный в тексте главы запрос, выявляющий циклы в иерархии (см. раздел 2.3 «Массивы в общих табличных выражениях», с. 49). Удалите из него столбцы path и cycle и добавьте конструкцию CYCLE vertex_to SET is_cycle USING path. Посмотрите, какие пути будут сформированы.

Затем замените в этой конструкции столбец vertex_to на vertex_from и повторите запрос. Чем отличаются результаты исходного и двух модифицированных запросов?

10 В распавшейся на части иерархии тоже могут быть дефекты

В разделе 2.3 «Массивы в общих табличных выражениях» (с. 49) были рассмотрены различные отклонения от правильной структуры иерархии. Мы исходили из того, что даже с дефектами иерархия представлена связным графом, имеющим одно начало, из которого можно найти пути ко всем другим вершинам. Однако возможна ситуация, когда в силу каких-то причин граф, описывающий реальную иерархию, распадается на отдельные подграфы, не связанные друг с другом. В таком случае необходимо откорректировать запросы для выявления дефектов иерархии, рассмотренные в тексте главы.

Для выполнения задания необходимо подготовить таблицу «Иерархия» (hier). Сначала удалим из нее все строки, а затем вставим исходное множество строк, за исключением, например, строки (3, 6, 3.5). Тем самым мы удалим из графа ребро, соединяющее вершины 3 и 6 (рис. 2.13).

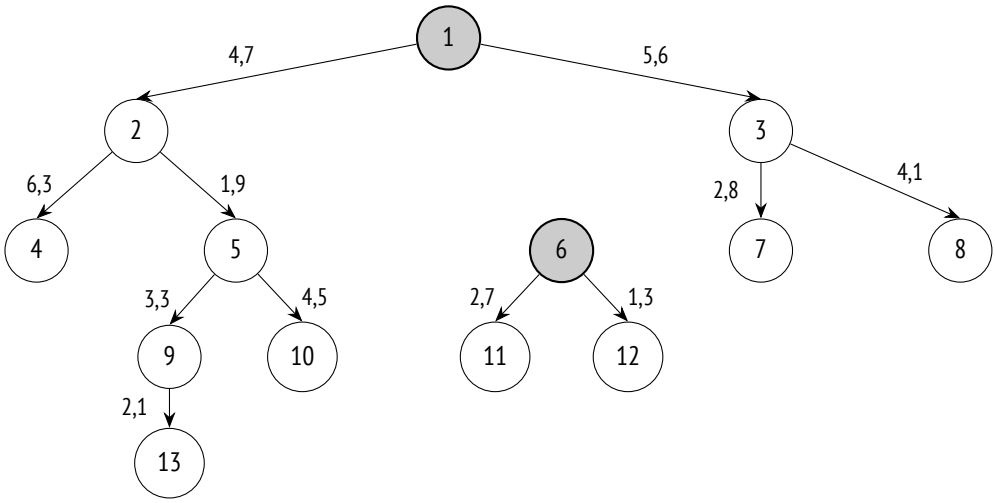


Рис. 2.13. *Отдельные подграфы после удаления ребра*

```
DELETE FROM hier;
DELETE 13
```

```
COPY hier FROM STDIN WITH ( format csv );
Вводите данные для копирования, разделяя строки переводом строки.
Закончите ввод строкой '\.' или сигналом EOF.
>> 1,2,4.7
1,3,5.6
2,4,6.3
2,5,1.9
3,7,2.8
3,8,4.1
5,9,3.3
5,10,4.5
6,11,2.7
6,12,1.3
9,13,2.1
\.
COPY 11
```

Теперь выполним запрос, приведенный в тексте главы, который формирует все пути, ведущие из начала иерархии ко всем другим вершинам.

```
WITH RECURSIVE search_hier( vertex_from, vertex_to, data, depth, path ) AS
( SELECT h.vertex_from, h.vertex_to, h.data, 1,
         ARRAY[ h.vertex_from, h.vertex_to ]
  FROM hier h
  WHERE h.vertex_from = 1
  UNION ALL
  SELECT h.vertex_from, h.vertex_to, h.data, sh.depth + 1,
         path || h.vertex_to
  FROM search_hier sh,
       hier h
  WHERE h.vertex_from = sh.vertex_to
)
SELECT * FROM search_hier
ORDER BY depth, vertex_from, vertex_to;
```

Как видно из полученного ответа, те пути, в которых задействована вершина 6, не были сформированы:

```
 vertex_from | vertex_to | data | depth |     path
-------------+-----------+------+-------+--------------
           1 |         2 |  4.7 |     1 | {1,2}
           1 |         3 |  5.6 |     1 | {1,3}
           2 |         4 |  6.3 |     2 | {1,2,4}
           2 |         5 |  1.9 |     2 | {1,2,5}
           3 |         7 |  2.8 |     2 | {1,3,7}
           3 |         8 |  4.1 |     2 | {1,3,8}
           5 |         9 |  3.3 |     3 | {1,2,5,9}
           5 |        10 |  4.5 |     3 | {1,2,5,10}
           9 |        13 |  2.1 |     4 | {1,2,5,9,13}
(9 строк)
```

Задание 1. Модифицируйте запрос таким образом, чтобы он формировал все пути для всех фрагментов иерархии. Обратите внимание, что отсутствие ребра (3, 6, 3.5) приведет к тому, что пути будут формироваться только в рамках каждого фрагмента иерархии.

Указание. Для отыскания вершины, являющейся начальной точкой иерархии, в тексте главы мы использовали такой запрос:

```
SELECT DISTINCT( vertex_from ) FROM hier h1
WHERE NOT EXISTS ( SELECT 1 FROM hier h2
                   WHERE h2.vertex_to = h1.vertex_from );
```

В случае, когда иерархия распадается на отдельные фрагменты, этот запрос выявит все вершины, являющиеся началами ее фрагментов.

Задание 2. Выполните и все другие запросы, приведенные в тексте главы, предназначенные для выявления дефектов иерархии. Модифицируйте их аналогичным образом.

11 Можно ли ускорить поиск маршрута между двумя городами?

В разделе 2.3 «Массивы в общих табличных выражениях» (с. 49) был рассмотрен поиск маршрута между двумя городами, возможно не связанными прямым рейсом. Предложенный запрос успешно решал поставленную задачу, однако его выполнение можно значительно ускорить. Опишем идею одного из возможных способов.

Обратите внимание, что объект «Маршруты» (routes) является представлением. При этом рекурсивное общее табличное выражение неоднократно обращается к нему, отбирая уникальные строки с помощью предложения DISTINCT ON. Предположительно выполнение запроса можно ускорить, используя вместо представления таблицу, содержащую только уникальные пары городов отправления и прибытия. К сожалению, у нас нет такой таблицы, но, к счастью, у нас есть общие табличные выражения.

Задание. Реализуйте описанную идею. Сравните время выполнения обоих вариантов запроса.

Указание. Общее табличное выражение, формирующее уникальные пары городов отправления и прибытия, можно сделать вложенным.

```
WITH RECURSIVE search_route( city_from, city_to, transfers, route ) AS
( WITH unique_routes AS
  ( SELECT ...
    FROM routes
  )
...
```

Однако можно избежать вложенности, поместив подзапрос unique_routes после подзапроса search_route. Такие ссылки «вперед» допустимы, если рекурсивное общее табличное выражение идет первым.

12 Пути к вершинам-листьям можно найти, не создавая их список заранее

В разделе 2.3 «Массивы в общих табличных выражениях» (с. 49) был приведен запрос для формирования всех путей от начала иерархии только к вершинам-листьям. В нем использовалось общее табличное выражение leaves, определяющее список таких вершин, к которому затем обращался главный запрос. Однако можно предложить и другое решение, не требующее формирования списка листьев: из всех путей, сформированных в общем табличном выражении, в главном запросе нужно отбирать только те, для которых не существует более длинного пути, включающего в себя данный.

Конечно, решение о том, какой подход выбрать, принимается с учетом специфики конкретной ситуации.

```
WITH RECURSIVE search_hier( vertex_from, vertex_to, data, depth, path ) AS
( SELECT h.vertex_from, h.vertex_to, h.data, 1,
         ARRAY[ h.vertex_from, h.vertex_to ]
  FROM hier h
  WHERE h.vertex_from = 1
  UNION ALL
  SELECT h.vertex_from, h.vertex_to, h.data, sh.depth + 1,
         path || h.vertex_to
  FROM search_hier sh,
       hier h
  WHERE h.vertex_from = sh.vertex_to
)
SELECT path[ array_length( path, 1 ) ] AS leaf, path
FROM search_hier sh1
WHERE NOT EXISTS
      ( SELECT 1
        FROM search_hier sh2
        WHERE array_length( sh2.path, 1 ) > array_length( sh1.path, 1 )
          AND sh2.path @> sh1.path
      )
ORDER BY path;
```

Вторым параметром функции array_length является номер измерения массива, который для одномерных массивов равен единице. Оператор @> проверяет, что левый массив включает в себя все элементы правого массива.

```
 leaf |     path
------+--------------
    4 | {1,2,4}
   13 | {1,2,5,9,13}
   10 | {1,2,5,10}
    7 | {1,3,7}
    8 | {1,3,8}
(5 строк)
```

Вопрос. Поскольку нас сейчас интересуют только полные пути, а не этапы их формирования, нельзя ли упростить приведенный запрос: убрать из обоих предложений SELECT в общем табличном выражении столбец h.vertex_from? А столбец h.vertex_to? Проверьте ваши предположения на практике.

13 В массив можно «собрать» не только путь, но и стоимости ребер вдоль этого пути

В разделе 2.3 «Массивы в общих табличных выражениях» (с. 49) был приведен запрос, выводящий неуникальные пути от начала иерархии к ее вершинам, а также общие стоимости этих путей. Модифицируйте запрос таким образом, чтобы он выводил не только суммарные стоимости выбранных путей, но также и стоимости ребер, образующих эти пути, в виде массивов. Это может выглядеть примерно так:

```
 vertex_to |     path     |    path_values    | total_value
-----------+--------------+-------------------+-------------
         9 | {1,2,4,9}    | {4.7,6.3,5.7}     |        16.7
         9 | {1,2,5,9}    | {4.7,1.9,3.3}     |         9.9
        13 | {1,2,4,9,13} | {4.7,6.3,5.7,2.1} |        18.8
        13 | {1,2,5,9,13} | {4.7,1.9,3.3,2.1} |        12.0
(4 строки)
```

Указание. Чтобы в таблице «Иерархия» (hier) образовались неуникальные пути, не забудьте предварительно добавить в нее еще одну строку (как это было сделано в тексте главы):

```
INSERT INTO hier VALUES ( 4, 9, 5.7 );
```

В противном случае запрос не вернет ни одной строки.

14 Повторим все эксперименты на «большой» двоичной иерархии

Для проведения экспериментов может потребоваться более масштабная иерархия, чем та, что использовалась в тексте главы. Перед ее созданием нужно очистить таблицу «Иерархия» (hier). Число удаленных строк у вас может оказаться другим, в зависимости от проведенных экспериментов.

```
DELETE FROM hier;
DELETE 11
```

Создать новую иерархию можно с помощью, например, такого запроса:

```
WITH RECURSIVE make_hier( vertex_from, vertex_to ) AS
( VALUES ( 1, 2 ), ( 1, 3 )
  UNION ALL
  SELECT sub.vertex_from, sub.vertex_to
  FROM make_hier mh
    JOIN unnest( ARRAY[ mh.vertex_to, mh.vertex_to ],
                 ARRAY[ mh.vertex_to * 2, mh.vertex_to * 2 + 1 ]
               ) AS sub( vertex_from, vertex_to ) -- subordinate
      ON mh.vertex_to = sub.vertex_from
)
INSERT INTO hier
SELECT vertex_from, vertex_to, round( ( random() * 10 )::numeric( 3, 1 ), 1 )
FROM make_hier
LIMIT 40000;
INSERT 0 40000
```

Как работает этот запрос, будет понятнее после изучения главы 4 «Конструкция LATERAL команды SELECT» (с. 225). Пока скажем только, что для каждого ребра (то есть каждой строки), сформированного на предыдущей итерации и содержащегося в рабочей таблице make_hier, будет создано еще два ребра, исходящих из него. Вызов функции unnest выполняется для каждой строки рабочей таблицы, а значения ее параметров берутся из этой строки. В рассматриваемом запросе функция unnest разворачивает два массива в два столбца таблицы, число формируемых ею строк также будет равно двум. При этом каждая строка из рабочей таблицы обычным образом соединяется с двумя строками, сформированными за счет вызова функции unnest. Таким образом шаг за шагом формируется результирующее множество строк, при этом в список SELECT попадают только столбцы, сформированные функцией unnest.

Сформированный запросом граф будет бинарным деревом. Столбец data таблицы «Иерархия» (hier) будет заполнен случайными числами от 0,0 до 10,0. Количество вершин иерархии можно задавать в предложении LIMIT.

Задание 1. Проведите эксперименты, рассмотренные в тексте главы, на основе сформированной «большой» иерархии. Когда потребуется, создайте в этом графе неуникальные пути и циклы с помощью дополнительных SQL-команд.

Задание 2. Модифицируйте запрос таким образом, чтобы в формируемой иерархии каждая вершина имела не две подчиненные вершины, а три.

Указание. К этому заданию можно вернуться после изучения главы 4 «Конструкция LATERAL команды SELECT» (с. 225).

15 Как выбрать альтернативные пути, имеющиеся в иерархии?

В разделе 2.3 «Массивы в общих табличных выражениях» (с. 49) мы занимались выявлением различных дефектов иерархической структуры. В этом упражнении рассмотрим более подробно нахождение альтернативных путей между одними и теми же вершинами. Перед началом экспериментов нужно позаботиться о том, чтобы таблица «Иерархия» (hier) была в исходном состоянии:

```
DELETE FROM hier;
DELETE 40000
COPY hier FROM STDIN WITH ( format csv );
Вводите данные для копирования, разделяя строки переводом строки.
Закончите ввод строкой '\.' или сигналом EOF.
>> 1,2,4.7
1,3,5.6
2,4,6.3
2,5,1.9
3,6,3.5
3,7,2.8
3,8,4.1
5,9,3.3
5,10,4.5
6,11,2.7
6,12,1.3
9,13,2.1
\.
COPY 12
```

Давайте удалим ограничение первичного ключа. Он был создан по столбцам `vertex_from` и `vertex_to`, чтобы не допускать наличия ребер-дубликатов.

```
ALTER TABLE hier DROP CONSTRAINT hier_pkey;
ALTER TABLE
```

Теперь мы сможем добавить еще два ребра, дублирующих уже существующие ребра, однако веса им назначим другие. Одно ребро будет соединять вершины 5 и 9, а другое — 9 и 13 (рис. 2.14).

```
INSERT INTO hier VALUES ( 5, 9, 7.2 ), ( 9, 13, 8.1 );
INSERT 0 2
```

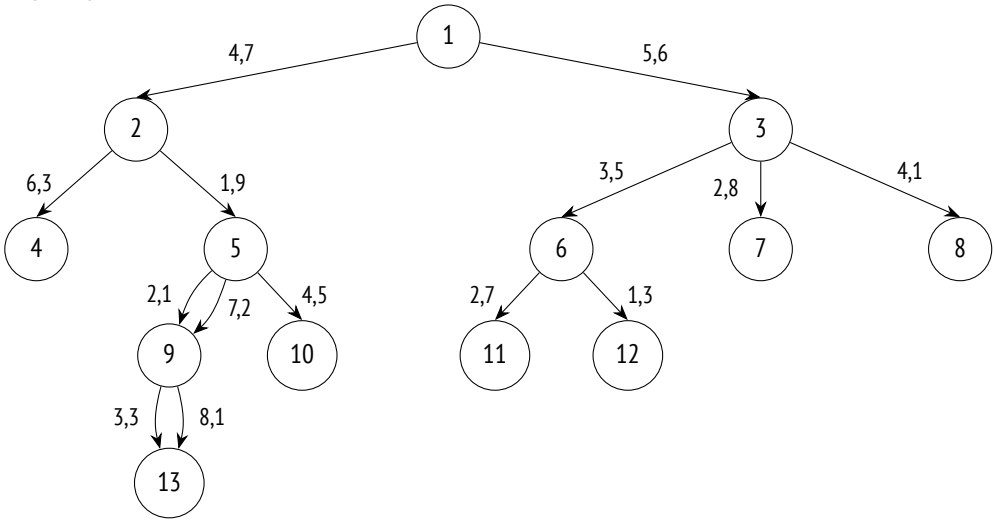


Рис. 2.14. *Иерархия с дубликатами ребер*

Используя один из запросов, приведенных в разделе 2.3, определим общие «стоимости» альтернативных путей, включающих вершины 5 и 9:

```
 vertex_to |     path     | total_value
-----------+--------------+-------------
         9 | {1,2,5,9}    |         9.9
         9 | {1,2,5,9}    |        13.8
        13 | {1,2,5,9,13} |        12.0
        13 | {1,2,5,9,13} |        18.0
        13 | {1,2,5,9,13} |        15.9
        13 | {1,2,5,9,13} |        21.9
(6 строк)
```

Таким образом, запрос успешно выполняется и при наличии ребер-дубликатов, позволяя сравнить альтернативные пути. Обратите внимание, что существует четыре таких пути к вершине 13, поскольку каждый из двух путей, ведущих к вершине 9, можно продолжить двумя способами.

Задание. Выполните и другие запросы, приведенные в разделе 2.3, на основе иерархии, дополненной ребрами-дубликатами. Проанализируйте полученные результаты.

16 С помощью запросов, разработанных для иерархий, можно исследовать и граф общего вида

В разделе 2.3 «Массивы в общих табличных выражениях» (с. 49) мы рассмотрели целый ряд запросов, позволяющих исследовать иерархическую структуру.

Задание. Попробуйте применить эти запросы к графу общего вида. При необходимости модифицируйте запросы. В качестве модельной системы можно рассмотреть, например, сеть автодорог региона. В ней могут существовать альтернативные пути между населенными пунктами, соседние населенные пункты могут соединяться двумя дорогами (ребра-дубликаты на графе). Возможно, эти избыточные пути в реальной дорожной сети можно было бы сделать запасными, выделять на их поддержание меньше ресурсов и т. п.

17 Главный запрос и все общие табличные выражения в WITH работают с одним и тем же снимком данных

В подразделе документации 7.8.4 «Изменение данных в WITH» сказано, что и главный запрос, и все общие табличные выражения в `WITH` работают с одним и тем же снимком данных.

Давайте проведем эксперимент, поместив все SQL-запросы, которые требуется выполнить для реализации процедуры бронирования, в один SQL-запрос с общими табличными выражениями.

В реальной работе код бронирования и номер билета должны генерироваться с помощью специальных процедур, гарантирующих их уникальность; мы же допустим упрощение, задав эти значения в виде констант. Но, чтобы наш эксперимент все же был приближен к реальности, в тех подзапросах, которые работают с подчиненными таблицами, не будем повторно задавать те же самые

константы, а будем получать значения кода бронирования и номера билета из временных таблиц. Так мы оправдаем наличие предложений RETURNING.

В первом подзапросе (bk) мы вставим строку в таблицу «Бронирования» (bookings), но значение поля total_amount в ней назначим равным нулю, поскольку на данном этапе оно неизвестно. После завершения процедуры оно должно стать равным сумме стоимостей всех перелетов, оформленных в рамках этой процедуры бронирования.

Во втором подзапросе (tk) оформим два билета. При этом берем код бронирования, обращаясь к временной таблице, сформированной с помощью предложения RETURNING в первом подзапросе.

В третьем подзапросе (tkf) сформируем перелеты. В нем мы тоже берем номер билета из подзапроса, а не вводим его как константу.

Главный запрос обновляет строку, вставленную в первом подзапросе, получив доступ к стоимостям всех оформленных перелетов.

```
BEGIN;
BEGIN
WITH bk AS
( INSERT INTO bookings ( book_ref, book_date, total_amount )
  VALUES ( 'ABC123', bookings.now(), 0 )
  RETURNING book_ref
),
tk AS
( INSERT INTO tickets ( ticket_no, book_ref, passenger_id, passenger_name )
  VALUES ( '9991234567890', ( SELECT book_ref FROM bk ), '1234 123456', 'IVAN PETROV' ),
         ( '9991234567891', ( SELECT book_ref FROM bk ), '4321 654321', 'PETR IVANOV' )
  RETURNING ticket_no, passenger_id
),
tkf AS
( INSERT INTO ticket_flights ( ticket_no, flight_id, fare_conditions, amount )
  VALUES ( ( SELECT ticket_no FROM tk WHERE passenger_id = '1234 123456' ),
           5572, 'Business', 12500 ),
         ( ( SELECT ticket_no FROM tk WHERE passenger_id = '4321 654321' ),
           13881, 'Economy', 8500 )
  RETURNING amount
)
UPDATE bookings
SET total_amount = ( SELECT sum( amount ) FROM tkf )
WHERE book_ref = ( SELECT book_ref FROM bk );
```

Задание 1. Сделайте обоснованное предположение: будет ли обновлена строка в таблице «Бронирования» (bookings)? Проверьте вашу гипотезу практически.

```
SELECT * FROM bookings WHERE book_ref = 'ABC123';
```

Какое число будет в столбце total_amount?

Поскольку эксперименты продолжатся, отмените транзакцию.

```
ROLLBACK;
ROLLBACK
```

Задание 2. Разделите приведенный запрос на отдельные запросы и выполните их в рамках транзакции на уровне изоляции, например, Repeatable Read. На этом уровне все запросы транзакции выполняются с тем снимком данных, который был сделан в момент начала выполнения ее первого запроса (см. подраздел документации 13.2.2 «Уровень изоляции Repeatable Read»). Будет ли теперь обновлена строка в таблице «Бронирования» (bookings)? Сопоставьте результаты двух проведенных экспериментов. Конечно, во втором эксперименте нам не удастся получать повторный доступ к коду бронирования и номеру билета с помощью подзапросов, но в данном случае для нас важнее сопоставить механизмы выполнения запроса с общими табличными выражениями и транзакции, имеющей аналогичный набор команд.

18 Процедуру архивирования таблиц можно ускорить

В разделе 2.4 «Модификация данных в общем табличном выражении» (с. 69) мы рассматривали запрос, который переносит в архивные таблицы часть строк из оперативных таблиц. Было отмечено, что время выполнения запроса можно значительно уменьшить.

Задание 1. Выясните, какое из общих табличных выражений вносит наибольший вклад в общее время выполнения запроса.

Указание 1. Воспользуйтесь командой EXPLAIN с параметром ANALYZE. Обратите внимание на строки «Trigger for constraint ...» в нижней части плана. Вспомните, что система должна гарантировать ссылочную целостность. Причем по умолчанию ограничение проверяется в самом конце выполнения запроса, после завершения фактического удаления строк из оперативных таблиц.

Указание 2. Можно воспользоваться тем решением, которое было реализовано в виде группы команд INSERT и DELETE. В среде утилиты psql включите таймер (\timing on) и выполните все эти команды (в рамках транзакции, конечно, чтобы можно было провести эксперимент неоднократно). Посмотрите, какая из команд будет выполняться дольше всех.

Задание 2. Предложите способ ускорения работы запроса. Возможно, удастся обойтись без его модификации?

Указание. Вспомните, что при удалении строк проверяется, нет ли в других таблицах, связанных с данной внешним ключом, строк, которые ссылаются на удаляемые строки. Каким образом можно ускорить эту проверку?

19 В тексте главы была первая часть процедуры архивирования таблиц, а это — ее завершение

В разделе 2.4 «Модификация данных в общем табличном выражении» (с. 69) мы реализовали первую часть процедуры архивирования ряда таблиц. Вторую часть — архивирование таблиц «Билеты» (tickets) и «Бронирования» (bookings) — рассмотрим в этом упражнении.

Из таблиц «Рейсы» (flights), «Перелеты» (ticket_flights) и «Посадочные талоны» (boarding_passes) мы переносили в архивные таблицы записи, относящиеся к рейсам, «возраст» которых не менее тридцати дней.

Из таблицы «Билеты» (tickets) будем переносить в архивную таблицу те строки, на которые не ссылается ни одна строка в таблице «Перелеты» (ticket_flights). Проще говоря, это «пустые» билеты, из которых уже перенесены в архив все когда-то оформленные в них перелеты.

Обратите внимание, что в предложении HAVING подзапроса empty_tickets параметром функции count является не *, а tf.*. Это нужно для того, чтобы учитывались только те строки, в которых столбцы из таблицы «Перелеты» (ticket_flights) имеют значения, отличные от NULL. В результате для «пустых» билетов будет получено значение 0. Заметим попутно, что в списке SELECT не требуется повторять вызовы функций, которые используются в предложении HAVING.

Также обратите внимание, что в предложении RETURNING подзапроса deleted_t фигурирует не *, а t.*; иначе были бы возвращены столбцы не только таблицы «Билеты» (tickets), но и вспомогательной таблицы empty_tickets, а они нам не нужны.

```
WITH empty_tickets AS
( SELECT t.ticket_no
  FROM tickets AS t LEFT OUTER JOIN ticket_flights AS tf
         ON tf.ticket_no = t.ticket_no
  GROUP BY t.ticket_no
  HAVING count( tf.* ) = 0
),
deleted_t AS
( DELETE FROM tickets AS t
  USING empty_tickets AS et
  WHERE t.ticket_no = et.ticket_no
  RETURNING t.*
)
INSERT INTO tickets_arch
SELECT * FROM deleted_t;
```

Задание 1. Проверьте приведенный запрос в действии. Реализуйте аналогичным способом перенос строк из таблицы «Бронирования» (bookings) в архивную таблицу. Выполняйте операции в рамках транзакции с ее последующей отменой, чтобы можно было повторить действия.

Задание 2. Подумайте об ускорении выполнения написанного вами запроса. Можно воспользоваться указаниями к выполнению предыдущего упражнения.

Глава 3
Аналитические возможности PostgreSQL

Из больших объемов данных можно извлекать ценную информацию, помогающую принимать обоснованные решения. Для этого используются специальные математические методы и соответствующее программное обеспечение. Однако зачастую можно получить полезные результаты и с помощью тех аналитических средств, которыми располагает PostgreSQL. Речь идет об агрегатных, статистических и оконных функциях, а также о группировках с помощью конструкций GROUPING SETS, CUBE и ROLLUP.

3.1. Агрегатные функции

Агрегатные функции уже были представлены в первой части учебника[1], где они широко использовались при рассмотрении различных аспектов языка SQL. Поэтому сейчас мы ограничимся кратким обзором этих функций, чтобы напомнить основные особенности их применения, а затем рассмотрим дополнительные вопросы: агрегирование в параллельном режиме и агрегирование данных типа JSON. Так называемые гипотезирующие агрегатные функции будут представлены в отдельном разделе 3.5. Эти вопросы не имеют прямой связи друг с другом, но все они представляют определенный интерес.

3.1.1. Краткий обзор

При агрегировании группа строк заменяется одной строкой, представляющей всю группу. Например, функция count подсчитывает количество строк в группе. Она всегда возвращает число, даже если в группу не попадает ни одной

[1] Моргунов, Е. П. PostgreSQL. Основы языка SQL. — СПб. : БХВ-Петербург, 2018. — 336 с. — ISBN 978-5-9775-4022-3.

строки — в этом случае значение будет равно нулю. А вот остальные функции в таких случаях возвращают NULL. Функции min и max отыскивают наименьшее и наибольшее значения указанного выражения в группе строк, поэтому результирующая строка будет содержать *неизмененное* значение, взятое из *одной* строки группы, которая в данном случае и будет являться представителем группы. Функция sum подсчитывает сумму, а avg вычисляет среднее, поэтому обе функции возвращают результат *обработки* исходных значений, полученных из *всех* строк группы.

В одном запросе можно использовать несколько агрегатных функций, что мы сейчас и попробуем. При этом покажем не только уже известные функции, рассмотренные в первой части учебника, но и новые функции bool_and и bool_or. В качестве полигона воспользуемся таблицей «Самолеты» (aircrafts). Получим значения разных агрегатных функций для самолетов компаний Airbus и Boeing, причем все модели каждой компании представим одной строкой:

```
SELECT
  string_agg( model, E'\n ' ORDER BY model ) AS models,
  array_agg( range ORDER BY model ) AS ranges,
  count( * ),
  min( range ),
  max( range ),
  round( avg( range ), 2 ) AS average,
  bool_and( range > 5000 ) AS b_and,
  bool_or( range < 5000 ) AS b_or
FROM aircrafts
WHERE left( model, strpos( model, ' ' ) - 1 ) IN ( 'Аэробус', 'Боинг' )
GROUP BY left( model, strpos( model, ' ' ) - 1 );
```

```
      models       |      ranges       | count | min  |  max  | average | b_and | b_or
-------------------+-------------------+-------+------+-------+---------+-------+------
 Аэробус A319-100 +| {6700,5700,5600}  |     3 | 5600 |  6700 | 6000.00 | t     | f
  Аэробус A320-200+|                   |       |      |       |         |       |
  Аэробус A321-200 |                   |       |      |       |         |       |
 Боинг 737-300    +| {4200,7900,11100} |     3 | 4200 | 11100 | 7733.33 | f     | t
  Боинг 767-300   +|                   |       |      |       |         |       |
  Боинг 777-300    |                   |       |      |       |         |       |
(2 строки)
```

Значения столбца models формируются конкатенацией значений поля model, взятых из всех строк каждой группы, содержащей описания моделей одной из компаний. Мы вставляем дополнительные символы-разделители: скобки

и символы перехода на новую строку \n (чтобы эти спецсимволы не интерпретировались буквально, перед строковым литералом добавлен символ E).

Похожим образом значения поля range, взятые из всех строк каждой группы, объединяются в массивы, представляющие значения столбца ranges.

Обратите внимание на предложение ORDER BY, присутствующее непосредственно в вызовах функций string_agg и array_agg. Эти и ряд других функций могут давать различные результаты в зависимости от порядка сортировки агрегируемых строк. Если этот порядок важен, его можно задать таким способом. Более подробно эта особенность описана в подразделе документации 4.2.7 «Агрегатные выражения».

Функция bool_and выполняет логическое умножение: она возвращает значение true, если для *всех* строк группы ее параметр также имеет значение true. Таким образом, в нашем запросе эта функция возвратит истинное значение, если дальность полета всех моделей конкретной компании превышает 5000 км.

Функция bool_or выполняет логическое сложение: она возвращает значение true, если в группе найдется *хотя бы одна* строка, для которой параметр функции также имеет значение true. Таким образом, в нашем запросе эта функция возвратит истинное значение, если дальность полета хотя бы одной модели конкретной компании меньше 5000 км.

Функции bool_and и bool_or являются расширением PostgreSQL.

Тип значения, возвращаемого агрегатной функцией, зачастую отличается от типа ее входного параметра. Например, функция avg, получив параметр типа integer, возвратит значение типа numeric. Функция sum при сложении чисел типа bigint также возвратит значение типа numeric, а не bigint. Функция count всегда возвращает значение типа bigint, даже если число строк в выборке заведомо не требует восьмибайтового целого числа для их подсчета. Эти сведения можно уточнить в таблице 9.60 «Агрегатные функции общего назначения», приведенной в разделе документации 9.21 «Агрегатные функции».

Из этой же таблицы можно узнать, что функции max и min могут работать не только с числами, но и со строками. В ряде случаев эта способность позволяет значительно упростить запрос. Пусть, например, требуется найти в таблице «Билеты» (tickets) имя и фамилию первого и последнего по алфавиту пассажиров. Индекс по столбцу passenger_name не создан.

Первое решение может выглядеть так:

```
SELECT
  ( SELECT passenger_name FROM tickets ORDER BY passenger_name LIMIT 1 ) AS min,
  ( SELECT passenger_name FROM tickets ORDER BY passenger_name DESC LIMIT 1 ) AS max;
       min        |      max
------------------+----------------
 ADELINA AFANASEVA | ZULFIYA ZOTOVA
(1 строка)
```

Но проще получить тот же результат с помощью функций min и max:

```
SELECT min( passenger_name ), max( passenger_name )
FROM tickets;
       min        |      max
------------------+----------------
 ADELINA AFANASEVA | ZULFIYA ZOTOVA
(1 строка)
```

Этим возможности не исчерпываются: функции могут работать и с массивами. Покажем это на примере представления «Маршруты» (routes). Столбец days_of_week представляет собой массив целых чисел — номеров дней недели, по которым выполняются рейсы:

```
SELECT min( days_of_week ), max( days_of_week )
FROM routes;
 min | max
-----+-----
 {1} | {7}
(1 строка)
```

Для лучшего понимания этого результата выведем все существующие графики полетов в течение недели:

```
SELECT DISTINCT days_of_week
FROM routes
ORDER BY days_of_week;
  days_of_week
-----------------
 {1}
 {1,2,3,4,5,6,7}
 {1,3,6}
 ...
```

```
 {3}
 {3,5,7}
 {3,6}
 {3,7}
 {4}
 {4,7}
 {6}
 {7}
(20 строк)
```

Таким образом, массивы, как и строки, сортируются в лексикографическом порядке, то есть сначала сравниваются первые элементы массивов, при их совпадении сравниваются вторые элементы и т. д.

Агрегатные функции тесно связаны с предложением GROUP BY. Конечно, можно написать запрос с этим предложением, но без агрегатной функции. Это будет фактически то же самое, что SELECT DISTINCT:

```
SELECT status
FROM flights
GROUP BY status;
  status
-----------
 Departed
 Arrived
 On Time
 Cancelled
 Delayed
 Scheduled
(6 строк)
```

Добавим агрегатную функцию count:

```
SELECT status, count( * )
FROM flights
GROUP BY status;
  status   | count
-----------+-------
 Departed  |    58
 Arrived   | 16707
 On Time   |   518
 Cancelled |   414
 Delayed   |    41
 Scheduled | 15383
(6 строк)
```

Воспользуемся командой `EXPLAIN` с параметром `COSTS OFF`. Она покажет, что *структуры* планов обоих запросов совершенно одинаковы:

```
        QUERY PLAN
--------------------------
 HashAggregate
   Group Key: status
   -> Seq Scan on flights
(3 строки)
```

Но, конечно, оценки затрат ресурсов будут немного выше для запроса с функцией `count`.

3.1.2. Агрегирование в параллельном режиме

Завершив краткий обзор возможностей агрегатных функций, перейдем к тем вопросам, которые были объявлены в начале раздела. Начнем с рассмотрения агрегирования в параллельном режиме.

Современные процессоры имеют несколько ядер, а серверы могут быть и многопроцессорными. Используя эти вычислительные мощности, PostgreSQL может ускорять запросы, выполняя их в параллельном режиме. В этом режиме может выполняться сканирование и соединение таблиц, объединение строк из различных источников в единый набор результатов (например, с помощью `UNION ALL`), а также агрегирование строк.

Не всегда параллельный план может дать выигрыш по времени выполнения запроса. В таких случаях планировщик не будет распараллеливать запрос, даже если в принципе это возможно. Как сказано в главе документации 15 «Параллельный запрос», наибольшего ускорения можно ожидать от запросов, обрабатывающих большой объем данных, но возвращающих пользователю всего несколько строк.

При выполнении запроса в параллельном режиме процесс, изначально обрабатывающий его, считается *ведущим*. Он может при необходимости запрашивать создание *фоновых рабочих процессов*.

В плане запроса, выполняемого в параллельном режиме, будет присутствовать узел Gather (или Gather Merge), который отвечает за сбор результатов работы

фоновых рабочих процессов. Если строки, порождаемые ими, отсортированы, то используется узел Gather Merge, который выполняет слияние этих строк с сохранением порядка сортировки. Узел Gather выдает строки рабочих процессов в произвольном порядке.

В параллельном режиме выполняется часть плана, которая находится ниже узла Gather (или Gather Merge). Как правило, рабочие процессы выполняют *один и тот же* фрагмент плана, но обрабатывают *разные* данные. Сам узел Gather и все узлы, располагающиеся выше него (если они есть), выполняются только ведущим процессом.

Для того чтобы планировщик решил распараллелить план запроса, необходимо выполнение целого ряда различных условий. В некоторых случаях распараллеливание может не применяться, а бывают такие ситуации, которые делают параллельную обработку в принципе невозможной. Подробно об этом говорится в разделе документации 15.2 «Когда может применяться распараллеливание запросов».

Сделав общее описание технологии параллельной обработки запросов, обратимся к конкретному примеру и детали поясним уже на нем.

Предположим, что руководство компании интересует общее число перелетов каждого класса обслуживания за весь отчетный период:

```
SELECT fare_conditions, count( flight_id )
FROM ticket_flights
GROUP BY fare_conditions;
 fare_conditions | count
-----------------+--------
 Business        | 107642
 Comfort         |  17291
 Economy         | 920793
(3 строки)
```

В самом этом запросе нет ничего примечательного. Но давайте посмотрим план его выполнения с помощью команды EXPLAIN с параметром ANALYZE:

```
EXPLAIN ( analyze, costs off, timing off )
SELECT fare_conditions, count( flight_id )
FROM ticket_flights
GROUP BY fare_conditions;
```

```
                                    QUERY PLAN
-------------------------------------------------------------------------------------------
 Finalize GroupAggregate (actual rows=3 loops=1)
   Group Key: fare_conditions
   -> Gather Merge (actual rows=9 loops=1)
       Workers Planned: 2
       Workers Launched: 2
       -> Sort (actual rows=3 loops=3)
           Sort Key: fare_conditions
           Sort Method: quicksort  Memory: 25kB
           Worker 0:  Sort Method: quicksort  Memory: 25kB
           Worker 1:  Sort Method: quicksort  Memory: 25kB
           -> Partial HashAggregate (actual rows=3 loops=3)
               Group Key: fare_conditions
               Batches: 1  Memory Usage: 24kB
               Worker 0:  Batches: 1  Memory Usage: 24kB
               Worker 1:  Batches: 1  Memory Usage: 24kB
               -> Parallel Seq Scan on ticket_flights (actual rows=348575 loops=3)
 Planning Time: 0.065 ms
 Execution Time: 222.978 ms
(18 строк)
```

План говорит о том, что здесь используется технология параллельного выполнения запросов. На рис. 3.1 план представлен в виде дерева.

Выполнение плана начинается с нижнего уровня, то есть с узлов-листьев. На основе результатов их выполнения продолжают работу узлы следующего уровня иерархии. Результаты обработки строк передаются от уровня к уровню в направлении от листьев дерева плана к его корню. Каждая ветвь дерева соответствует одному из параллельных процессов. Узлы Gather Merge и Finalize GroupAggregate выполняются ведущим процессом.

В самом нижнем узле плана находится операция Parallel Seq Scan. Ключевое слово Parallel в ее имени означает, что последовательное сканирование таблицы `ticket_flights` выполняется в *параллельном* режиме. Число запланированных параллельных рабочих процессов отражается значением Workers Planned, а число таких процессов, запущенных фактически, — Workers Launched. В нашем примере эти значения одинаковы и равны 2. Однако в общем случае они могут и не совпадать: число фактически запущенных рабочих процессов может оказаться меньше запланированного. Это зависит от параметров сервера баз данных (их рассмотрение выходит за рамки книги) и от фактической его загрузки выполнением других процессов.

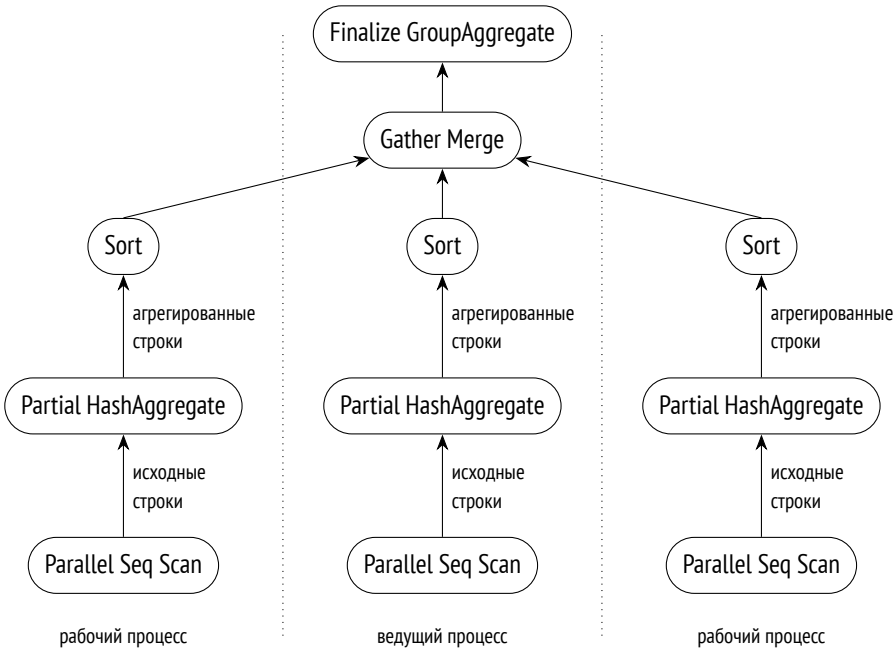


Рис. 3.1. *Графическое представление плана запроса*

Число выполнений узла Parallel Seq Scan (отражающееся значением loops) равно 3. Это говорит о том, что параллельно работали три процесса. Значение rows показывает *среднее* число строк, возвращенных каждым процессом из таблицы «Перелеты» (`ticket_flights`), общее число строк в которой составляет 1 045 726. Точное число строк для каждого рабочего процесса можно увидеть, если в команду `EXPLAIN` добавить параметр `VERBOSE`. Если выполнить эту команду несколько раз, то можно заметить, что число строк, выбираемых каждым процессом, будет незначительно меняться.

Почему мы говорим о трех процессах, когда число фактически запущенных рабочих процессов равно 2 (это значение Workers Launched)? Дело в том, что в данном случае ведущий процесс тоже участвует в работе, отбирая и группируя строки. Поскольку каждый рабочий процесс возвращает небольшое число агрегированных строк (три, по числу классов обслуживания), ведущий процесс успевает не только принять эти строки и обработать их, но и имеет возможность выполнять «черновую» работу наравне с рабочими процессами.

Над узлом Parallel Seq Scan располагается узел Partial HashAggregate. Он с помощью построения хеш-таблицы агрегирует наборы строк, полученных последовательным сканированием таблицы. При этом каждое подмножество строк агрегируется отдельно — это частичное (Partial) агрегирование, которое выполняется в каждом фоновом рабочем процессе. В результате каждый из этих трех процессов (loops=3) формирует по три строки (rows=3). Наверное, гипотетически возможна ситуация, когда одному из рабочих процессов достанутся только строки, имеющие одно и то же значение класса обслуживания — «Economy». Но, как правило, все процессы получают наборы строк, примерно одинаковые по «разнообразию» значений группируемого столбца.

Следующий узел — Sort. Здесь строки, агрегированные одним процессом, сортируются этим же процессом. Число строк, конечно, не изменяется.

Следующий узел — Gather Merge. Он выполняется одним ведущим процессом. Этот узел предназначен для сбора результатов работы фоновых рабочих процессов. Такое название узла говорит о том, что он выполняет *слияние* полученных строк. Это возможно, поскольку все частные результаты были предварительно отсортированы в узле Sort. Порядок сортировки сохраняется и при слиянии множеств строк. Данная операция формирует 9 строк (rows=9), но выполняется она всего один раз (loops=1).

Самый верхний узел плана также выполняется только ведущим процессом. Это Finalize GroupAggregate. Поскольку для этого узла в плане указано Group Key: fare_conditions, группировка производится по столбцу `fare_conditions`. В результате агрегируются наборы строк, возвращенные параллельными процессами и собранные вместе на стадии Gather Merge.

До сих пор мы использовали «традиционный» синтаксис вызова агрегатных функций. Однако PostgreSQL поддерживает еще один вариант — предложение FILTER. Этот синтаксис описан в подразделе документации 4.2.7 «Агрегатные выражения». Давайте, используя его, напишем запрос, в котором обойдемся без предложения GROUP BY.

```
SELECT
  count( flight_id ) FILTER ( WHERE fare_conditions = 'Economy' ) AS economy,
  count( flight_id ) FILTER ( WHERE fare_conditions = 'Comfort' ) AS comfort,
  count( flight_id ) FILTER ( WHERE fare_conditions = 'Business' ) AS business
FROM ticket_flights;
```

Строго говоря, новый вариант нельзя считать эквивалентным предыдущему. Если бы появился еще один класс обслуживания, то для получения корректного результата пришлось бы модифицировать запрос, добавляя в него еще один вызов функции count с конструкцией FILTER. Таким образом, структура запроса зависит от данных, представленных в таблице.

Напомним, что если в качестве параметра функции count указано имя столбца, то учитываются лишь те строки, в которых его значение отлично от NULL. Конечно, в данном случае мы могли бы вместо flight_id поставить символ *, поскольку в таблице «Перелеты» (ticket_flights) нет строк со значением NULL в этом столбце.

Получим тот же результат, что выдал предыдущий запрос. Отличие между ними только в форме вывода:

```
 economy | comfort | business
---------+---------+----------
  920793 |   17291 |   107642
(1 строка)
```

А что покажет команда EXPLAIN с параметром ANALYZE?

```
EXPLAIN ( analyze, costs off, timing off ) SELECT
  count( flight_id ) FILTER ( WHERE fare_conditions = 'Economy' ) AS economy,
  count( flight_id ) FILTER ( WHERE fare_conditions = 'Comfort' ) AS comfort,
  count( flight_id ) FILTER ( WHERE fare_conditions = 'Business' ) AS business
FROM ticket_flights;
                                QUERY PLAN
---------------------------------------------------------------------------
 Finalize Aggregate (actual rows=1 loops=1)
   -> Gather (actual rows=3 loops=1)
       Workers Planned: 2
       Workers Launched: 2
       -> Partial Aggregate (actual rows=1 loops=3)
           -> Parallel Seq Scan on ticket_flights (actual rows=348575 loops=3)
 Planning Time: 0.064 ms
 Execution Time: 178.702 ms
(8 строк)
```

Как следует из вывода команды EXPLAIN, этот вариант запроса работает немного быстрее предыдущего. Необходимо сказать, что, во-первых, время выполнения запроса на разных компьютерах может сильно различаться, поэтому следует рассматривать не абсолютное время, а лишь *соотношение* времени выполнения

различных запросов. И во-вторых, каждый запрос нужно выполнять несколько раз, чтобы получить некоторое устоявшееся время (хотя, конечно, оно не будет в точности одним и тем же при каждом выполнении запроса, поскольку даже на одном компьютере время выполнения одного и того же запроса может различаться в зависимости от его загруженности выполнением других задач). Также при первом выполнении запроса время может быть значительно больше, чем при последующих, поскольку требуемые данные могут отсутствовать в памяти и их придется прочитать с диска.

В самом нижнем узле плана так же, как и в предыдущем плане, находится операция Parallel Seq Scan. Число как запланированных, так и фактически запущенных параллельных рабочих процессов, как и прежде, равно двум. В следующем узле плана выполняется частичное агрегирование, но оно основано на сортировке строк, а не на построении хеш-таблицы, поэтому мы видим узел Partial Aggregate, а не Partial HashAggregate. В каждом из параллельных процессов частичное агрегирование фактически формирует одну строку (rows=1), поскольку вся выборка является одной группой, ведь мы не использовали предложение GROUP BY. При этом число параллельно работающих процессов будет равно трем (поэтому loops=3).

Для сбора частичных результатов служит узел Gather. Он выполняется ведущим процессом. Здесь за один проход аккумулируются три строки (rows=3 loops=1), полученные от рабочих процессов. Поскольку в данном случае каждый из частичных результатов содержит только одну строку и общий результат также состоит только из одной строки, то сортировка частичных результатов не производится. А раз так, то в качестве узла, объединяющего частичные результаты, используется Gather, а не Gather Merge, поскольку слияние не выполняется.

На верхнем уровне плана находится узел Finalize Aggregate. Он также выполняется ведущим процессом. Здесь просто суммируются значения соответственных полей трех результирующих строк, сформированных на предыдущих уровнях плана. Поскольку в данном запросе мы не использовали предложение GROUP BY, то на верхнем уровне плана используется Finalize Aggregate, а не Finalize GroupAggregate.

Обратите внимание, что в рассмотренном запросе число параллельно работающих процессов совпадает с числом агрегатных выражений — и то, и другое равно трем. Это случайное совпадение. Предлагаем читателю самостоятельно

провести простой эксперимент: оставить в запросе только два агрегатных выражения из трех и посмотреть, каким будет план выполнения.

Важно, что не все агрегатные функции позволяют организовать параллельную работу. Например, такие функции, как `min` и `max`, позволяют это сделать, поскольку можно разделить все данные на несколько частей, в каждой из них отыскать «частный» минимум или максимум, поручив эту работу параллельным процессам, и, получив от процессов найденные ими значения, найти минимальное или максимальное среди них. Такой алгоритм даст корректный результат. Однако не все агрегатные функции позволяют объединять «частные» результаты. В качестве примера можно привести функции для вычисления моды и медианы. Они будут рассмотрены в разделе 3.2 «Статистические функции» (с. 126). В разделе документации 9.21 «Агрегатные функции» для каждой стандартной агрегатной функции приведено указание, может ли она использоваться в параллельных запросах.

3.1.3. Агрегирование данных типа JSON

Теперь рассмотрим следующий вопрос, заявленный в начале раздела: агрегирование данных, представленных в формате JSON. PostgreSQL хорошо поддерживает работу с такими данными, включая и наличие агрегатных функций.

Рассмотрим функции для получения JSON-объектов из полей сгруппированных строк. Сначала покажем это на простом искусственном примере, в котором используется функция `jsonb_object_agg`:

```
WITH people_info( num, key, value ) AS
( VALUES ( 1, 'weight', 80.5 ), ( 1, 'height', 175 ), ( 1, 'age', 35 ),
         ( 2, 'weight', 76.4 ), ( 2, 'height', 183 ), ( 2, 'age', 45 ),
         ( 3, 'weight', 68.8 ), ( 3, 'height', 169 ), ( 3, 'age', 40 )
)
SELECT num, jsonb_object_agg( key, value ) AS info
FROM people_info
GROUP BY num;
 num |                    info
-----+-----------------------------------------
   1 | {"age": 35, "height": 175, "weight": 80.5}
   2 | {"age": 45, "height": 183, "weight": 76.4}
   3 | {"age": 40, "height": 169, "weight": 68.8}
(3 строки)
```

Значения полей JSON-объектов, содержащие числовые данные, поддаются агрегированию. Например, можно вычислить среднее значение поля age из предыдущего примера:

```
WITH people_info( num, key, value ) AS
( VALUES ( 1, 'weight', 80.5 ), ( 1, 'height', 175 ), ( 1, 'age', 35 ),
         ( 2, 'weight', 76.4 ), ( 2, 'height', 183 ), ( 2, 'age', 45 ),
         ( 3, 'weight', 68.8 ), ( 3, 'height', 169 ), ( 3, 'age', 40 )
),
jsonbs AS
( SELECT num, jsonb_object_agg( key, value ) AS info
  FROM people_info
  GROUP BY num
)
SELECT avg( ( info->'age' )::double precision )
FROM jsonbs;
 avg
-----
  40
(1 строка)
```

Обратите внимание на оператор ->. Он выдает значение поля JSON-объекта в виде значения типа JSON. В нашем примере этот объект имеет тип jsonb, а не json, поэтому его приведение к типу double precision выполняется корректно. Но если бы мы использовали вместо функции jsonb_object_agg функцию json_object_agg, нам пришлось бы воспользоваться оператором ->>, который выдает поле JSON-объекта в виде *текстового значения*, потому что для типа jsonb определена операция приведения к числовому типу, а для json — нет. Попутно напомним, что в документации на PostgreSQL рекомендуется использовать тип jsonb, а не json, если только у вас нет каких-то особых причин для использования именно типа json (это обосновывается в разделе документации 8.14 «Типы JSON»).

Подробно все операции с типами json и jsonb описаны в разделе документации 9.16 «Функции и операторы JSON».

Теперь рассмотрим более реалистичный пример. Предположим, что требуется организовать просмотр в удобной форме всей информации о каждом бронировании, включая все оформленные в его рамках билеты и перелеты.

Сначала приведем запрос, а затем дадим детальные пояснения к нему.

```
SELECT
  b.book_ref,
  b.book_date,
  b.total_amount,
  jsonb_pretty(
      jsonb_object_agg(
          t.ticket_no,
          jsonb_build_array(
              t.passenger_id,
              t.passenger_name,
              -- все перелеты для текущего билета из ticket_flights,
              -- отсортированные по плановому времени вылета
              ( SELECT jsonb_agg(
                    -- описание одного перелета по данным одной строки
                    jsonb_build_array(
                        f.flight_no,
                        f.scheduled_departure,
                        f.departure_city || ' - ' || f.arrival_city,
                        jsonb_build_object(
                            'fare_cond', tf.fare_conditions,
                            'amount', tf.amount
                        )
                    ) ORDER BY f.scheduled_departure
                )
                FROM ticket_flights tf
                  JOIN flights_v f ON f.flight_id = tf.flight_id -- USING ( flight_id )
                WHERE tf.ticket_no = t.ticket_no
                GROUP BY tf.ticket_no
              )
          )
      )
  ) AS tickets_info
FROM bookings b
  JOIN tickets t ON t.book_ref = b.book_ref -- USING ( book_ref )
WHERE b.book_ref = 'D56F95'
GROUP BY b.book_ref \gx
```

Хотя результат получается очень объемным, приведем его весь, чтобы была видна сложная многоуровневая структура данных. Обратите внимание на квадратные и фигурные скобки в выводе: в фигурные скобки заключаются JSON-объекты, а в квадратные скобки — JSON-массивы. При этом в JSON-массивах могут присутствовать элементы разных типов. О различиях между JSON-объектами и JSON-массивами сказано в подразделе документации 8.14.1 «Синтаксис вводимых и выводимых значений JSON».

```
-[ RECORD 1 ]+-------------------------------------------
book_ref     | D56F95
book_date    | 2017-08-15 06:23:00+07
total_amount | 45000.00
tickets_info | {                                         +
             |     "0005432384336": [                    +
             |         "1534 038771",                    +
             |         "GALINA GUSEVA",                  +
             |         [                                 +
             |             [                             +
             |                 "PG0233",                 +
             |                 "2017-09-03T16:15:00+07:00",+
             |                 "Москва - Брянск",        +
             |                 {                         +
             |                     "amount": 9900.00,    +
             |                     "fare_cond": "Business" +
             |                 }                         +
             |             ],                            +
             |             [                             +
             |                 "PG0649",                 +
             |                 "2017-09-03T19:35:00+07:00",+
             |                 "Брянск - Белгород",      +
             |                 {                         +
             |                     "amount": 3300.00,    +
             |                     "fare_cond": "Economy"  +
             |                 }                         +
             |             ],                            +
             |             [                             +
             |                 "PG0481",                 +
             |                 "2017-09-04T17:00:00+07:00",+
             |                 "Белгород - Анапа",       +
             |                 {                         +
             |                     "amount": 6300.00,    +
             |                     "fare_cond": "Economy"  +
             |                 }                         +
             |             ]                             +
             |         ]                                 +
             |     ],                                    +
             |     "0005432384337": [                    +
             |         "2864 491425",                    +
             |         "VLADIMIR KUZNECOV",              +
             |         [                                 +
             |             [                             +
             |                 "PG0233",                 +
             |                 "2017-09-03T16:15:00+07:00",+
             |                 "Москва - Брянск",        +
```

```
|                 {                           +
|                     "amount": 3300.00,      +
|                     "fare_cond": "Economy"  +
|                 }                           +
|             ],                              +
|             [                               +
|                 "PG0649",                   +
|                 "2017-09-03T19:35:00+07:00",+
|                 "Брянск - Белгород",        +
|                 {                           +
|                     "amount": 3300.00,      +
|                     "fare_cond": "Economy"  +
|                 }                           +
|             ],                              +
|             [                               +
|                 "PG0481",                   +
|                 "2017-09-04T17:00:00+07:00",+
|                 "Белгород - Анапа",         +
|                 {                           +
|                     "amount": 18900.00,     +
|                     "fare_cond": "Business" +
|                 }                           +
|             ]                               +
|         ]                                   +
|     ]                                       +
| }
```

Прежде чем приступить к разбору запроса, коснемся способа синтаксического оформления соединения таблиц. В том случае, когда оно выполняется по одноименным столбцам, присутствующим в обеих таблицах, можно использовать конструкцию `USING` вместо предложения `ON`, как показано в комментариях к запросу. Читатель может выбрать любой способ в зависимости от своих предпочтений.

Итак, значением поля `tickets_info` является JSON-объект. Он формируется с помощью агрегатной функции `jsonb_object_agg` на основе группировки строк по коду бронирования `b.book_ref`. Его ключами служат номера билетов, например `0005432384336`, а значениями являются JSON-массивы. Эти массивы формируются функцией `jsonb_build_array` и содержат номер документа пассажира (`t.passenger_id`), его имя и фамилию (`t.passenger_name`), а также вложенный JSON-массив, который формируется с помощью подзапроса. Обратите внимание, что функция `jsonb_build_array` не является агрегатной: она формирует JSON-массив из полей *одной строки* выборки. А вот агрегатная

функция jsonb_agg, вызываемая в подзапросе, формирует JSON-массив, описывающий все перелеты, включенные в данный билет. Предложение ORDER BY f.scheduled_departure относится именно к ней и позволяет упорядочить перелеты в массиве согласно плановому времени вылета. Подробно об использовании предложения ORDER BY с агрегатными функциями сказано в подразделе документации 4.2.7 «Агрегатные выражения».

Функция jsonb_build_array в подзапросе собирает все необходимые сведения об *одном* перелете: номер рейса, дата и время вылета, маршрут, стоимость перелета и класс обслуживания. Вложенная функция jsonb_build_object позволяет представить сведения о стоимости перелета (tf.amount) и классе обслуживания (tf.fare_conditions) в наглядной форме в виде JSON-объекта с ключами amount и fare_cond.

Добавим, что предложение GROUP BY tf.ticket_no добавлено в подзапрос исключительно для наглядности. Предложение WHERE tf.ticket_no = t.ticket_no и так ограничивает выборку перелетами, относящимися к одному билету, а в списке SELECT присутствует только вызов агрегатной функции и не указан ни один столбец.

Обратите внимание на формат вывода временных отметок, в котором между значениями даты и времени стоит буква T: 2017-09-03T19:35:00+07:00. Ее добавляет функция jsonb_build_array в соответствии со стандартом ISO 8601, хотя в подразделе документации 8.5.2 «Вывод даты/времени» сказано, что при выводе PostgreSQL разделяет дату и время пробелом.

Также характерно, что все ключи и все значения, кроме числовых, заключаются в двойные кавычки, а не в одинарные.

3.2. Статистические функции

Накопление больших объемов данных позволяет выявить закономерности, которым эти данные подчиняются, и получить полезную информацию для принятия решений. В этом процессе важную роль играют статистические функции, представленные в разделе документации 9.21 «Агрегатные функции» (см. таблицу 9.61 «Агрегатные функции для статистических вычислений»).

Сразу предупредим читателя, что материал этого раздела нужно рассматривать лишь как простую иллюстрацию возможностей PostgreSQL по статистической обработке данных, а не как учебное пособие по математической статистике.

Применение статистических функций рассмотрим на следующем примере. Известно, что в случаях длительных задержек рейсов авиакомпании несут имиджевые потери и вынуждены принимать затратные меры, например за свой счет размещать пассажиров в гостиницах. Поэтому представляется важным получить полную картину, сложившуюся в отношении задержек рейсов нашей авиакомпании.

За информацией обратимся к представлению «Рейсы» (`flights_v`), в котором содержатся все основные данные о рейсах, в том числе и об их задержках.

Сначала определим, с какой точностью фиксировались моменты отправлений рейсов и, следовательно, с какой точностью можно вычислить их задержки. Начнем с проверки наличия задержанных рейсов, время отправления которых (плановое или фактическое) содержит ненулевое число секунд:

```
SELECT
  count( * ) FILTER ( WHERE extract( sec FROM scheduled_departure ) > 0 )
    AS scheduled_departure_nonzero_secs,
  count( * ) FILTER ( WHERE extract( sec FROM actual_departure ) > 0 )
    AS actual_departure_nonzero_secs
FROM flights_v
WHERE actual_departure > scheduled_departure;
 scheduled_departure_nonzero_secs | actual_departure_nonzero_secs
----------------------------------+------------------------------
                                0 |                            0
(1 строка)
```

Теперь посмотрим, какие значения имеет число минут в плановом и фактическом времени вылета:

```
SELECT array_agg( DISTINCT extract( minute FROM scheduled_departure ) ) AS scheduled_mins
FROM flights_v
WHERE actual_departure > scheduled_departure;
           scheduled_mins
-------------------------------------
 {0,5,10,15,20,25,30,35,40,45,50,55}
(1 строка)
```

Число минут в плановом времени отправления рейсов кратно пяти. Однако это определяется не низкой точностью приборов для измерения времени, а произвольным решением руководства компании. Это сделано для удобства пассажиров. В принципе можно было бы *назначать* (не измерять!) число минут, равное, например, 6, 11, 16 и т. д., но это не очень удобно.

```
SELECT DISTINCT extract( minute FROM actual_departure ) AS actual_mins
FROM flights_v
WHERE actual_departure > scheduled_departure
ORDER BY actual_mins;
```

```
 actual_mins
-------------
           0
           1
           2
           3
 ...
          57
          58
          59
(60 строк)
```

Получается, что фактическое время отправления рейсов может содержать любое допустимое число минут от 0 до 59.

Как мы увидели, точность фиксации времени отправления рейсов, как планового, так и фактического, не превышает минуты, поскольку компонент «секунды» в этих столбцах всегда равен нулю. Таким образом, длительность задержек рейсов также имеет смысл вычислять только с точностью до минут.

Теперь можно перейти к анализу задержек рейсов. Начнем с вычисления основных статистических показателей выборки: среднего значения длительности задержки, дисперсии, среднеквадратического отклонения, моды и медианы.

Функция avg, вычисляющая среднее значение, нам уже хорошо известна.

Дисперсия и среднеквадратическое отклонение характеризуют величину разброса полученных значений вокруг среднего значения.

Для вычисления дисперсии предназначены две функции — var_pop и var_samp. Первая из них вычисляет дисперсию для генеральной совокупности (то есть для *всего* исследуемого множества объектов), а вторая — для выборки из этой совокупности. Здесь под выборкой понимается не множество строк, возвращенных

запросом, а подмножество объектов, выбранных случайным образом из генеральной совокупности. Поскольку мы имеем возможность исследовать *все* задержанные рейсы, которые и будут образовывать генеральную совокупность, воспользуемся функцией `var_pop`. Эта функция не может работать с типом данных `interval`, поэтому переведем значения задержек, представленные в виде интервалов, в целое число минут.

Аналогично для вычисления среднеквадратического отклонения воспользуемся функцией `stddev_pop`, а не `stddev_samp`.

Для обсуждения моды и медианы нам потребуется два понятия: дискретная случайная величина и вариационный ряд. Случайную величину называют дискретной, если множество ее возможных значений конечно либо счетно. Счетным называется бесконечное множество, элементы которого можно пронумеровать натуральными числами. Таким образом, величина задержки рейса будет являться дискретной случайной величиной. Вариационным рядом называется последовательность полученных значений случайной величины, упорядоченных по возрастанию.

Мода — это наиболее типичное значение в выборке, оно имеет наибольшую частоту встречаемости в вариационном ряде. Вычислим ее с помощью функции `mode`.

Медиана — это такое значение, которое делит вариационный ряд на две равные части (по числу элементов). Поскольку мы считаем значения задержек дискретными величинами, то для определения медианы воспользуемся функцией `percentile_disc`. Эта функция возвращает *значение* такого элемента вариационного ряда, позиция которого отделяет указанную в аргументе *долю* элементов этого ряда. Для получения медианы нужно отделить половину значений, следовательно, параметром функции будет число 0,5.

Обратите внимание на синтаксис вызова функций `mode` и `percentile_disc`. Предложение `ORDER BY` является принципиально важным, поскольку мода и медиана могут быть определены только для вариационного ряда, то есть набора значений, упорядоченных по возрастанию. Заметим попутно, что для вычисления моды и медианы необходим доступ ко всему вариационному ряду, а не к его фрагментам, поэтому распараллелить запрос, вызывающий эти функции, невозможно.

Функции mode и percentile_disc относятся к группе так называемых *сортирующих функций* (Ordered-Set Aggregate Functions), которые описаны в разделе документации 9.21 «Агрегатные функции» (см. таблицу 9.62 «Сортирующие агрегатные функции»).

```
WITH delays AS
( SELECT actual_departure - scheduled_departure AS delay
  FROM flights_v
  WHERE actual_departure > scheduled_departure
)
SELECT count( * ),
       min( delay ) AS minimal_delay,
       max( delay ) AS maximal_delay,
       avg( delay ) AS average,
       round( var_pop( extract( epoch FROM delay ) / 60 )::numeric, 2 )
         AS variance,
       round( stddev_pop( extract( epoch FROM delay ) / 60 )::numeric, 2 )
         AS standard_deviation,
       mode() WITHIN GROUP ( ORDER BY delay ),
       percentile_disc( 0.5 ) WITHIN GROUP ( ORDER BY delay ) AS median
FROM delays \gx
-[ RECORD 1 ]------+---------------
count              | 16056
minimal_delay      | 00:01:00
maximal_delay      | 04:37:00
average            | 00:12:45.09716
variance           | 1784.72
standard_deviation | 42.25
mode               | 00:03:00
median             | 00:03:00
```

Глядя на полученный результат, можно предположить, что большая часть задержек рейсов являются кратковременными. Тем не менее поскольку среднее значение больше медианы, а максимальное значение — более четырех с половиной часов, представляется целесообразным сформировать вариационный ряд задержек рейсов для получения более детальной картины.

```
SELECT ( actual_departure - scheduled_departure ) AS delay,
       count( * )
FROM flights_v
WHERE actual_departure > scheduled_departure
GROUP BY delay
ORDER BY delay;
```

Попутно заметим, что в предложениях GROUP BY и ORDER BY можно использовать имена вычисляемых столбцов, что мы и показали в этом запросе на примере столбца delay.

```
  delay   | count
----------+-------
 00:01:00 |  2249
 00:02:00 |  3645
 00:03:00 |  3651
 00:04:00 |  2801
 00:05:00 |  1653
 00:06:00 |   784
 00:07:00 |   324
 00:08:00 |   103
 00:09:00 |    34
 00:10:00 |    10
 00:11:00 |     2
 02:04:00 |     1
 02:12:00 |     1
 02:13:00 |     1
 02:15:00 |     3
 ...
 02:48:00 |     6
 02:49:00 |     6
 02:50:00 |     9
 02:51:00 |     6
 02:52:00 |    14
 02:53:00 |    15
 02:54:00 |    11
 02:55:00 |    10
 ...
 03:08:00 |    22
 03:09:00 |    10
 03:10:00 |     5
 03:11:00 |    11
 03:12:00 |    12
 03:13:00 |    15
 03:14:00 |    10
 03:15:00 |    14
 ...
 04:20:00 |     1
 04:27:00 |     1
 04:28:00 |     1
 04:37:00 |     1
(123 строки)
```

Из результатов выборки можно увидеть, что она явно разбивается на две части: короткие задержки, не превышающие одиннадцать минут, и длительные задержки, превышающие два часа. Для определенности будем считать задержки, не превышающие одного часа, короткими, а превышающие этот рубеж — длительными.

Давайте обратим внимание на короткие задержки. Получим только для них те же показатели, которые мы получали в начале исследования для всей выборки. Для упрощения запросов создадим представление, выбирающее только такие задержки. За основу возьмем общее табличное выражение из исходного запроса и добавим в предложение WHERE еще одно условие:

```
CREATE VIEW short_delays AS
SELECT actual_departure - scheduled_departure AS delay
FROM flights_v
WHERE actual_departure > scheduled_departure
  AND actual_departure <= scheduled_departure + interval '1 hour';
CREATE VIEW
```

Воспользуемся созданным представлением для вычисления требуемых статистических показателей:

```
SELECT count( * ),
       min( delay ) AS minimal_delay,
       max( delay ) AS maximal_delay,
       avg( delay ) AS average,
       round( var_pop( extract( epoch FROM delay ) / 60 )::numeric, 2 )
         AS variance,
       round( stddev_pop( extract( epoch FROM delay ) / 60 )::numeric, 2 )
         AS standard_deviation,
       mode() WITHIN GROUP ( ORDER BY delay ),
       percentile_disc( 0.5 ) WITHIN GROUP ( ORDER BY delay ) AS median
FROM short_delays \gx
-[ RECORD 1 ]------+----------------
count              | 15256
minimal_delay      | 00:01:00
maximal_delay      | 00:11:00
average            | 00:03:09.505768
variance           | 2.51
standard_deviation | 1.59
mode               | 00:03:00
median             | 00:03:00
```

Теперь мы видим, что среднее значение почти совпадает с модой и медианой.

Получить более информативную картину можно с помощью вычисления *процентилей*. Если выборку длительностей задержек упорядочить по возрастанию, а затем разбить на равные — по числу элементов — диапазоны, то значения, находящиеся на границах этих диапазонов, будут называться процентилями соответствующих уровней. Например, если выборка разбивается на десять диапазонов, в каждый из которых входит по 10 % значений, то такие процентили называются *децилями*. В этом случае мы сможем сказать, например, что 70 % значений из выборки не превышают по величине значения дециля уровня 0,7.

Теперь перейдем к конкретному запросу. Вычислим процентили (децили) для коротких задержек.

```
SELECT percentile_disc(
    ARRAY[ 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0 ]
  )
  WITHIN GROUP ( ORDER BY delay ) AS deciles
FROM short_delays;
```

```
                                        deciles
-------------------------------------------------------------------------------------------
 {00:01:00,00:02:00,00:02:00,00:03:00,00:03:00,00:03:00,00:04:00,00:04:00,00:05:00,00:11:00}
(1 строка)
```

Мы использовали функцию percentile_disc, вычисляющую дискретные процентили, так как в нашем случае интервалы времени являются дискретными и измеряются с точностью до минут. Эта функция может получать в качестве параметра массив чисел, представляющих собой доли выборки. В нашем примере вычисляются децили, поэтому и массив содержит числовые значения от 0,1 до 1,0 с шагом 0,1, то есть по 10 %. Результатом является также массив. Он содержит граничные значения величин задержек рейсов для каждого диапазона, содержащего 10 % элементов выборки.

Мы можем интерпретировать полученные результаты таким образом: 10 % всех задержек не превышают по длительности одну минуту, 20 % всех задержек не превышают по длительности две минуты, 30 % всех задержек также не превышают по длительности две минуты и т. д. И все 100 % задержек не превышают по длительности одиннадцать минут.

Обратите внимание на синтаксис вызова функции percentile_disc: здесь обязательным является предложение WITHIN GROUP, в котором необходимо указать порядок сортировки в предложении ORDER BY.

На наш взгляд, представление вычисленных децилей в виде массива не очень наглядно. Давайте попробуем развернуть массив в виде столбца таблицы с помощью функции unnest. А для того чтобы значению каждого дециля сопоставить его уровень, применим эту функцию и к массиву [0.1, 0.2, ..., 1.0]:

```
WITH deciles AS
( SELECT percentile_disc( ARRAY[ 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0 ] )
         WITHIN GROUP ( ORDER BY delay ) AS deciles
  FROM short_delays
)
SELECT unnest( ARRAY[ 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0 ] )
         AS level,
       unnest( deciles ) AS decile
FROM deciles;
 level |  decile
-------+----------
   0.1 | 00:01:00
   0.2 | 00:02:00
   0.3 | 00:02:00
   0.4 | 00:03:00
   0.5 | 00:03:00
   0.6 | 00:03:00
   0.7 | 00:04:00
   0.8 | 00:04:00
   0.9 | 00:05:00
   1.0 | 00:11:00
(10 строк)
```

Обратите внимание, что значения децилей повторяются. Разобраться, почему это происходит, можно с помощью такого запроса:

```
SELECT delay, count( * ),
       round( count( * )::numeric /
              -- общее число коротких задержек
              ( SELECT count( * ) FROM short_delays ) * 100, 2
            ) AS percent
FROM short_delays
GROUP BY delay
ORDER BY delay;
```

В этом запросе мы не только подсчитываем количество задержек каждой длительности, но и вычисляем их доли в общем количестве задержек (выраженные в процентах). Обратите внимание, что первый параметр функции round должен иметь тип numeric. Для этого достаточно привести к типу numeric только первый компонент выражения.

Итак, запрос выдает такой результат:

```
  delay   | count | percent
----------+-------+---------
 00:01:00 |  2249 |   14.74
 00:02:00 |  3645 |   23.89
 00:03:00 |  3651 |   23.93
 00:04:00 |  2801 |   18.36
 00:05:00 |  1653 |   10.84
 00:06:00 |   784 |    5.14
 00:07:00 |   324 |    2.12
 00:08:00 |   103 |    0.68
 00:09:00 |    34 |    0.22
 00:10:00 |    10 |    0.07
 00:11:00 |     2 |    0.01
(11 строк)
```

Теперь становится понятно, почему в столбце decile предыдущей выборки есть повторяющиеся значения. Поскольку доля задержек в одну минуту равна 14,74 %, а доли задержек в одну и две минуты в сумме составляют 38,63 %, то децили уровней 0,2 (20 %) и 0,3 (30 %) приходятся на задержку в две минуты. Мы видим также, что задержки длительностью от шести до одиннадцати минут в сумме не набирают даже десяти процентов от общего количества задержек. Поэтому децилем уровня 0,9 является значение пять минут.

Для группы задержек большой длительности (более одного часа) можно провести аналогичные вычисления. Давайте получим значения тех же показателей, которые мы вычисляли в начале исследования для всей выборки, только теперь для группы длительных задержек.

Как и при работе с короткими задержками, сначала создадим представление, позволяющее упростить запросы. Оно будет отличаться от представления, выбирающего короткие задержки, только условием WHERE:

```
CREATE VIEW long_delays AS
SELECT actual_departure - scheduled_departure AS delay
FROM flights_v
WHERE actual_departure > scheduled_departure + interval '1 hour';
CREATE VIEW
```

Теперь получим результаты для группы задержек большой длительности.

```
SELECT count( * ),
       min( delay ) AS minimal_delay,
       max( delay ) AS maximal_delay,
       avg( delay ) AS average,
       round( var_pop( extract( epoch FROM delay ) / 60 )::numeric, 2 )
         AS variance,
       round( stddev_pop( extract( epoch FROM delay ) / 60 )::numeric, 2 )
         AS standard_deviation,
       mode() WITHIN GROUP ( ORDER BY delay ),
       percentile_disc( 0.5 ) WITHIN GROUP ( ORDER BY delay ) AS median
FROM long_delays \gx
-[ RECORD 1 ]------+-------------
count              | 800
minimal_delay      | 02:04:00
maximal_delay      | 04:37:00
average            | 03:15:41.625
variance           | 548.52
standard_deviation | 23.42
mode               | 03:08:00
median             | 03:15:00
```

Остальные статистические характеристики, присущие группе длительных задержек рейсов, предлагаем читателю вычислить самостоятельно.

За рамками нашего рассмотрения остались такие вопросы, как вычисление коэффициента корреляции и ковариации, построение парной линейной регрессии. Соответствующие функции реализованы в PostgreSQL (см. таблицу 9.61 «Агрегатные функции для статистических вычислений» в разделе документации 9.21 «Агрегатные функции»). Однако их корректное применение потребует от читателя подготовки в области математической статистики, а настоящий учебник не предназначен для ее получения. Поэтому мы предлагаем читателям, имеющим необходимые знания в этой сфере, самостоятельно ознакомиться со статистическими функциями, не рассмотренными в настоящем учебнике.

3.3. GROUPING SETS, CUBE и ROLLUP

3.3.1. Группировка с помощью GROUPING SETS

Предположим, что служба снабжения и ремонта нашей авиакомпании запросила информацию о количестве маршрутов, обслуживаемых самолетами разных моделей в каждом аэропорту. При этом требуется получить и общее число маршрутов, обслуживаемых самолетами каждой модели. Отчет необходимо представить в таком виде:

```
           airport          |        model         | routes_count
----------------------------+----------------------+--------------
 Абакан                     | Боинг 737-300        |            1
 Абакан                     | Аэробус A319-100     |            2
 Абакан                     | Сессна 208 Караван   |            4
   Всего по а/п Абакан      |                      |            7
 Анадырь                    | Аэробус A319-100     |            4
   Всего по а/п Анадырь     |                      |            4
 Астрахань                  | Аэробус A319-100     |            1
 Астрахань                  | Бомбардье CRJ-200    |            3
   Всего по а/п Астрахань   |                      |            4
...
 Якутск                     | Аэробус A319-100     |            1
 Якутск                     | Сухой Суперджет-100  |            1
 Якутск                     | Бомбардье CRJ-200    |            3
   Всего по а/п Якутск      |                      |            5
                            | Боинг 777-300        |           10
                            | Боинг 767-300        |           26
                            | Аэробус A321-200     |           32
                            | Боинг 737-300        |           36
                            | Аэробус A319-100     |           46
                            | Сухой Суперджет-100  |          158
                            | Сессна 208 Караван   |          170
                            | Бомбардье CRJ-200    |          232
 ИТОГО                      |                      |          710
(387 строк)
```

Необходимые сведения имеются в представлении «Маршруты» (`routes`). Список аэропортов можно взять как из столбца `departure_airport_name` (аэропорты отправления), так и из столбца `arrival_airport_name` (аэропорты прибытия). Названия моделей самолетов, летающих по каждому маршруту, возьмем из представления «Самолеты» (`aircrafts`).

Для получения ответов на эти вопросы можно с помощью оператора UNION ALL объединить четыре запроса с группировкой. При этом для получения единообразной структуры формируемых строк придется вводить столбцы, содержащие значения NULL:

```
\timing on
Секундомер включён.
SELECT r.departure_airport_name AS airport, a.model, count( * ) AS routes_count
FROM routes r
  JOIN aircrafts a ON a.aircraft_code = r.aircraft_code
GROUP BY airport, model
UNION ALL ------------------------------------------------------------------------
SELECT departure_airport_name AS airport, NULL AS model, count( * ) AS routes_count
FROM routes
GROUP BY airport
UNION ALL ------------------------------------------------------------------------
SELECT NULL AS airport, a.model, count( * ) AS routes_count
FROM routes r
  JOIN aircrafts a ON a.aircraft_code = r.aircraft_code
GROUP BY a.model
UNION ALL ------------------------------------------------------------------------
SELECT NULL AS airport, NULL AS model, count( * ) AS routes_count
FROM routes
ORDER BY airport, routes_count, model;
 ...
Время: 154,733 мс
```

Надо заметить, что отчет, выводимый этим запросом, немного отличается от того, который приведен в начале раздела. Итоговые строки по каждому аэропорту не имеют пометки «Всего по а/п» в столбце airport, а в последней строке нет слова «ИТОГО». Предлагаем читателю самостоятельно модифицировать запрос, с тем чтобы устранить эти различия. В качестве одного из вариантов можно рассмотреть включение приведенного запроса в более крупный запрос в виде общего табличного выражения.

Можно ли ускорить приведенный запрос? Да, конечно. Для этого можно прибегнуть к конструкции GROUPING SETS, которая описана в подразделе документации 7.2.4 «GROUPING SETS, CUBE и ROLLUP». Она позволяет выполнить несколько группировок в рамках одного запроса, а затем объединить их результаты аналогично конструкции UNION ALL. Таким образом, выражения для группировки, использованные в трех подзапросах предыдущего запроса, вклю-

чаются в конструкцию GROUPING SETS. В четвертом подзапросе группировка выполняется по всей выборке; для такого случая в GROUPING SETS предусмотрено выражение ( ), то есть пустые скобки.

```
SELECT CASE
    WHEN r.departure_airport_name IS NULL AND a.model IS NULL
      THEN 'ИТОГО'
    WHEN a.model IS NULL
      THEN '  Всего по а/п ' || r.departure_airport_name
    ELSE r.departure_airport_name
  END AS airport,
  a.model,
  count( * ) AS routes_count
FROM routes r
  JOIN aircrafts a ON a.aircraft_code = r.aircraft_code
GROUP BY GROUPING SETS
( ( r.departure_airport_name, a.model ),
  ( r.departure_airport_name ),
  ( a.model ),
  ()
)
ORDER BY airport, routes_count, a.model;
 ...
Время: 41,711 мс
\timing off
Секундомер выключен.
```

Ускорение налицо. Выяснить его причину поможет упражнение 10 (с. 199).

Как видно из запроса, первая группировка производится по двум столбцам: r.departure_airport_name и a.model. Во второй и третьей группировках используется по одному столбцу: r.departure_airport_name и a.model соответственно. Таким образом, в списке столбцов SELECT будет находиться *объединение* списков столбцов, участвующих в группировках. При выполнении запроса формируются подмножества результирующих строк в соответствии с каждой группировкой, представленной в конструкции GROUPING SETS, а затем эти подмножества объединяются в итоговую выборку. При этом в тех столбцах, имена которых *не представлены* в данной группировке, будут находиться значения NULL. Это важное замечание. Например, при группировке в соответствии с набором (a.model) в поле r.departure_airport_name сформированных строк будут значения NULL.

Результат выполнения запроса будет таким:

```
              airport               |        model        | routes_count
------------------------------------+---------------------+--------------
 Абакан                             | Боинг 737-300       |            1
 Абакан                             | Аэробус A319-100    |            2
 Абакан                             | Сессна 208 Караван  |            4
 Анадырь                            | Аэробус A319-100    |            4
 ...
 Воронеж                            | Бомбардье CRJ-200   |            2
   Всего по а/п Абакан              |                     |            7
   Всего по а/п Анадырь             |                     |            4
   Всего по а/п Астрахань           |                     |            4
 ...
   Всего по а/п Якутск              |                     |            5
 Геленджик                          | Сессна 208 Караван  |            2
 Геленджик                          | Бомбардье CRJ-200   |            3
 ...
 Иркутск                            | Бомбардье CRJ-200   |            4
 ИТОГО                              |                     |          710
 Йошкар-Ола                         | Бомбардье CRJ-200   |            1
 ...
 Якутск                             | Бомбардье CRJ-200   |            3
                                    | Боинг 777-300       |           10
 ...
                                    | Сессна 208 Караван  |          170
                                    | Бомбардье CRJ-200   |          232
(387 строк)
```

В полученной выборке итоги по аэропортам оказались собранными в единый блок, размещенный после строки для аэропорта Воронеж. Строка с общим итогом также оказалась на ненадлежащем месте (между строками для аэропортов Иркутск и Йошкар-Ола).

Каким образом можно исправить ситуацию? Нужно вспомнить, что в описании команды SELECT, приведенном в документации, сказано: в предложении ORDER BY можно указывать не только имена или номера *выходных столбцов* из списка SELECT (это могут быть и вычисляемые столбцы), но и произвольные *выражения от входных столбцов*, то есть столбцов исходных таблиц. При этом строки могут быть уже сгруппированы. Вполне допустимо сортировать выборку по тем столбцам, которых нет в списке SELECT.

В предложении ORDER BY находится выходной вычисляемый столбец airport, в котором присутствуют не только названия аэропортов, но и строки «Всего по а/п ...» и «ИТОГО». Все эти значения сортируются вместе, поэтому итоговые

строки по аэропортам оказываются собранными в единый блок. Нужно заменить столбец airport на входной столбец r.departure_airport_name. Тогда итоговые строки для каждого аэропорта не будут отделяться от строк, показывающих вклад отдельных моделей самолетов, летающих из него. При этом итоговая строка для аэропорта будет располагаться *после* этих строк, поскольку вторым столбцом в предложении ORDER BY идет поле routes_count, а его итоговое значение для конкретного аэропорта будет больше любого из значений routes_count для отдельных моделей самолетов.

Но если из аэропорта совершают полеты самолеты только одной модели, значение поля routes_count для этой модели и итоговое значение routes_count для этого аэропорта совпадают, как в нашем примере с Анадырем. Как добиться, чтобы в таких случаях итоговая строка все же располагалась *после* единственной модели самолета?

В строках, сгруппированных по набору столбцов (r.departure_airport_name), значение поля a.model будет равно NULL. При сортировке *по возрастанию* эти значения NULL будут по умолчанию располагаться *в конце* множества строк, как будто они являются самыми большими из всех значений. Поэтому строки, содержащие итоговые значения числа маршрутов для каждого аэропорта, будут располагаться в конце групп строк, соответствующих каждому аэропорту.

```
...
ORDER BY r.departure_airport_name, routes_count, a.model;
```

Группа строк, содержащих суммарное количество маршрутов для каждой модели самолетов, находится в конце выборки. Это достигается также благодаря тому, что в поле r.departure_airport_name в этих строках находятся значения NULL. А значения NULL, в свою очередь, оказываются в этих строках потому, что они являются продуктом группирования по набору (a.model), в котором отсутствует столбец r.departure_airport_name.

Ну и наконец, итоговая строка, содержащая общее число маршрутов, располагается в самом конце выборки. Эта строка получена при группировании по пустому набору столбцов, то есть по всей выборке, поэтому в ней значения полей r.departure_airport_name и a.model будут равны NULL. Для наглядности в ее выходном поле airport выводится значение «ИТОГО», но в ее входном (исходном) поле r.departure_airport_name, которое является первым в предложении ORDER BY, находится значение NULL. Такое же значение этого поля имеют и строки,

соответствующие итогам, вычисленным для каждой модели самолета. Поэтому расположение данной строки определяется значением поля routes_count, а оно наибольшее среди всех строк.

3.3.2. Группировка с помощью ROLLUP

Для двух типичных ситуаций существуют сокращенные варианты записи конструкции GROUPING SETS. Они оба описаны в подразделе документации 7.2.4 «GROUPING SETS, CUBE и ROLLUP». Первый вариант такой:

```
ROLLUP ( a, b, c, ... )
```

Эта запись равнозначна следующей конструкции:

```
GROUPING SETS
( ( a, b, c, ... ),
  ...
  ( a, b ),
  ( a ),
  ()
)
```

Проиллюстрируем это на примере. Предположим, что руководство компании хотело бы видеть динамику бронирований авиабилетов с разбиением по дням, декадам и месяцам. Необходимо отражать как количество бронирований, так и общие суммы этих бронирований (в миллионах рублей). Конечно, общий итог за отчетный период также необходимо подводить. Уточним, что третья декада может иметь длительность от 8 до 11 дней в зависимости от месяца.

```
SELECT extract( month FROM book_date ) AS month,
       CASE WHEN extract( day FROM book_date ) <= 10 THEN 1
            WHEN extract( day FROM book_date ) <= 20 THEN 2
            ELSE 3
       END AS ten_days,
       extract( day FROM book_date ) AS day,
       count( * ) AS book_num,
       round( sum( total_amount ) / 1000000, 2 ) AS amount
FROM bookings
GROUP BY ROLLUP( month, ten_days, day )
ORDER BY month, ten_days, day;
```

Получим такую выборку:

```
 month | ten_days | day | book_num |  amount
-------+----------+-----+----------+----------
     6 |        3 |  21 |        3 |     0.44
     6 |        3 |  22 |        6 |     0.56
     6 |        3 |  23 |       17 |     1.82
     6 |        3 |  24 |       67 |     5.47
     6 |        3 |  25 |      169 |    13.69
     6 |        3 |  26 |      328 |    26.46
     6 |        3 |  27 |      686 |    49.48
     6 |        3 |  28 |     1102 |    83.89
     6 |        3 |  29 |     1885 |   145.31
     6 |        3 |  30 |     2582 |   196.23
     6 |        3 |     |     6845 |   523.35
     6 |          |     |     6845 |   523.35
     7 |        1 |   1 |     3425 |   267.22
     7 |        1 |   2 |     4066 |   321.96
 ...
     7 |        3 |  29 |     5489 |   425.30
     7 |        3 |  30 |     5432 |   427.33
     7 |        3 |  31 |     5573 |   444.18
     7 |        3 |     |    60718 |  4791.76
     7 |          |     |   166611 | 13193.57
     8 |        1 |   1 |     5510 |   444.54
     8 |        1 |   2 |     5569 |   446.35
 ...
     8 |        2 |  15 |     6449 |   483.56
     8 |        2 |     |    32217 |  2492.42
     8 |          |     |    89332 |  7050.06
       |          |     |   262788 | 20766.98
(66 строк)
```

Прежде чем продолжить, сделаем небольшое отступление, связанное с тем, что операции бронирования авиабилетов производятся в разных часовых поясах. В таблице «Бронирования» (bookings) столбец book_date имеет тип данных timestamp with time zone. Поэтому при записи в базу данных время выполнения всех операций приводится к одной точке отсчета — времени по Гринвичу. При выполнении выборок дата и время бронирований выводятся в соответствии с настройкой часового пояса на компьютере пользователя. Благодаря этому операции бронирования, произведенные в разных часовых поясах в одно и то же астрономическое время, при просмотре записей получат одно и то же значение поля book_date. А выводимое значение этого поля зависит от настройки часового пояса в системе, в которой выполняется выборка.

Для просмотра этой настройки можно воспользоваться командой SHOW:

```
SHOW timezone;
     TimeZone
------------------
 Asia/Krasnoyarsk
(1 строка)
```

Изменить настройку на время сеанса работы можно с помощью команды SET:

```
SET timezone = 'Europe/Moscow';
SET
```

Читатель может увидеть результат изменения настройки часового пояса, сделав выборку из таблицы «Бронирования» (bookings) и посмотрев на значения столбца book_date до ее изменения и после.

Рассмотрим простой пример. Предположим, что две операции бронирования были произведены одновременно, причем одна из них — в Красноярске, а другая — во Владивостоке. Обратите внимание, что во Владивостоке уже наступил следующий день. При выводе мы получаем значения отметок времени с учетом настройки часового пояса на компьютере пользователя, то есть по московскому времени. Видно, что время выполнения этих операций совпадает.

```
SELECT '2024-08-13 22:00:00+07'::timestamptz;
      timestamptz
------------------------
 2024-08-13 18:00:00+03
(1 строка)
SELECT '2024-08-14 01:00:00+10'::timestamptz;
      timestamptz
------------------------
 2024-08-13 18:00:00+03
(1 строка)
```

Приведенные рассуждения нужно учитывать, выполняя запросы к базе данных, в которых используются значения типа timestamptz. В зависимости от настройки часового пояса на вашем компьютере могут получиться результаты, отличающиеся от приведенных в книге, поскольку при изменении этой настройки конкретные операции бронирования могут «переходить» на следующий (или предыдущий) день.

В книге используется часовой пояс Красноярска; вернем исходное значение настройки:

```
SET timezone = 'Asia/Krasnoyarsk';
SET
```

В запросе присутствует конструкция:

```
...
GROUP BY ROLLUP( month, ten_days, day )
...
```

Вместо нее будут сформированы и обработаны четыре набора группировок в конструкции GROUPING SETS:

```
...
GROUP BY GROUPING SETS
( ( month, ten_days, day ),
  ( month, ten_days ),
  ( month ),
  ()
)
...
```

Убедиться в этом можно, посмотрев планы выполнения запросов с помощью команды EXPLAIN.

Может показаться, что в группировке (month, ten_days, day) столбец ten_days является лишним, и в силу этого вся конструкция ROLLUP будет избыточной. Однако его наличие позволяет значительно упростить вывод результирующей выборки в правильном порядке. Предлагаем читателю самостоятельно провести эксперимент, заменив конструкцию ROLLUP на GROUPING SETS и удалив этот столбец из группировки.

Обратите внимание, что последовательность значений в столбце amount не является точным примером того, что называется накопленной суммой (running sum, running total). Промежуточные — накопленные — итоги подводятся только по каждой декаде и по каждому месяцу, а не каждый день.

3.3.3. Группировка с помощью CUBE

Теперь рассмотрим вторую типовую конструкцию — CUBE. Она выглядит так:

```
CUBE ( a, b, ... )
```

Это будет равнозначно конструкции GROUPING SETS, в которой представлены все подмножества исходного множества столбцов группировки, то есть булеа́н этого множества, как говорят математики.

Рассмотрим конкретный пример. Предположим, что в предложении GROUP BY запроса присутствует следующая запись:

```
CUBE ( a, b, c )
```

Она будет равнозначна такой записи:

```
GROUPING SETS
( ( a, b, c ),
  ( a, b    ),
  ( a,    c ),
  ( a       ),
  (    b, c ),
  (    b    ),
  (       c ),
  (         )
)
```

Обратите внимание, что для каждого подмножества, содержащего более одного столбца, приводится только один вариант записи этого подмножества. Это объясняется тем, что в множестве элементы не упорядочены. Поэтому, например, если представлена комбинация (a, c), то комбинация (c, a) уже не нужна. Это будет корректно работать в запросах, поскольку в предложении GROUP BY порядок следования выражений не важен.

В качестве элемента в приведенных конструкциях может выступать и *список* выражений, как сказано в подразделе документации 7.2.4 «GROUPING SETS, CUBE и ROLLUP».

Начнем рассмотрение конструкции CUBE с абстрактного примера. Обратите внимание, что в исходных данных есть значения NULL, соответствующие столбцу key2:

```
WITH t( key1, key2, val ) AS
( VALUES ( 'a', 'x', 1 ), ( 'a', 'x', 2 ), ( 'a', 'y', 4 ),
         ( 'a', NULL, 8 ), ( 'a', NULL, 16 ),
         ( 'b', 'x', 32 ), ( 'b', 'x', 64 ), ( 'b', 'y', 128 )
)
SELECT key1, key2, sum( val )
FROM t
GROUP BY CUBE ( key1, key2 )
ORDER BY key1, key2, sum( val );
```

```
 key1 | key2 | sum
------+------+-----
 a    | x    |   3
 a    | y    |   4
 a    |      |  24
 a    |      |  31
 b    | x    |  96
 b    | y    | 128
 b    |      | 224
      | x    |  99
      | y    | 132
      |      |  24
      |      | 255
(11 строк)
```

При преобразовании конструкции CUBE в эквивалентную ей GROUPING SETS получается такая картина:

```
GROUP BY GROUPING SETS
( ( key1, key2 ),
  ( key1 ),
  ( key2 ),
  ( )
)
```

Группировки проводятся по различным наборам столбцов, но результат обязан быть обычной таблицей. Чтобы сохранить регулярную структуру, в тех строках результата, которые получены группировкой по неполному набору столбцов, не участвующие в группировке поля получают значения NULL. Они находятся как раз в тех полях, которые в группировке не участвовали. Назовем такие значения NULL *порожденными* в отличие от тех, которые присутствовали в исходных данных изначально.

В полученной выборке присутствуют значения NULL. Возникает вопрос: какие из них попали в выборку из исходных данных, а какие являются порожденными? Например, в строках, содержащих в поле sum значение 24, значение NULL в поле key2 взято из исходных данных. А в строке, содержащей в поле sum значение 31, значение NULL в поле key2 появляется потому, что столбец key2 не участвовал в группировке, сформировавшей эту строку. Предпоследняя строка выборки в поле key1 содержит порожденное значение NULL, а в поле key2 — значение NULL, взятое из исходных данных. Происхождение значений NULL в последней строке читатель может выяснить самостоятельно.

Иногда возникает необходимость определять происхождение значений NULL программным путем, то есть непосредственно в запросе, а не с помощью последующих рассуждений. Это позволит выяснить, является ли полученная строка результатом группировки и какие столбцы участвовали в этой группировке. Если значение NULL в конкретном поле строки — порожденное, значит, данный столбец не участвовал в группировке, сформировавшей эту строку.

Помочь в решении задачи может функция GROUPING, представленная в разделе документации 9.21 «Агрегатные функции» (см. таблицу 9.64 «Операции группировки»). Она формирует целое число на основе битовой маски, в которой каждая позиция соответствует одному параметру этой функции. При этом последнему параметру соответствует самый младший бит. Конкретный бит принимает значение 1, если соответствующий ему параметр (то есть столбец или выражение на его основе) *не участвовал* в том наборе группировки, на основе которого сформирована результирующая строка, а в этом поле появится порожденное значение NULL. В противном случае значением бита будет 0, а если в соответствующем поле строки и появится значение NULL, то его источником будут исходные данные.

Давайте воспользуемся функцией GROUPING. При этом вызовем ее в списке SELECT не один раз для всей строки, а для каждого столбца по отдельности. В этом случае возвращаемое ею значение будет легче интерпретировать, поскольку оно будет совпадать со значением единственного бита в битовой маске.

Имя функции в запросе написано в верхнем регистре, потому что так сделано в документации, однако можно использовать и нижний регистр символов.

```
WITH t( key1, key2, val ) AS
( VALUES ( 'a', 'x', 1 ), ( 'a', 'x', 2 ), ( 'a', 'y', 4 ),
         ( 'a', NULL, 8 ), ( 'a', NULL, 16 ),
         ( 'b', 'x', 32 ), ( 'b', 'x', 64 ), ( 'b', 'y', 128 )
)
SELECT
  key1, GROUPING( key1 ) AS key1_g,
  key2, GROUPING( key2 ) AS key2_g,
  sum( val )
FROM t
GROUP BY CUBE ( key1, key2 )
ORDER BY key1, key2, sum( val );
```

```
 key1 | key1_g | key2 | key2_g | sum
------+--------+------+--------+-----
 a    |      0 | x    |      0 |   3
 a    |      0 | y    |      0 |   4
 a    |      0 |      |      0 |  24
 a    |      0 |      |      1 |  31
 b    |      0 | x    |      0 |  96
 b    |      0 | y    |      0 | 128
 b    |      0 |      |      1 | 224
      |      1 | x    |      0 |  99
      |      1 | y    |      0 | 132
      |      1 |      |      0 |  24
      |      1 |      |      1 | 255
(11 строк)
```

Давайте снова посмотрим на строки, в которых поле sum содержит значения 24 и 31, а значения поля key2 равны NULL. Функция GROUPING для этого поля вернула значение 0 для строк с sum = 24.Следовательно, в этих строках значение NULL взято из исходных данных, а столбец key2 участвовал в группировке при формировании этой строки. А вот для строки с sum = 31 функция GROUPING вернула значение 1, поэтому значение NULL было порождено в процессе группировки, в которой столбец key2 не участвовал.

Возьмем последнюю строку выборки. Функция GROUPING вернула 1 для обоих столбцов, значит, эта строка сформирована на основе всех исходных строк, то есть ни один из столбцов key1 и key2 в группировке не участвовал.

Таким образом, если мы видим в конкретном поле строки NULL, а соответствующее значение функции GROUPING для этого поля равно 1, то столбец в группировке не участвовал.

Теперь рассмотрим такую ситуацию: руководство нашей авиакомпании просит представить детальный отчет о распределении продаж билетов по направлениям, моделям самолетов и классам обслуживания. Под направлением будем понимать пару «аэропорт отправления — аэропорт прибытия». Еще одно важное уточнение: строго говоря, в этом запросе фигурируют не билеты, а перелеты, которые являются составными элементами билетов. Но употребление слова «билеты» более интуитивное.

В качестве, условно говоря, базовой таблицы используем представление «Маршруты» (routes), содержащее сведения о парах аэропортов, образующих

направления полетов, и о моделях самолетов, летающих по каждому маршруту. Сведения об интересующих нас классах обслуживания, назначенных для каждого перелета конкретного пассажира, содержатся в таблице «Перелеты» (ticket_flights), но соединить ее непосредственно с представлением «Маршруты» (routes) нельзя, так как у них нет общих атрибутов. Поэтому воспользуемся таблицей «Рейсы» (flights) в качестве посредника. Наконец, таблица «Самолеты» (aircrafts) нужна только для того, чтобы вместо кода самолета, содержащегося в представлении «Маршруты» (routes), выводить наименование модели самолета.

Опять воспользуемся функцией GROUPING. Однако в этот раз будем формировать единую битовую маску для всех столбцов, фигурирующих в конструкции CUBE, чтобы продемонстрировать и такой подход. Тем не менее включим в запрос и вызовы функции для каждого отдельного столбца, но закомментируем их с целью уменьшения ширины выборки. Предлагаем читателю самостоятельно провести эксперименты, удалив комментарии.

Значение, которое возвращает функция GROUPING для конкретной строки, зависит от того, в каком порядке ей переданы фактические параметры — имена группируемых столбцов. Интерпретация возвращаемого значения, видимо, будет наиболее естественной, если этот порядок совпадает с тем, в каком имена столбцов перечислены в конструкции GROUP BY CUBE.

Битовая маска не очень наглядно показывает, какие столбцы (выражения) из предложения GROUP BY использовались для формирования конкретной агрегированной строки. Мы предусмотрели в выборке столбец grouped_cols, в котором будет выводиться список имен этих столбцов. Формировать его будем, используя вызовы функции GROUPING для каждого столбца, включенного в предложение GROUP BY. Полученные имена столбцов объединим с помощью функции concat_ws, которая конкатенирует значения параметров, отличных от NULL, используя в качестве разделителя значение своего первого параметра.

Конечно, в принципе можно получать значения столбца grouped_cols путем сопоставления битовой маски со списком всех столбцов из предложения GROUP BY, но такое решение будет более сложным.

Приведем запрос, а затем дадим детальные пояснения к нему.

```
SELECT
  r.departure_airport AS da,
  -- GROUPING( r.departure_airport ) AS da_g,
  r.arrival_airport AS aa,
  -- GROUPING( r.arrival_airport ) AS aa_g,
  a.model,
  -- GROUPING( a.model ) AS m_g,
  left( tf.fare_conditions, 1 ) AS fc,
  -- GROUPING( tf.fare_conditions ) AS fc_g,
  count( * ),
  round( sum( tf.amount ) / 1000000, 2 ) AS t_amount,
  GROUPING( r.departure_airport, r.arrival_airport, a.model, tf.fare_conditions )::bit( 4 )
    AS mask,
  concat_ws( ',',
    CASE WHEN GROUPING( r.departure_airport ) = 0 THEN 'da' END,
    CASE WHEN GROUPING( r.arrival_airport ) = 0 THEN 'aa' END,
    CASE WHEN GROUPING( a.model ) = 0 THEN 'm' END,
    CASE WHEN GROUPING( tf.fare_conditions ) = 0 THEN 'fc' END
  ) AS grouped_cols
FROM routes r
  JOIN flights f ON f.flight_no = r.flight_no
  JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  JOIN aircrafts a ON a.aircraft_code = r.aircraft_code
GROUP BY CUBE
( ( da, aa ),
  a.model,
  tf.fare_conditions
)
ORDER BY da, aa, a.model, fc;
```

В разделе документации 9.21 «Агрегатные функции» сказано, что аргументы функции GROUPING на самом деле не вычисляются, но они должны в точности соответствовать выражениям, заданным в предложении GROUP BY на их уровне запроса. Поэтому, например, нельзя передать в качестве аргумента выражение left(tf.fare_conditions, 1) вместо имени столбца tf.fare_conditions. Использовать в качестве параметров псевдонимы столбцов, объявленные в этом же списке SELECT, также не получится.

Правильная сортировка выборки обеспечивается за счет того, что в некоторых полях отдельных строк имеют место значения NULL. Они появляются в тех строках, которые получены при группировании не по всем столбцам, перечисленным в конструкции GROUP BY CUBE. Важно, что в исходных таблицах для этих столбцов предусмотрены ограничения NOT NULL, поэтому значения NULL могут появиться только в сгруппированных строках.

Обратите внимание, что в этом запросе в качестве первого элемента конструкции CUBE выступает пара столбцов r.departure_airport и r.arrival_airport (точнее, псевдонимов da и aa), заключенная в скобки. При преобразовании конструкции CUBE в эквивалентную конструкцию GROUPING SETS такая пара (в общем случае — список элементов) выступает в качестве единого элемента. Об этом сказано в подразделе документации 7.2.4 «GROUPING SETS, CUBE и ROLLUP». Таким образом, в результате не будет групп столбцов (da) и (aa). Однако *внутри* конструкции GROUPING SETS и при вызове функции GROUPING эта же пара столбцов в скобки не заключается, то есть они рассматриваются уже как отдельные параметры.

```
...
GROUP BY GROUPING SETS
( ( da, aa, model, fc ),
  ( da, aa, model ),
  ( da, aa, fc ),
  ( da, aa ),
  ( model, fc ),
  ( model ),
  ( fc ),
  ()
)                    )
...
```

В упомянутом подразделе документации приводится напоминание о том, что конструкция вида (a, b) обычно воспринимается в выражениях как конструктор строки. Однако в предложении GROUP BY это правило не применяется на верхнем уровне выражений, и запись (a, b) воспринимается как список выражений. Если бы нам по какой-либо причине потребовался именно конструктор строки в предложении GROUP BY, нужно было бы использовать запись ROW(a, b). Конечно, в таком случае нужно было бы и в списке SELECT объединить выражения аналогичным образом.

В приведенном запросе используются псевдонимы для некоторых столбцов. Это сделано с целью уменьшения ширины заголовков. Денежные суммы представлены в миллионах рублей.

Более наглядным было бы использование названий аэропортов, а не их кодов, но коды выбраны из соображений компактности выводимой выборки.

```
 da  | aa  |        model         | fc |  count  | t_amount | mask | grouped_cols
-----+-----+----------------------+----+---------+----------+------+--------------
 AAQ | EGO | Сухой Суперджет-100 | B  |     534 |    10.09 | 0000 | da,aa,m,fc
 AAQ | EGO | Сухой Суперджет-100 | E  |    3714 |    23.52 | 0000 | da,aa,m,fc
 AAQ | EGO | Сухой Суперджет-100 |    |    4248 |    33.62 | 0001 | da,aa,m
 AAQ | EGO |                      | B  |     534 |    10.09 | 0010 | da,aa,fc
 AAQ | EGO |                      | E  |    3714 |    23.52 | 0010 | da,aa,fc
 AAQ | EGO |                      |    |    4248 |    33.62 | 0011 | da,aa
 AAQ | SVO | Боинг 737-300        | B  |     495 |    18.12 | 0000 | da,aa,m,fc
 AAQ | SVO | Боинг 737-300        | E  |    4759 |    58.35 | 0000 | da,aa,m,fc
 AAQ | SVO | Боинг 737-300        |    |    5254 |    76.47 | 0001 | da,aa,m
 AAQ | SVO |                      | B  |     495 |    18.12 | 0010 | da,aa,fc
 AAQ | SVO |                      | E  |    4759 |    58.35 | 0010 | da,aa,fc
 AAQ | SVO |                      |    |    5254 |    76.47 | 0011 | da,aa
 ...
 YKS | MJZ | Бомбардье CRJ-200    | E  |    1946 |    15.88 | 0000 | da,aa,m,fc
 YKS | MJZ | Бомбардье CRJ-200    |    |    1946 |    15.88 | 0001 | da,aa,m
 YKS | MJZ |                      | E  |    1946 |    15.88 | 0010 | da,aa,fc
 YKS | MJZ |                      |    |    1946 |    15.88 | 0011 | da,aa
     |     | Аэробус A319-100     | B  |    9055 |  1028.20 | 1100 | m,fc
     |     | Аэробус A319-100     | E  |   43798 |  1677.96 | 1100 | m,fc
     |     | Аэробус A319-100     |    |   52853 |  2706.16 | 1101 | m
 ...
     |     | Сессна 208 Караван   | E  |   14672 |    96.37 | 1100 | m,fc
     |     | Сессна 208 Караван   |    |   14672 |    96.37 | 1101 | m
     |     | Сухой Суперджет-100 | B  |   45415 |  1520.85 | 1100 | m,fc
     |     | Сухой Суперджет-100 | E  |  320283 |  3593.63 | 1100 | m,fc
     |     | Сухой Суперджет-100 |    |  365698 |  5114.48 | 1101 | m
     |     |                      | B  |  107642 |  5505.18 | 1110 | fc
     |     |                      | C  |   17291 |   566.12 | 1110 | fc
     |     |                      | E  |  920793 | 14695.68 | 1110 | fc
     |     |                      |    | 1045726 | 20766.98 | 1111 |
(2337 строк)
```

В полученной выборке представлены строки, соответствующие всем восьми подмножествам исходного множества столбцов, включенных в конструкцию CUBE. Проверить это можно, выбрав уникальные (DISTINCT) значения столбца grouped_cols. Предлагаем читателю выполнить в качестве упражнения соответствующую модификацию рассматриваемого запроса.

Обратите внимание на предпоследний столбец, в котором представлена битовая маска, возвращаемая функцией GROUPING, и на последний столбец, содержащий имена столбцов, участвовавших в группировке, сформировавшей конкретную строку выборки.

Итак, комбинация (da, aa), a.model, tf.fare_conditions): битовая маска — 0000, так как эта строка сформирована на основе наиболее полного набора группируемых столбцов.

Комбинация (da, aa), a.model): битовая маска — 0001, так как набор группируемых столбцов для этой строки не содержит *последнего* столбца — tf.fare_conditions, — которому соответствует *самый младший* бит маски.

Для остальных комбинаций группируемых столбцов нужно использовать аналогичные рассуждения.

Таким образом, мы получили полный отчет с помощью одного запроса. Надо сказать, что точно такой же результат (конечно, за исключением битовой маски) можно получить и с помощью операции UNION ALL и восьми подзапросов, но это решение более громоздкое. А будет ли оно более быстрым? Этот вопрос рассмотрен в упражнении 14 (с. 206).

3.4. Оконные функции

Оконные функции уже были кратко рассмотрены в главе 6 первой части учебника[1]. Сейчас лишь напомним основные понятия оконных функций, а в следующих подразделах более детально рассмотрим ряд конкретных аспектов этой технологии.

Оконные функции имеют много общего с агрегатными, но между ними есть и принципиальные различия. При обычном агрегировании каждая группа строк, формируемая в соответствии с предложением GROUP BY, *заменяется одной* строкой. Но при использовании оконных функций *сохраняются все* индивидуальные строки.

В разделе документации 3.5 «Оконные функции» сказано, что вызовы этих функций допускаются только в предложении SELECT (в списке вывода) и в предложении ORDER BY команды SELECT. Во всех остальных предложениях этой команды, включая WHERE, GROUP BY и HAVING, их вызывать нельзя, поскольку

[1] Моргунов, Е. П. PostgreSQL. Основы языка SQL. — СПб. : БХВ-Петербург, 2018. — 336 с. — ISBN 978-5-9775-4022-3.

оконные функции логически выполняются после обработки этих предложений. Таким образом, если в команде SELECT присутствует предложение GROUP BY (и, возможно, HAVING), то оконные функции имеют дело с уже сгруппированными строками. Пример приведен в подразделе 3.4.3 «Совместное использование оконных и агрегатных
функций» (с. 173).

Если в запросе присутствуют и агрегатные функции, и оконные, то оконные вызываются *после* агрегатных. Поэтому в принципе можно включить вызов агрегатной функции в качестве параметра оконной функции, а вот поступить наоборот — нельзя.

В технологии оконных функций базовым является понятие *раздела* (partition). Раздел включает в себя все строки, имеющие одинаковые значения определенного выражения, вычисляемого для каждой строки из выборки. Это может быть, например, значение одного или нескольких столбцов или значение какой-либо функции. Разделы определяются предложением PARTITION BY в конструкции OVER.

В процессе обработки раздела каждая его строка поочередно помещается, образно говоря, в фокус внимания; для нее некоторым образом определяются связанные с ней строки, которые и учитываются при вычислении оконной функции. Такая строка считается текущей, а связанные с ней строки образуют ее *оконный кадр* (window frame). Это второе базовое понятие технологии оконных функций. Существует целый ряд способов для задания правил формирования оконного кадра. Они будут показаны в подразделе 3.4.2 «Способы формирования оконного кадра» (с. 164), а сейчас лишь напомним, какие строки образуют его по умолчанию.

Если в конструкции OVER присутствует предложение ORDER BY, определяющее порядок сортировки, то это будут строки раздела, начиная с первой и заканчивая текущей (рис. 3.2). Бывают ситуации, когда кадр не завершается на текущей строке, а включает в себя еще одну или более строк, располагающихся в выборке *после* текущей строки. Это происходит, когда значения выражения сортировки этих строк и текущей строки совпадают. Такие строки называются *родственными* (peer).

Если же предложение ORDER BY в конструкции OVER отсутствует, то оконный кадр текущей строки образуют все строки раздела, в котором эта строка находится,

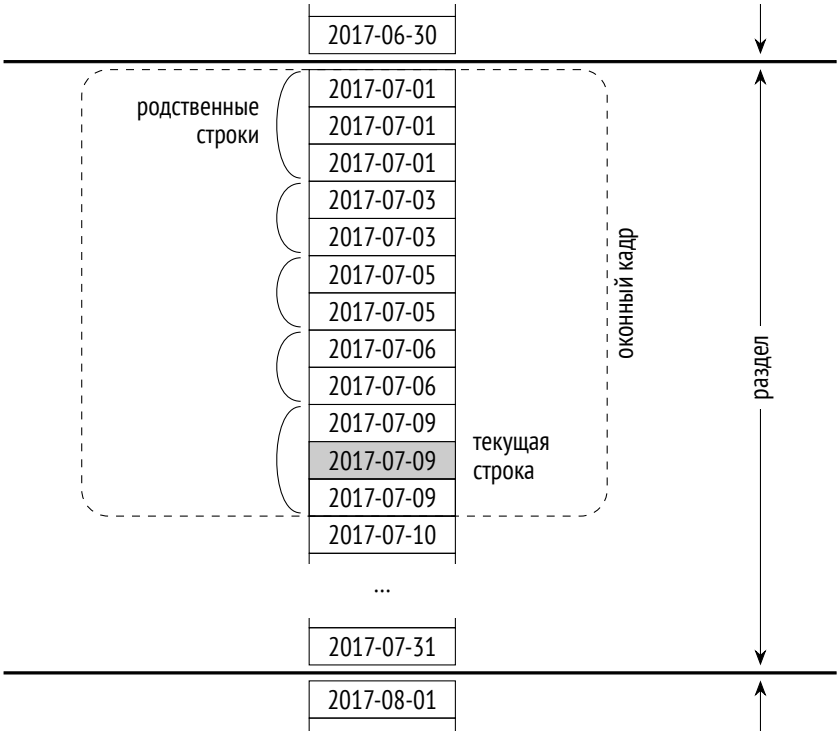


Рис. 3.2. *Оконный кадр по умолчанию с предложением ORDER BY*

поскольку все они будут считаться родственными ей (рис. 3.3).

Таким образом, оконная функция получает доступ ко всем строкам оконного кадра текущей строки, но при этом не заменяет все эти строки одной строкой.

Подробно технология оконных функций описана в разделах документации 3.5 «Оконные функции», 4.2.8 «Вызовы оконных функций», 7.2.5 «Обработка оконных функций», а также в справке по команде SELECT (подраздел «Предложение WINDOW»).

3.4.1. Использование агрегатных функций в качестве оконных

Хотя в PostgreSQL есть целый ряд специальных оконных функций, представленных в разделе документации 9.22 «Оконные функции», но обычные агрегатные функции также могут использоваться в качестве оконных.

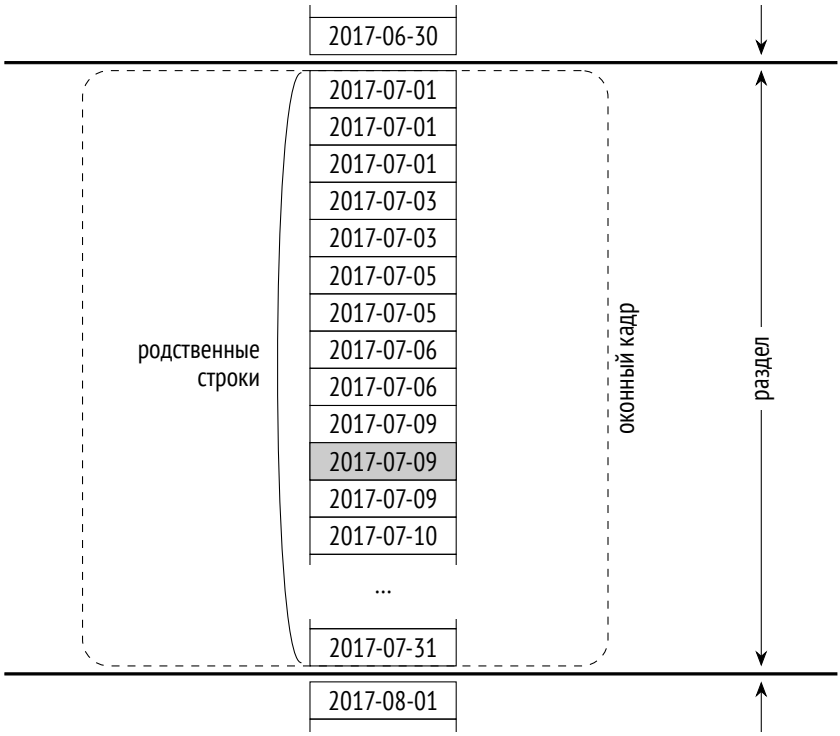


Рис. 3.3. *Оконный кадр по умолчанию без предложения ORDER BY*

Вызов оконной функции (как специальной, так и агрегатной, как в этом примере) требует конструкции OVER и может выглядеть так:

```
count( * ) OVER ( PARTITION BY flight_no ORDER BY scheduled_departure )
```

Предположим, в нашей авиакомпании проводится оптимизация расписания. Руководство обратило внимание на маршрут Москва — Санкт-Петербург, по которому следуют несколько рейсов. Цель исследования: выяснить, какая доля пассажиров, перевезенных за день, приходится на каждый из этих рейсов. Для получения информативной картины необходимо вывести результаты по каждой дате отдельно, а не в виде единого показателя за весь отчетный период.

В запрос включим все сведения, представленные в базе данных «Авиаперевозки» за период с 16 июля по 15 августа 2017 г. Кроме того, будем учитывать только выполненные рейсы (со статусом Arrived).

```
\timing on
Секундомер включён.
WITH passengers AS
( SELECT f.scheduled_departure, f.flight_no, count( * ) AS pass_count
  FROM flights f
    JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  WHERE f.departure_airport IN ( 'DME', 'SVO', 'VKO' )
    AND f.arrival_airport = 'LED'
    AND f.status = 'Arrived'
  GROUP BY f.flight_no, f.scheduled_departure
)
SELECT
  scheduled_departure,
  flight_no,
  pass_count,
  sum( pass_count )
    OVER ( PARTITION BY date_trunc( 'day', scheduled_departure ) )
    AS day_pass_count,
  round( pass_count / sum( pass_count )
    OVER ( PARTITION BY date_trunc( 'day', scheduled_departure ) ), 2 )
    AS fract
FROM passengers
ORDER BY scheduled_departure, flight_no;
  scheduled_departure   | flight_no | pass_count | day_pass_count | fract
------------------------+-----------+------------+----------------+-------
 2017-07-16 13:35:00+07 | PG0405    |         79 |            441 |  0.18
 2017-07-16 13:45:00+07 | PG0227    |         93 |            441 |  0.21
 2017-07-16 14:20:00+07 | PG0470    |        169 |            441 |  0.38
 2017-07-16 16:35:00+07 | PG0469    |         34 |            441 |  0.08
 2017-07-16 22:30:00+07 | PG0472    |         64 |            441 |  0.15
 2017-07-16 23:05:00+07 | PG0404    |          2 |            441 |  0.00
 2017-07-17 13:35:00+07 | PG0405    |         85 |            468 |  0.18
 2017-07-17 13:45:00+07 | PG0227    |        112 |            468 |  0.24
 2017-07-17 14:20:00+07 | PG0470    |        169 |            468 |  0.36
 2017-07-17 16:35:00+07 | PG0469    |         35 |            468 |  0.07
 2017-07-17 22:30:00+07 | PG0472    |         65 |            468 |  0.14
 2017-07-17 23:05:00+07 | PG0404    |          2 |            468 |  0.00
 2017-07-18 13:35:00+07 | PG0405    |         83 |            468 |  0.18
 ...
 2017-08-15 13:45:00+07 | PG0227    |        169 |            536 |  0.32
 2017-08-15 14:20:00+07 | PG0470    |        169 |            536 |  0.32
 2017-08-15 15:25:00+07 | PG0228    |         61 |            536 |  0.11
 2017-08-15 16:35:00+07 | PG0469    |         40 |            536 |  0.07
(212 строк)
Время: 118,759 мс
```

Списки кодов аэропортов в запросе представлены в виде литералов. Это сделано с целью упрощения запроса. Более общим решением было бы такое:

```
WHERE f.departure_airport =
        ANY ( SELECT airport_code FROM airports WHERE city = 'Москва' )
  AND f.arrival_airport =
        ANY ( SELECT airport_code FROM airports WHERE city = 'Санкт-Петербург' )
```

Однако при использовании этих конструкций выполнение запроса значительно замедляется, поскольку планировщик неверно определяет количество аэропортов и выбирает метод вложенного цикла для выполнения соединения наборов строк. Оптимизация этого запроса представляет собой отдельную интересную задачу для самостоятельного решения. В качестве одной из идей можно прибегнуть к материализации требуемых рейсов в общем табличном выражении (предложение MATERIALIZED).

В общем табличном выражении passengers выполняется подготовительная работа по подсчету числа пассажиров, перевезенных на каждом из выполненных рейсов за каждый день. В предложении GROUP BY здесь фигурируют столбцы уникального ключа таблицы «Рейсы» (flights), они же присутствуют и в списке SELECT. Мы могли бы вместо них поставить в предложение GROUP BY столбец flight_id, оставив список SELECT неизменным. Это решение тоже было бы корректным, поскольку данный столбец является первичным ключом, остальные столбцы функционально зависят от него. О таких ситуациях говорится в подразделе «Предложение GROUP BY» описания команды SELECT, приведенного в документации.

Нам нужно, сохранив в отчете информацию о числе пассажиров, перевезенных на каждом из рейсов в конкретный день, вычислить долю каждого из этих рейсов в общем числе пассажиров, перевезенных за этот день. Таким образом, требуется сохранить все индивидуальные строки, полученные в общем табличном выражении, и при этом вычислить итоговые суммы и доли по группам этих строк. Эта задача решается с помощью оконных функций.

Разделы в выборке формируются по датам выполненных рейсов (предложение PARTITION BY). Поскольку в конструкции OVER нет предложения ORDER BY, то в качестве оконного кадра каждой строки будет выступать весь раздел, в котором эта строка находится. Следовательно, значение оконной функции для всех строк раздела будет одним и тем же. Хотя вызов функции sum присутствует

в списке SELECT, но вызывается она не для каждой строки, а один раз для каждого раздела; затем вычисленное значение используется в списке SELECT для всех строк раздела, поэтому значения в столбце day_pass_count повторяются.

Обратите внимание, что одна и та же конструкция OVER с оконной (агрегатной) функцией sum фигурирует в запросе дважды. При этом во втором случае (для вычисления доли каждого рейса в дневном количестве перевезенных пассажиров) нельзя написать просто:

```
round( pass_count / day_pass_count, 2 ) AS fract
```

Дело в том, что в списке SELECT нельзя сослаться на псевдоним столбца, вычисляемого в этом же списке SELECT. Поэтому, в частности, не удастся воспользоваться результатами других вычислений, произведенных здесь же. В таком случае можно либо повторить вычисления на основе исходных значений, доступных на этой стадии выполнения команды SELECT (что и сделано в нашем запросе), либо выполнить однократное вычисление оконной функции в подзапросе, а затем результат ее вычисления использовать в главном запросе. Например:

```
WITH passengers AS
...
passengers_2 AS
( SELECT
    scheduled_departure,
    flight_no,
    pass_count,
    sum( pass_count ) OVER ( PARTITION BY date_trunc( 'day', scheduled_departure ) )
      AS day_pass_count -- это значение теперь вычисляется однократно
  FROM passengers
)
SELECT
  scheduled_departure,
  flight_no,
  pass_count,
  day_pass_count,
  round( pass_count / day_pass_count, 2 ) AS fract
FROM passengers_2
ORDER BY scheduled_departure, flight_no;

 ...
Время: 115,138 мс
```

Видно, что быстродействие обоих вариантов запроса примерно одинаковое.

Вернемся к исходному варианту запроса. Вызов оконной функции sum фигурирует в списке SELECT и в плане запроса дважды, но при этом она получает один и тот же аргумент, поэтому двукратное вычисление функции не требуется. Важно, что формирование раздела выполняется только один раз, поскольку определения обоих разделов в команде SELECT совершенно одинаковые. Команда EXPLAIN с параметром VERBOSE покажет, что узел WindowAgg в полученном плане только один. Приведем конкретный фрагмент этого плана:

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 ...
   -> WindowAgg
        Output: passengers.scheduled_departure,
        passengers.flight_no,
        sum(passengers.pass_count) OVER (?),
        round(((passengers.pass_count)::numeric / sum(passengers.pass_count) OVER (?)), 2),
        (date_trunc('day'::text, passengers.scheduled_departure))
...
```

Обратите внимание на первый аргумент функции round. Значение операнда passengers.pass_count автоматически приведено к типу numeric, поскольку второй операнд, функция sum, уже имеет такой тип. О приведении типов в подобных ситуациях можно прочитать в разделе документации 10.2 «Операторы».

Рассмотренный запрос можно модифицировать, учитывая, что вызов оконной функции выполняется дважды и при этом определение раздела в предложении OVER одно и то же. Для упрощения запросов в таких ситуациях служит предложение WINDOW, в котором и определяются параметры окна (правило формирования раздела, сортировка и др.). Здесь же окну присваивается имя, на которое можно затем ссылаться в запросе.

```
WITH passengers AS
...
SELECT
  scheduled_departure,
  flight_no,
  pass_count,
  sum( pass_count ) OVER day_flights_win AS day_pass_count,
  round( pass_count / sum( pass_count ) OVER day_flights_win, 2 ) AS fract
FROM passengers
WINDOW day_flights_win AS ( PARTITION BY date_trunc( 'day', scheduled_departure ) )
ORDER BY scheduled_departure, flight_no;
```

Выше мы получили картину распределения числа пассажиров между рейсами каждого дня. Видны рейсы-лидеры и рейсы-аутсайдеры по объему перевозок. Однако для принятия обоснованного решения о сокращении числа рейсов на маршруте Москва — Санкт-Петербург необходимо проанализировать динамику перевозки пассажиров по каждому рейсу за каждую неделю отчетного периода. Если известно, что рейс считается рентабельным при перевозке за неделю не менее определенного числа пассажиров, такая информация позволит увидеть, начиная с какого дня недели рейс начинает приносить прибыль. Также необходимо попытаться увидеть другие закономерности. Ведь возможно, например, что какой-то вечерний рейс более популярен у пассажиров, вылетающих в пятницу, а какой-то утренний — у пассажиров, вылетающих в субботу.

Давайте реализуем эти рассуждения в виде конкретного запроса. В нем общее табличное выражение такое же, как и в предыдущем запросе, поскольку на первой стадии процесса мы формируем те же самые исходные данные, что и прежде. Однако в главном запросе мы компонуем эти данные по-другому.

```
WITH passengers AS
...
SELECT
  flight_no,
  scheduled_departure,
  extract( week FROM scheduled_departure ) AS week,
  extract( isodow FROM scheduled_departure ) AS dow,
  pass_count,
  sum( pass_count ) OVER flight_week_win AS running_count
FROM passengers
WINDOW flight_week_win AS
( PARTITION BY flight_no, extract( week FROM scheduled_departure )
  ORDER BY extract( isodow FROM scheduled_departure )
)
ORDER BY flight_no, week, dow;
```

В этом запросе мы воспользовались предложением `WINDOW`, хотя вызов оконной функции всего один. Наверное, среди читателей найдутся те, кто согласится, что запрос в таком виде читать легче, чем если бы всю конструкцию для формирования окна мы поместили в список `SELECT`.

Здесь используется функция `extract`, позволяющая получить из значения временной отметки два результата: номер недели календарного года и номер дня недели в формате «1 — понедельник, 7 — воскресенье».

Заметим, что в предложении ORDER BY на уровне всего запроса используются в том числе и псевдонимы вычисляемых столбцов из списка SELECT.

Номер рейса в совокупности с номером недели определяют текущий раздел. Поскольку строки в разделе обрабатываются в порядке номеров дней недели, то оконная (агрегатная) функция sum подсчитывает накопленное число пассажиров, перевезенных на конкретном рейсе *с начала* отчетной недели.

Обратите внимание, что число строк, формируемых в обоих запросах, одинаковое. Однако в разделы эти строки собраны по-разному. В первой выборке разделы для работы оконной функции формируются по календарным датам, а в каждом разделе собраны все выполненные за этот день рейсы. Во второй выборке разделы формируются по рейсу и неделе, а в каждом разделе представлены все дни конкретной недели.

И еще одно замечание: порядок сортировки в предложении ORDER BY в конструкции OVER не обязан совпадать с порядком сортировки на уровне всего запроса, что мы и видим в нашем примере. Даже если эти порядки совпадают, подраздел документации 7.2.5 «Обработка оконных функций» рекомендует явно задавать порядок сортировки на верхнем уровне запроса, поскольку предложения ORDER BY и PARTITION BY конструкции OVER обеспечивают правильный порядок сортировки в процессе работы оконной функции, но не гарантируют, что он сохранится при выводе результирующей выборки.

Настало время выполнить запрос и показать полученную выборку.

```
 flight_no |  scheduled_departure   | week | dow | pass_count | running_count
-----------+-----------------------+------+-----+------------+---------------
 PG0227    | 2017-07-16 13:45:00+07 |   28 |   7 |         93 |            93
 PG0227    | 2017-07-17 13:45:00+07 |   29 |   1 |        112 |           112
 PG0227    | 2017-07-18 13:45:00+07 |   29 |   2 |        109 |           221
 PG0227    | 2017-07-19 13:45:00+07 |   29 |   3 |        112 |           333
 PG0227    | 2017-07-20 13:45:00+07 |   29 |   4 |        118 |           451
 PG0227    | 2017-07-21 13:45:00+07 |   29 |   5 |        106 |           557
 PG0227    | 2017-07-22 13:45:00+07 |   29 |   6 |        123 |           680
 PG0227    | 2017-07-23 13:45:00+07 |   29 |   7 |        133 |           813
 PG0227    | 2017-07-24 13:45:00+07 |   30 |   1 |        131 |           131
 PG0227    | 2017-07-25 13:45:00+07 |   30 |   2 |        146 |           277
 PG0227    | 2017-07-26 13:45:00+07 |   30 |   3 |        147 |           424
 PG0227    | 2017-07-27 13:45:00+07 |   30 |   4 |        158 |           582
 PG0227    | 2017-07-28 13:45:00+07 |   30 |   5 |        168 |           750
 PG0227    | 2017-07-29 13:45:00+07 |   30 |   6 |        164 |           914
 PG0227    | 2017-07-30 13:45:00+07 |   30 |   7 |        170 |          1084
 ...
```

```
 PG0472    | 2017-07-16 22:30:00+07 |   28 |   7 |         64 |           64
 PG0472    | 2017-07-17 22:30:00+07 |   29 |   1 |         65 |           65
 PG0472    | 2017-07-18 22:30:00+07 |   29 |   2 |         66 |          131
 PG0472    | 2017-07-19 22:30:00+07 |   29 |   3 |         67 |          198
 PG0472    | 2017-07-20 22:30:00+07 |   29 |   4 |         68 |          266
 PG0472    | 2017-07-21 22:30:00+07 |   29 |   5 |         74 |          340
 PG0472    | 2017-07-22 22:30:00+07 |   29 |   6 |         77 |          417
 PG0472    | 2017-07-23 22:30:00+07 |   29 |   7 |         90 |          507
 ...
 PG0472    | 2017-08-07 22:30:00+07 |   32 |   1 |        130 |          130
 PG0472    | 2017-08-08 22:30:00+07 |   32 |   2 |        108 |          238
 PG0472    | 2017-08-09 22:30:00+07 |   32 |   3 |        122 |          360
 PG0472    | 2017-08-10 22:30:00+07 |   32 |   4 |        126 |          486
 PG0472    | 2017-08-11 22:30:00+07 |   32 |   5 |        133 |          619
 PG0472    | 2017-08-12 22:30:00+07 |   32 |   6 |        119 |          738
 PG0472    | 2017-08-13 22:30:00+07 |   32 |   7 |        117 |          855
 PG0472    | 2017-08-14 22:30:00+07 |   33 |   1 |        121 |          121
(212 строк)
```

Предположим, что рейс считается рентабельным, если на нем перевозятся не менее 500 пассажиров в неделю. Исходя из этого критерия можно сказать, что рейс PG0227 на 29-й неделе календарного 2017 года (с 17 по 23 июля) приносил прибыль, начиная с пятницы 21 июля. Рейс PG0472 на этой же неделе стал прибыльным только в воскресенье 23 июля, а на 32-й неделе — начиная с пятницы 11 августа.

3.4.2. Способы формирования оконного кадра

Специфика конкретной задачи определяет множество строк раздела, которые необходимо учитывать при вычислении оконной функции для каждой строки выборки. Например, может потребоваться принимать во внимание все строки раздела, начиная с первой и заканчивая текущей строкой, или учитывать только строку раздела, отстоящую от текущей на заданное число строк. Множество строк, рассматриваемых оконной функцией, задается оконным кадром с помощью специальных синтаксических средств.

В качестве примера рассмотрим фрагмент сложной задачи анализа финансового положения нашей авиакомпании. Мы ограничимся лишь изучением процесса поступления денег за счет продажи авиабилетов. Будем исходить из того,

что равномерное ежедневное поступление денежных средств является хорошим показателем стабильной работы авиакомпании.

Для начала мы рассмотрим так называемое *скользящее среднее* значение. Этот показатель отличается от обычного среднего значения тем, что вычисляется в каждой точке выборки на основе значений из некоторого интервала от текущей точки в «прошлое». Он позволяет получить сглаженный ряд значений. Для определения равномерности поступления денег от бронирования билетов мы вычисляем разность между дневной суммой бронирований и скользящим средним значением, вычисленным для этого же дня. Для наглядности представим ее еще и в процентах. Чем меньше эта разность, тем более равномерным является поступление.

Теперь перейдем непосредственно к запросу.

```
WITH day_amounts ( b_date, day_sum ) AS
( SELECT date_trunc( 'day', book_date ),
         round( sum( total_amount ) / 1000000, 2 )
  FROM bookings
  GROUP BY 1
)
SELECT
  to_char( b_date, 'YYYY-MM-DD' ) AS date,
  day_sum,
  round( avg( day_sum ) OVER moving_win, 2 ) AS mv_avg_5,
  round( day_sum - avg( day_sum ) OVER moving_win, 2 ) AS delta,
  round( ( day_sum - avg( day_sum ) OVER moving_win ) / day_sum * 100, 2 ) AS percent
FROM day_amounts
WINDOW moving_win AS
( ORDER BY b_date ROWS BETWEEN 4 PRECEDING AND CURRENT ROW
)
ORDER BY date;
```

В конструкции WITH мы выполняем подготовительную работу: вычисляем суммы бронирований, выполненных за каждую дату. При этом суммы будем представлять в миллионах рублей, чтобы сделать отчет более наглядным.

Обратимся к определению окна в предложении WINDOW. Здесь отсутствует предложение PARTITION BY, поэтому разделом будет служить вся выборка. Интервал сглаживания для вычисления скользящего среднего примем равным, скажем, пяти дням. Конечно, он может быть и другим, если его выбор будет каким-то

образом обоснован. Поскольку этот интервал отсчитывается от текущего момента времени назад, в прошлое, то в качестве оконного кадра текущей строки должны выступать сама эта строка и еще четыре строки, предшествующие ей в порядке сортировки, указанном в предложении ORDER BY. Выполнение этого требования обеспечивает конструкция ROWS BETWEEN 4 PRECEDING AND CURRENT ROW. Она определяет множество строк, составляющих оконный кадр, путем указания его начала (4 PRECEDING) и конца (CURRENT ROW). В выражении 4 PRECEDING целое число 4 означает смещение первой строки оконного кадра от текущей строки к началу раздела.

В рассматриваемой конструкции требует пояснения ключевое слово ROWS. Оно задает так называемый *режим формирования кадра*. В этом режиме выражение CURRENT ROW означает только текущую строку, а родственные (peer) строки не учитываются. Аналогично выражение 4 PRECEDING означает ровно четыре строки, предшествующие текущей строке, также без учета родственных строк. Но в нашем случае для каждой даты формируется ровно одна строка, поэтому родственных строк вообще не существует.

Этот режим имеет особенность: если предложение ORDER BY сортирует строки не уникальным образом, то возможно получение непредсказуемых результатов (см. подраздел «Предложение WINDOW» в описании команды SELECT, приведенном в документации).

Настало время выполнить запрос. Вот что он выдает:

```
    date    | day_sum | mv_avg_5 | delta  | percent
------------+---------+----------+--------+---------
 2017-06-21 |    0.44 |     0.44 |   0.00 |    0.00
 2017-06-22 |    0.56 |     0.50 |   0.06 |   10.71
 2017-06-23 |    1.82 |     0.94 |   0.88 |   48.35
 2017-06-24 |    5.47 |     2.07 |   3.40 |   62.11
 2017-06-25 |   13.69 |     4.40 |   9.29 |   67.89
 2017-06-26 |   26.46 |     9.60 |  16.86 |   63.72
 2017-06-27 |   49.48 |    19.38 |  30.10 |   60.82
 2017-06-28 |   83.89 |    35.80 |  48.09 |   57.33
 2017-06-29 |  145.31 |    63.77 |  81.54 |   56.12
 2017-06-30 |  196.23 |   100.27 |  95.96 |   48.90
 2017-07-01 |  267.22 |   148.43 | 118.79 |   44.46
 2017-07-02 |  321.96 |   202.92 | 119.04 |   36.97
 2017-07-03 |  372.18 |   260.58 | 111.60 |   29.99
 ...
```

```
 2017-08-01 |  444.54 |   437.64 |   6.90 |    1.55
 2017-08-02 |  446.35 |   437.54 |   8.81 |    1.97
 2017-08-03 |  430.93 |   438.67 |  -7.74 |   -1.80
 2017-08-04 |  451.32 |   443.46 |   7.86 |    1.74
 2017-08-05 |  452.65 |   445.16 |   7.49 |    1.66
 2017-08-06 |  453.36 |   446.92 |   6.44 |    1.42
 2017-08-07 |  447.97 |   447.25 |   0.72 |    0.16
 2017-08-08 |  471.32 |   455.32 |  16.00 |    3.39
 2017-08-09 |  471.70 |   459.40 |  12.30 |    2.61
 2017-08-10 |  487.51 |   466.37 |  21.14 |    4.34
 2017-08-11 |  497.21 |   475.14 |  22.07 |    4.44
 2017-08-12 |  480.76 |   481.70 |  -0.94 |   -0.20
 2017-08-13 |  502.46 |   487.93 |  14.53 |    2.89
 2017-08-14 |  528.43 |   499.27 |  29.16 |    5.52
 2017-08-15 |  483.56 |   498.48 | -14.92 |   -3.09
(56 строк)
```

Как видим, в начале отчетного периода отклонения текущей дневной выручки от значения скользящего среднего были велики, а затем они перестали превышать 5–6 %.

При интерпретации результатов нужно учитывать, что границы оконного кадра не могут выходить за пределы раздела. Поэтому при вычислении скользящего среднего для каждой из первых четырех строк выборки оконный кадр фактически будет состоять не из пяти строк, а из меньшего их количества. Так, в составе оконного кадра для первой строки окажется только сама эта строка, для второй — первая и вторая строки и т. д. Поэтому для первой строки значения в столбцах `delta` и `percent` равны нулю.

Режим `ROWS` проиллюстрирован на рис. 3.4a и 3.4b. На первом из них показана ситуация, когда оконный кадр начинается на строке, смещенной от текущей к началу раздела на заданное число строк, а завершается на текущей строке (`ROWS BETWEEN 4 PRECEDING AND CURRENT ROW`). На втором рисунке оконный кадр начинается на текущей строке, а завершается на строке, смещенной от текущей к концу раздела на заданное число строк (`ROWS BETWEEN CURRENT ROW AND 4 FOLLOWING`).

Существует еще два режима формирования оконного кадра: `RANGE` и `GROUPS`. В них так же, как и в режиме `ROWS`, оконный кадр определяется своим началом и концом. При использовании режима `RANGE` сортировка строк раздела должна выполняться только по одному столбцу. Смещение задается максимальной разностью значений этого столбца в текущей строке и в той строке, на которой

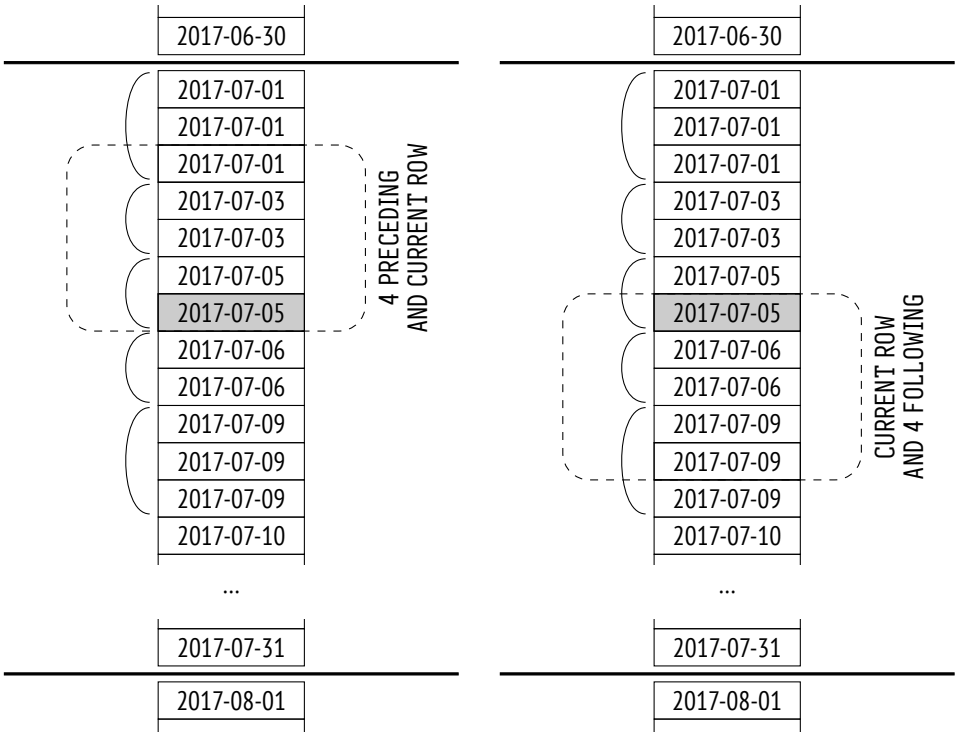


Рис. 3.4. *Оконный кадр в режиме ROWS*

должен начинаться или заканчиваться оконный кадр. Если в выборке есть родственные строки, то при поиске начала оконного кадра используется первая строка из группы этих строк, а при поиске его конца — последняя из них.

Режим `RANGE` проиллюстрирован на рис. 3.5a и 3.5b. На первом из них показана ситуация, когда оконный кадр начинается на первой строке из группы родственных строк, смещенных к началу раздела от текущей строки на заданный интервал, а завершается на последней строке, родственной текущей (`RANGE BETWEEN interval '4 days' PRECEDING AND CURRENT ROW`). Обратите внимание на пропуски в последовательности дат. Заметьте также, что вычитание интервала, равного четырем дням, из даты `2017-07-06` дает `2017-07-02`. Однако такой даты в выборке нет. В таком случае началом кадра будет наименьшая дата из выборки, превышающая эту расчетную дату, то есть `2017-07-03`.

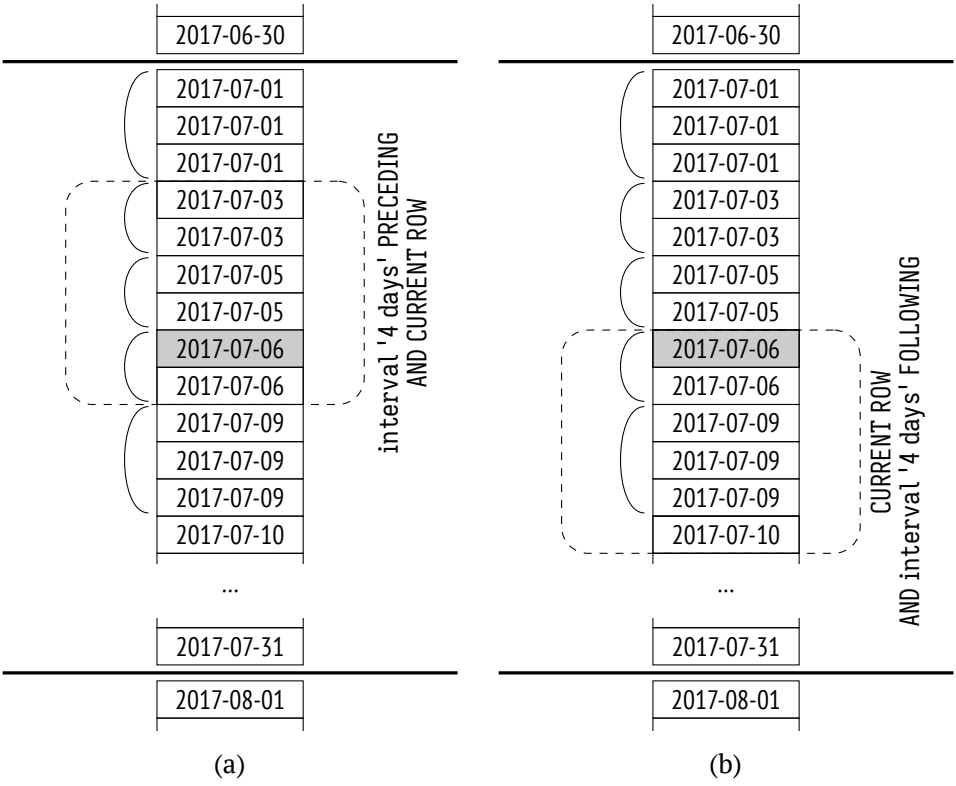


Рис. 3.5. *Оконный кадр в режиме RANGE*

На втором рисунке оконный кадр начинается на первой строке, родственной текущей. В данном случае такой строкой оказывается сама текущая строка. Завершается кадр на последней из родственных строк, смещенных от текущей к концу раздела на заданный интервал (`RANGE BETWEEN CURRENT ROW AND interval '4 days' FOLLOWING`). В данном случае у строки, на которую приходится конец кадра, нет других родственных строк.

Наш запрос можно записать иначе с учетом того, что размер оконного кадра должен быть равен четырем дням, а родственные строки в нашем случае отсутствуют. Изменится только определение окна:

```
WINDOW moving_win AS
( ORDER BY b_date RANGE BETWEEN interval '4 days' PRECEDING AND CURRENT ROW
)
```

При использовании режима GROUPS начало и конец кадра задаются в терминах групп родственных строк. Например, в выражении 4 PRECEDING целое число 4 означало бы, что первая группа оконного кадра расположена в выборке со смещением к началу раздела на 4 группы от текущей группы (к которой относится текущая строка). Но поскольку группы состоят из строк, когда говорят о *первой группе* оконного кадра, то имеют в виду *первую строку* из этой группы, а когда говорят о *последней группе* оконного кадра, то имеют в виду *последнюю строку* из этой группы. Предлагаем читателю самостоятельно смоделировать ситуацию в предметной области «Авиаперевозки», в которой было бы логично использовать режим GROUPS.

Режим GROUPS проиллюстрирован на рис. 3.6a и 3.6b. На первом из них показана ситуация, когда оконный кадр начинается на первой строке из группы родственных строк, смещенных к началу раздела от текущей строки на заданное число групп, а завершается на последней строке из группы строк, родственных текущей (GROUPS BETWEEN 2 PRECEDING AND CURRENT ROW). В данном случае текущая строка и является последней из родственных строк. Обратите внимание и на пропуски в последовательности дат.

На втором рисунке кадр начинается на первой строке из группы строк, родственных текущей, а завершается на последней строке из группы родственных строк, смещенных от текущей строки к концу раздела на заданное число групп (GROUPS BETWEEN CURRENT ROW AND 2 FOLLOWING).

Если режим формирования оконного кадра в конструкции OVER не задан, подразумевается RANGE UNBOUNDED PRECEDING, что равносильно более длинной записи RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. Если при этом задано предложение ORDER BY, то оконный кадр начинается на первой строке раздела, а заканчивается на последней строке, родственной текущей. Если же предложение ORDER BY не задано, то оконным кадром будет весь раздел, поскольку все его строки станут родственными текущей строке.

Подробно режимы формирования оконного кадра рассматриваются в подразделе документации 4.2.8 «Вызовы оконных функций».

При задании оконного кадра можно исключать текущую строку или даже ряд строк из обработки. Покажем это на примере, аналогичном предыдущему, но на этот раз будем вычислять средние дневные суммы бронирований за шесть дней недели, исключая текущую дату.

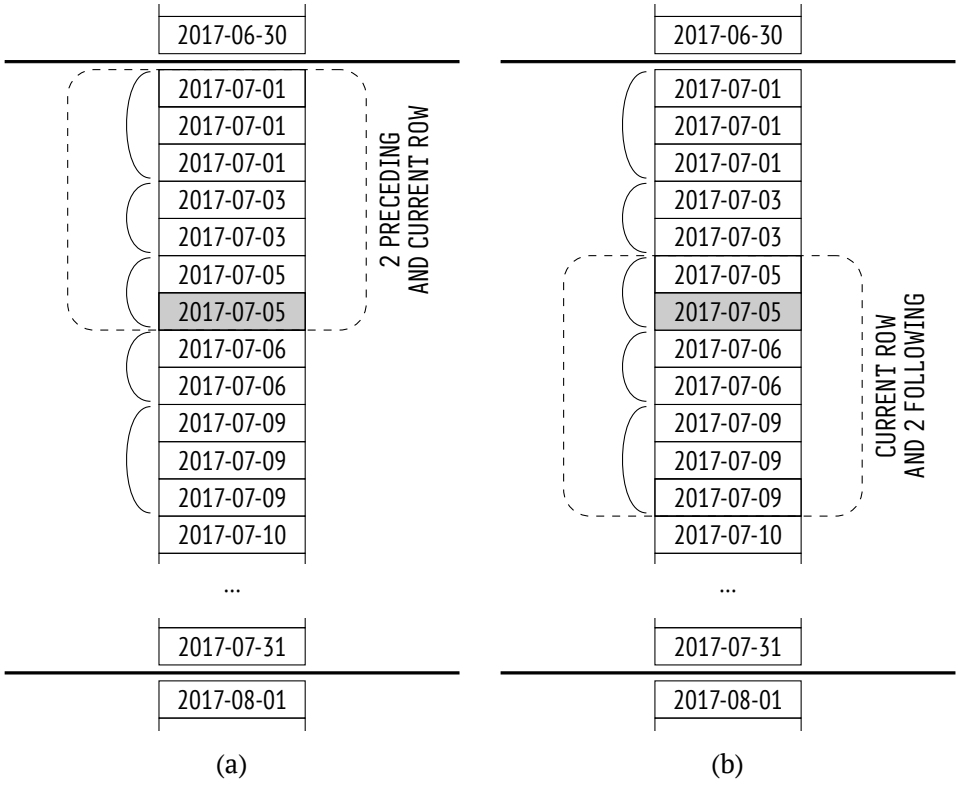


Рис. 3.6. *Оконный кадр в режиме GROUPS*

Запрос, решающий поставленную задачу, походит на предыдущий, поэтому ограничимся лишь кратким пояснением.

В общем табличном выражении выполняется такая же подготовительная работа. В главном запросе воспользуемся функцией `extract` для получения номера недели и номера дня недели в формате «1 — понедельник, 7 — воскресенье». Разделы сформируем для каждой недели. Конструкция `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING EXCLUDE CURRENT ROW` говорит о том, что в состав оконного кадра входят все строки раздела, за исключением текущей строки.

Как и в предыдущем запросе, для оценки величины отклонения текущего значения дневной выручки от ее среднего значения будем использовать не только разность этих показателей, но и их соотношение в процентах.

```
WITH day_amounts ( b_date, day_sum ) AS
( SELECT date_trunc( 'day', book_date ),
         round( sum( total_amount ) / 1000000, 2 )
  FROM bookings
  GROUP BY 1
)
SELECT
  to_char( b_date, 'YYYY-MM-DD' ) AS date,
  extract( week FROM b_date ) AS week,
  extract( isodow FROM b_date ) AS dow,
  day_sum,
  round( avg( day_sum ) OVER moving_win, 2 ) AS avg_6_days,
  round( day_sum - avg( day_sum ) OVER moving_win, 2 ) AS delta,
  round( ( day_sum - avg( day_sum ) OVER moving_win ) / day_sum * 100, 2 ) AS percent
FROM day_amounts
WINDOW moving_win AS
( PARTITION BY extract( week FROM b_date )
  ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
  EXCLUDE CURRENT ROW
)
ORDER BY date;
```

Обратите внимание, что в определении окна мы повторили выражение, которое использовали для вычисления значений выходного столбца с именем week. При этом нельзя написать PARTITION BY week, поскольку в предложениях PARTITION BY и ORDER BY, присутствующих в определении окна, не допускается использовать имена или номера выходных столбцов.

Теперь можно выполнить запрос. Получаем такой результат:

```
    date    | week | dow | day_sum | avg_6_days |  delta  | percent
------------+------+-----+---------+------------+---------+----------
 2017-06-21 |   25 |   3 |    0.44 |       5.39 |   -4.95 | -1123.86
 2017-06-22 |   25 |   4 |    0.56 |       5.36 |   -4.80 |  -856.25
 2017-06-23 |   25 |   5 |    1.82 |       5.04 |   -3.22 |  -176.92
 2017-06-24 |   25 |   6 |    5.47 |       4.13 |    1.34 |    24.54
 2017-06-25 |   25 |   7 |   13.69 |       2.07 |   11.62 |    84.86
 2017-06-26 |   26 |   1 |   26.46 |     177.35 | -150.89 |  -570.25
 2017-06-27 |   26 |   2 |   49.48 |     173.51 | -124.03 |  -250.67
 ...
 2017-08-12 |   32 |   6 |  480.76 |     479.70 |    1.07 |     0.22
 2017-08-13 |   32 |   7 |  502.46 |     476.08 |   26.38 |     5.25
 2017-08-14 |   33 |   1 |  528.43 |     483.56 |   44.87 |     8.49
 2017-08-15 |   33 |   2 |  483.56 |     528.43 |  -44.87 |    -9.28
(56 строк)
```

Из полученной выборки видно, что ближе к концу отчетного периода отклонение текущего значения от среднего не превышает десяти процентов. Будем считать это хорошим финансовым результатом.

В заключение добавим, что предложение EXCLUDE предусматривает еще три варианта действий. Первый из них — EXCLUDE GROUP — позволяет исключить из состава кадра не только текущую строку, но и все родственные ей строки. Второй — EXCLUDE TIES — исключает только строки, родственные текущей, но сама эта строка остается в составе кадра. Вариант EXCLUDE NO OTHERS просто задает явное указание не исключать строки из обработки, что и делается по умолчанию. Предлагаем читателю самостоятельно смоделировать ситуации в сфере авиаперевозок для иллюстрации вариантов предложения EXCLUDE, не рассмотренных в книге.

3.4.3. Совместное использование оконных и агрегатных функций

Из документации известно (см. подраздел 7.2.5 «Обработка оконных функций»), что при использовании в одном запросе как оконных, так и агрегатных функций с предложениями GROUP BY и, возможно, HAVING оконные функции работают с уже *сгруппированными* строками. Покажем это на примере.

Предположим, что требуется оценить нагрузку на электронную систему бронирования авиабилетов в выходные дни (субботу и воскресенье) по сравнению с рабочими днями. В каждой операции бронирования может оформляться несколько билетов, при этом в каждом билете может присутствовать несколько перелетов, каждый из которых требует использования ресурсов системы. Таким образом, в качестве меры загруженности системы бронирования логично использовать число перелетов, оформленных в ней за один день.

Идея запроса такова: для каждой отчетной недели подсчитать среднее число перелетов, оформленных за пять рабочих дней, и сравнить с числом перелетов, оформленных в выходные дни. Поскольку в таблице «Перелеты» (ticket_flights) нет информации о дате бронирования, придется обратиться к таблице «Бронирования» (bookings). Связать эти две таблицы можно через таблицу «Билеты» (tickets).

```
SELECT
  to_char( date_trunc( 'day', b.book_date ), 'YYYY-MM-DD' ) AS b_date,
  -- b.book_date::date AS b_date,  -- можно сделать и так
  extract( week FROM b.book_date ) AS week,
  extract( isodow FROM b.book_date ) AS dow,
  count( * ) AS day_tf_count,
  round(
    avg( count( * ) )
      FILTER ( WHERE extract( isodow FROM book_date ) BETWEEN 1 AND 5 )
      OVER week_win,
    0
  ) AS avg_5_days,
  round(
    avg( count( * ) )
      FILTER ( WHERE extract( isodow FROM book_date ) IN ( 6, 7 ) )
      OVER week_win,
    0
  ) AS avg_67 -- выходные дни
FROM bookings b
  JOIN tickets t ON t.book_ref = b.book_ref
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
GROUP BY b_date, week, dow
WINDOW week_win AS ( PARTITION BY extract( week FROM b.book_date ) )
ORDER BY b_date;
```

Сгруппируем эти данные по датам, получим номера дней недели с помощью функции extract и вычислим общее количество перелетов, оформленных в течение каждого дня (это столбец day_tf_count).

После группировки в действие вступает функция avg. Это агрегатная функция, которая используется здесь в роли оконной. Поскольку в этом запросе она работает с уже сгруппированными строками, то в качестве ее аргумента выступает вызов функции count (и вместо него нельзя написать имя выходного столбца day_tf_count). При этом вызовы оконной функции дополняются предложением FILTER, которое позволяет включить в обработку только те сгруппированные строки, в которых значения столбца book_date соответствуют рабочим дням (столбец avg_5_days) или выходным (столбец avg_67).

Обратите внимание, что в условии WHERE предложения FILTER используется не имя выходного столбца dow, а выражение, которое было использовано для его вычисления ранее. При этом предложение FILTER относится к сгруппированным строкам, однако условие WHERE опирается на исходные данные из индивидуальных строк.

В подразделе документации 4.2.8 «Вызовы оконных функций» сказано, что предложение FILTER могут принимать только агрегирующие оконные функции, как, например, avg в нашем запросе. А вот оконные функции общего назначения, рассмотренные в подразделе 3.4.4 «Оконные функции общего назначения» (с. 176), этим свойством не обладают.

Полученные средние значения округляем до целого числа, поскольку речь идет о числе перелетов.

В определении окна в конструкции WINDOW отсутствует предложение ORDER BY. В таком случае, как сказано в подразделе документации 4.2.8 «Вызовы оконных функций», в качестве оконного кадра выступают все строки раздела, поскольку все они становятся родственными текущей строке. Упорядочение здесь не требуется, так как достаточно условия отбора в предложении FILTER. Такой — неявный — способ задания раздела и оконного кадра эквивалентен следующему — явному: ROWS (или RANGE) BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING. А вот задать режим GROUPS без предложения ORDER BY не получится. В общем случае, независимо от режима (RANGE, ROWS или GROUPS), смещение UNBOUNDED PRECEDING означает, что оконный кадр начинается на первой строке раздела, а смещение UNBOUNDED FOLLOWING — что он завершается на последней строке раздела.

Возвращаясь к нашему запросу, получаем такой результат:

```
   b_date   | week | dow | day_tf_count | avg_5_days | avg_67
------------+------+-----+--------------+------------+--------
 2017-06-21 |   25 |   3 |           20 |         41 |    494
 2017-06-22 |   25 |   4 |           22 |         41 |    494
 2017-06-23 |   25 |   5 |           80 |         41 |    494
 2017-06-24 |   25 |   6 |          284 |         41 |    494
 2017-06-25 |   25 |   7 |          704 |         41 |    494
 2017-06-26 |   26 |   1 |         1323 |       5187 |  14684
 2017-06-27 |   26 |   2 |         2639 |       5187 |  14684
 2017-06-28 |   26 |   3 |         4295 |       5187 |  14684
 2017-06-29 |   26 |   4 |         7408 |       5187 |  14684
 ...
 2017-08-10 |   32 |   4 |        24656 |      23944 |  25666
 2017-08-11 |   32 |   5 |        25344 |      23944 |  25666
 2017-08-12 |   32 |   6 |        25109 |      23944 |  25666
 2017-08-13 |   32 |   7 |        26222 |      23944 |  25666
 2017-08-14 |   33 |   1 |        27459 |      26499 |
 2017-08-15 |   33 |   2 |        25538 |      26499 |
(56 строк)
```

Как можно интерпретировать полученные результаты?

В начале отчетного периода среднее число перелетов, оформленных в выходные дни, значительно превышало их среднее число, рассчитанное для рабочих дней каждой конкретной недели. Затем первый показатель стал немного меньше второго (этот фрагмент выборки не показан). Однако в конце отчетного периода вновь проявилось превышение первого показателя над вторым. Оно стало уже не таким заметным, как в первые недели, тем не менее оно имеет место. Таким образом, можно выдвинуть гипотезу, что в выходные дни электронная система бронирования авиабилетов загружена больше, чем в рабочие. Каковы причины этого явления и какие решения могли бы в этом случае быть приняты руководством компании, читатель может предположить самостоятельно.

3.4.4. Оконные функции общего назначения

Функции first_value, last_value и nth_value. В разделе документации 9.22 «Оконные функции» представлен целый ряд собственно оконных функций. Давайте начнем их рассмотрение с таких представителей этого семейства, как `first_value`, `last_value` и `nth_value`. Первая из них возвращает значение столбца или выражения, вычисленное для первой строки оконного кадра, вторая функция делает то же самое для последней строки оконного кадра, а третья функция — для его *n*-й строки.

В качестве примера снова обратимся к теме оценки равномерности поступления денег за счет продажи авиабилетов. Для каждой даты будем вычислять разности между общей суммой бронирований, произведенных в этот день, и суммами бронирований, полученными в первый день месяца, в последний день месяца и в день, являющийся его серединой. Будем считать, что если для всех дат текущего месяца три эти разности окажутся небольшими, то это в какой-то степени говорит о равномерном поступлении денег от продажи билетов на счет авиакомпании.

Сначала представим запрос, решающий поставленную задачу, а затем приведем его описание. В псевдонимах столбцов `d_fv`, `d_lv`, `d_nv` буква `d` означает «delta», то есть разность.

```
WITH day_amounts ( b_date, day_sum ) AS
( SELECT date_trunc( 'day', book_date ),
         round( sum( total_amount ) / 1000000, 2 )
  FROM bookings
  GROUP BY 1
),
days_per_month ( month, days_count ) AS
( SELECT extract( 'mon' FROM b_date ),
         count( * )::integer
  FROM day_amounts
  GROUP BY 1
),
day_amounts_2 AS
( SELECT
    to_char( b_date, 'YYYY-MM-DD' ) AS date,
    day_sum,
    first_value( day_sum ) OVER month_win AS fv,
    last_value( day_sum ) OVER month_win AS lv,
    nth_value( day_sum,
               ( SELECT days_count
                 FROM days_per_month
                 WHERE month = extract( 'mon' FROM b_date )
               ) / 2
    ) OVER month_win AS nv
  FROM day_amounts
  WINDOW month_win AS
  ( PARTITION BY extract( 'mon' FROM b_date )
    ORDER BY b_date
    ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
  )
)
SELECT
  date,
  day_sum,
  fv,
  lv,
  nv,
  day_sum - fv AS d_fv,
  day_sum - lv AS d_lv,
  day_sum - nv AS d_nv
FROM day_amounts_2
ORDER BY date;
```

В первом общем табличном выражении вычисляются дневные суммы полных стоимостей бронирований, представленные в миллионах рублей, во втором —

фактическое количество дней в каждом из месяцев, попадающих в выборку. Хотя в нашей авиакомпании операции бронирования происходят каждый день, однако возможны неполные месяцы в начале и в конце временного периода, представленного в конкретном варианте базы данных (small, medium, big). Поэтому нельзя просто считать, что в июне тридцать дней, в июле и августе — тридцать один и т. д., а приходится использовать агрегатную функцию count. При этом ее результат типа bigint приводится к типу integer, поскольку он используется в общем табличном выражении day_amounts_2 для вычисления второго параметра функции nth_value, задающего номер строки оконного кадра, а параметр имеет тип integer.

Значения трех оконных функций вычисляются в общем табличном выражении day_amounts_2, а не в главном запросе, чтобы не повторять в нем эти вычисления дважды. Ведь нельзя написать так:

```
...
SELECT
  to_char( b_date, 'YYYY-MM-DD' ) AS date,
  day_sum,
  first_value( day_sum ) OVER month_win AS fv,
...
  day_sum - fv AS d_fv,
...
```

Здесь пришлось бы повторить вызов оконной функции еще раз:

```
...
SELECT
  to_char( b_date, 'YYYY-MM-DD' ) AS date,
  day_sum,
  first_value( day_sum ) OVER month_win AS fv,
...
  day_sum - first_value( day_sum ) OVER month_win AS d_fv,
...
```

В главном запросе вычисляются разности между значением дневной суммы полных стоимостей бронирований и этими тремя — заранее вычисленными — значениями оконных функций. Тем самым решается поставленная задача. Столбцы fv, lv и nv включены в запрос только для наглядности.

Важно учесть тот факт, что все три оконные функции, используемые в запросе, работают с оконным кадром, и нам нужно, чтобы оконный кадр совпадал со всем разделом. В принципе этого можно добиться, исключив предложение

ORDER BY из определения окна, однако такой вариант не подходит, поскольку для собственно оконных функций важно упорядочение строк в разделе. Ведь, например, в нашем случае для определения первой, средней и последней строк раздела необходимо упорядочить строки по дате операции бронирования. Существует второй, устраивающий нас вариант формирования оконного кадра: нужно в определение окна добавить конструкцию ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

Обратите внимание, что в предложении ORDER BY, находящемся в определении окна, используется имя входного столбца b_date, а имя выходного столбца date здесь использовать нельзя. А вот в предложении ORDER BY на уровне всего запроса использовать имя выходного столбца date можно. Подробно эти особенности освещаются в подразделе документации 4.2.8 «Вызовы оконных функций» и в подразделе «Предложение WINDOW» описания команды SELECT.

Функция to_char в общем табличном выражении day_amounts_2 используется для того, чтобы даты выводились в формате YYYY-MM-DD независимо от того формата, который установлен в текущем сеансе работы с базой данных.

В первом общем табличном выражении можно было бы воспользоваться операцией приведения типа вместо функции date_trunc:

```
WITH day_amounts ( b_date, day_sum ) AS
( SELECT book_date::date,
...
```

Добавим в заключение, что вложенные вызовы оконных функций недопустимы. Поэтому приходится использовать подзапрос для получения значения второго аргумента функции nth_value. Нельзя было бы написать так:

```
...
( SELECT ...
...
    nth_value( day_sum,
               ( count( * ) OVER month_win::integer ) / 2
    ) OVER month_win AS nv,
...
```

Теперь представим результат выполнения запроса. Включим в него все строки, соответствующие датам июня, а из дат июля и августа — только первую, последнюю и ту, что соответствует середине месяца. Для этих дат значение одного из столбцов d_fv, d_lv и d_nv будет равно нулю.

```
    date    | day_sum |   fv   |   lv   |   nv   |  d_fv  |  d_lv   |  d_nv
------------+---------+--------+--------+--------+--------+---------+---------
 2017-06-21 |    0.44 |   0.44 | 196.23 |  13.69 |   0.00 | -195.79 |  -13.25
 2017-06-22 |    0.56 |   0.44 | 196.23 |  13.69 |   0.12 | -195.67 |  -13.13
 2017-06-23 |    1.82 |   0.44 | 196.23 |  13.69 |   1.38 | -194.41 |  -11.87
 2017-06-24 |    5.47 |   0.44 | 196.23 |  13.69 |   5.03 | -190.76 |   -8.22
 2017-06-25 |   13.69 |   0.44 | 196.23 |  13.69 |  13.25 | -182.54 |    0.00
 2017-06-26 |   26.46 |   0.44 | 196.23 |  13.69 |  26.02 | -169.77 |   12.77
 2017-06-27 |   49.48 |   0.44 | 196.23 |  13.69 |  49.04 | -146.75 |   35.79
 2017-06-28 |   83.89 |   0.44 | 196.23 |  13.69 |  83.45 | -112.34 |   70.20
 2017-06-29 |  145.31 |   0.44 | 196.23 |  13.69 | 144.87 |  -50.92 |  131.62
 2017-06-30 |  196.23 |   0.44 | 196.23 |  13.69 | 195.79 |    0.00 |  182.54
 2017-07-01 |  267.22 | 267.22 | 444.18 | 451.08 |   0.00 | -176.96 | -183.86
 ...
 2017-07-15 |  451.08 | 267.22 | 444.18 | 451.08 | 183.86 |    6.90 |    0.00
 ...
 2017-07-31 |  444.18 | 267.22 | 444.18 | 451.08 | 176.96 |    0.00 |   -6.90
 2017-08-01 |  444.54 | 444.54 | 483.56 | 447.97 |   0.00 |  -39.02 |   -3.43
 ...
 2017-08-07 |  447.97 | 444.54 | 483.56 | 447.97 |   3.43 |  -35.59 |    0.00
 ...
 2017-08-15 |  483.56 | 444.54 | 483.56 | 447.97 |  39.02 |    0.00 |   35.59
(56 строк)
```

Глядя на полученный результат, можно сказать, что в течение той части июня, которая представлена в базе данных, считать поступление денег равномерным нельзя. Однако видна тенденция возрастания доходов. Вывод относительно июля и августа предлагаем читателю сделать самостоятельно, выполнив запрос и получив полную выборку.

Функции lead и lag. Для анализа динамики изменений можно сравнивать значение показателя с его значением за предыдущий период, например за тот же месяц прошлого года. Давайте опять обратимся к теме бронирований билетов. Будем сравнивать сумму полных стоимостей бронирований, оформленных в текущий день, с суммами, полученными в тот же день неделю назад.

Для решения задачи воспользуемся оконной функцией lead. Она также представлена в разделе документации 9.22 «Оконные функции». Эта функция возвращает значение для строки, положение которой задается смещением от текущей строки к *концу* раздела (рис. 3.7). При сортировке строк раздела по убыванию дат функция lead выдаст желаемый результат, поскольку возвратит значение суммы, соответствующее более старой (меньшей) дате.

```
WITH day_amounts ( b_date, day_sum ) AS
( SELECT date_trunc( 'day', book_date ),
         round( sum( total_amount ) / 1000000, 2 )
  FROM bookings
  GROUP BY 1
)
SELECT
  to_char( b_date, 'YYYY-MM-DD' ) AS date,
  extract( isodow FROM b_date ) AS dow, -- день недели
  day_sum,
  lead( day_sum, 7 ) OVER all_rows_win AS week_ago,
  day_sum - lead( day_sum, 7 ) OVER all_rows_win AS delta
FROM day_amounts
WINDOW all_rows_win AS ( ORDER BY b_date DESC )
ORDER BY b_date DESC;
```

```
    date    | dow | day_sum | week_ago | delta
------------+-----+---------+----------+--------
 2017-08-15 |   2 |  483.56 |   471.32 |  12.24
 2017-08-14 |   1 |  528.43 |   447.97 |  80.46
 2017-08-13 |   7 |  502.46 |   453.36 |  49.10
 2017-08-12 |   6 |  480.76 |   452.65 |  28.11
 2017-08-11 |   5 |  497.21 |   451.32 |  45.89
 2017-08-10 |   4 |  487.51 |   430.93 |  56.58
 2017-08-09 |   3 |  471.70 |   446.35 |  25.35
 2017-08-08 |   2 |  471.32 |   444.54 |  26.78
 2017-08-07 |   1 |  447.97 |   444.18 |   3.79
 2017-08-06 |   7 |  453.36 |   427.33 |  26.03
 2017-08-05 |   6 |  452.65 |   425.30 |  27.35
 ...
 2017-07-06 |   4 |  437.31 |   145.31 | 292.00
 2017-07-05 |   3 |  422.44 |    83.89 | 338.55
 2017-07-04 |   2 |  408.98 |    49.48 | 359.50
 2017-07-03 |   1 |  372.18 |    26.46 | 345.72
 2017-07-02 |   7 |  321.96 |    13.69 | 308.27
 2017-07-01 |   6 |  267.22 |     5.47 | 261.75
 2017-06-30 |   5 |  196.23 |     1.82 | 194.41
 2017-06-29 |   4 |  145.31 |     0.56 | 144.75
 2017-06-28 |   3 |   83.89 |     0.44 |  83.45
 2017-06-27 |   2 |   49.48 |          |
 2017-06-26 |   1 |   26.46 |          |
 2017-06-25 |   7 |   13.69 |          |
 2017-06-24 |   6 |    5.47 |          |
 2017-06-23 |   5 |    1.82 |          |
 2017-06-22 |   4 |    0.56 |          |
 2017-06-21 |   3 |    0.44 |          |
(56 строк)
```

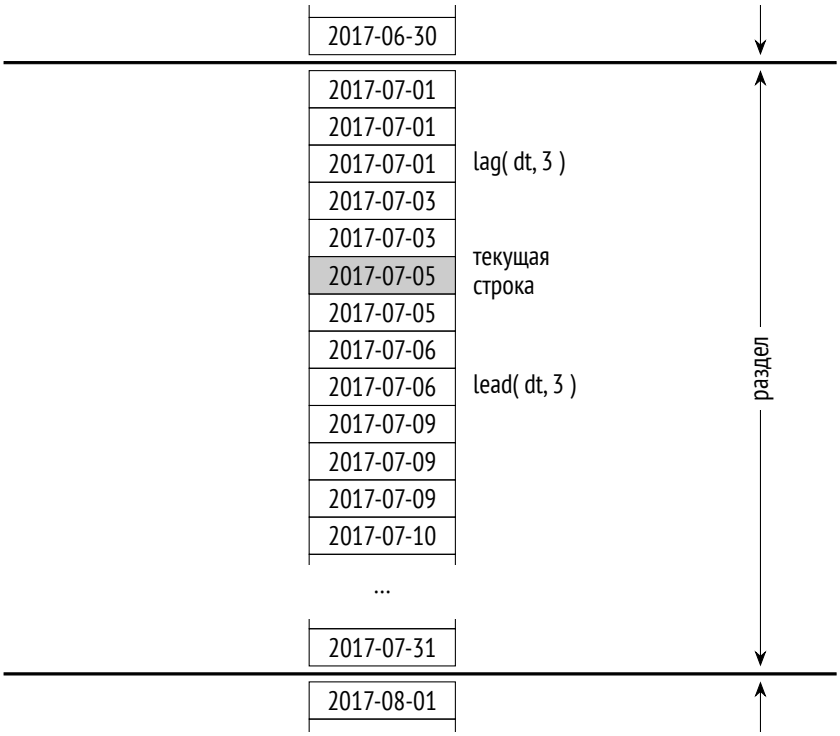


Рис. 3.7. *Функции lead и lag*

В общем табличном выражении вычисляются дневные суммы бронирований, они представлены в миллионах рублей. В определение окна мы не включили предложение `PARTITION BY`, чтобы в качестве раздела выступала вся выборка.

Значения в столбцах `day_sum` и `week_ago` сдвинуты друг относительно друга на неделю. Обратите внимание, что в конце выборки в столбцах `week_ago` и `delta` нет значений. Это объясняется тем, что функция `lead` возвращает значение `NULL`, если требуемой строки в выборке нет. Если нас такой исход не устраивает, можно передать функции третий параметр — значение, возвращаемое в подобных ситуациях. В нашем примере можно было бы в качестве эксперимента возвращать по умолчанию значение 0.

Глядя на полученный результат, можно сказать, что в конце июня и самом начале июля был быстрый прирост доходов, а в августе он стал не таким значительным. Вывод насчет той тенденции, которая сложилась в период времени,

не представленный в сокращенном варианте выборки, читатель может сделать сам, выполнив запрос и ознакомившись с полной выборкой.

В том же разделе документации представлена и функция lag. Она возвращает значение для строки, положение которой задается смещением от текущей строки к *началу* раздела (рис. 3.7). Аналогично функции lead она имеет третий параметр — значение по умолчанию. Рассмотренную задачу можно решить и с помощью этой функции: нужно лишь в определении окна задать сортировку по возрастанию дат и заменить функцию lead на функцию lag. Сортировку на уровне всего запроса оставим без изменений.

```
...
  lag( day_sum, 7 ) OVER all_rows_win AS week_ago,
  day_sum - lag( day_sum, 7 ) OVER all_rows_win AS delta
FROM day_amounts
WINDOW all_rows_win AS ( ORDER BY b_date )
ORDER BY b_date DESC;
```

Если сравнить рассмотренные функции, то можно отметить, что функции lead и lag работают в пределах всего раздела, а функции first_value, last_value и nth_value — в пределах оконного кадра. При этом первые две функции определяют позицию искомой строки *относительно текущей строки* с учетом порядка сортировки. А другая группа функций использует, образно выражаясь, *абсолютные* координаты, то есть первую, последнюю и *n*-ю строки.

Остальные оконные функции. Использование остальных оконных функций из раздела документации 9.22 «Оконные функции» покажем на примере небольшой таблицы «Самолеты» (aircrafts), что облегчит интерпретацию полученных результатов.

Предположим, что в нашей авиакомпании рассматривается вопрос приобретения новой модели самолета, а именно «Туполев Ту-204», имеющей дальность полета 6700 км. Совпадение значения этого показателя новой модели с моделью «Аэробус A319-100» позволит нам проиллюстрировать различия между функциями rank и dense_rank и особенности работы других функций.

В приведенном запросе создается один раздел, которым будет являться вся выборка. Столь короткие псевдонимы выбраны только для компактности вывода.

```
WITH aircrafts_plus AS
( SELECT aircraft_code, model, range
  FROM aircrafts
  UNION ALL
  VALUES ( 'T20', 'Туполев Ту-204', 6700 )
)
SELECT
  aircraft_code AS ac,
  model,
  range,
  row_number() OVER all_rows_win AS rn,
  rank() OVER all_rows_win AS r,
  dense_rank() OVER all_rows_win AS dr,
  round( ( percent_rank() OVER all_rows_win )::numeric, 2 ) AS pr,
  cume_dist() OVER all_rows_win AS cd,
  ntile( 4 ) OVER all_rows_win AS nt4,
  ntile( 3 ) OVER all_rows_win AS nt3
FROM aircrafts_plus
WINDOW all_rows_win AS ( ORDER BY range DESC )
ORDER BY rn;
```

```
 ac  |        model        | range | rn | r  | dr |  pr  | cd  | nt4 | nt3
-----+---------------------+-------+----+----+----+------+-----+-----+-----
 773 | Боинг 777-300       | 11100 |  1 |  1 |  1 | 0.00 | 0.1 |   1 |   1
 763 | Боинг 767-300       |  7900 |  2 |  2 |  2 | 0.11 | 0.2 |   1 |   1
 T20 | Туполев Ту-204      |  6700 |  3 |  3 |  3 | 0.22 | 0.4 |   1 |   1
 319 | Аэробус A319-100    |  6700 |  4 |  3 |  3 | 0.22 | 0.4 |   2 |   1
 320 | Аэробус A320-200    |  5700 |  5 |  5 |  4 | 0.44 | 0.5 |   2 |   2
 321 | Аэробус A321-200    |  5600 |  6 |  6 |  5 | 0.56 | 0.6 |   2 |   2
 733 | Боинг 737-300       |  4200 |  7 |  7 |  6 | 0.67 | 0.7 |   3 |   2
 SU9 | Сухой Суперджет-100 |  3000 |  8 |  8 |  7 | 0.78 | 0.8 |   3 |   3
 CR2 | Бомбардье CRJ-200   |  2700 |  9 |  9 |  8 | 0.89 | 0.9 |   4 |   3
 CN1 | Сессна 208 Караван  |  1200 | 10 | 10 |  9 | 1.00 |   1 |   4 |   3
(10 строк)
```

Глядя на полученную выборку, дадим комментарий по каждой из использованных оконных функций.

Функция `row_number` возвращает номер текущей строки в пределах раздела.

Функции `rank` и `dense_rank` обе возвращают ранг текущей строки, однако между ними есть важное отличие. При совпадении значений критерия сортировки, заданного предложением `ORDER BY` в определении окна, такие строки будут являться родственными. Обе функции назначают всем родственным строкам одинаковый ранг. Но строке, следующей за такой группой родственных строк, функция `dense_rank` назначает следующее число, по сути, ранжируя группы

строк. В отличие от нее, функция rank назначает такой ранг, какой был бы назначен этой строке, если бы родственных строк в данной выборке вообще не было, то есть она учитывает количество одинаковых значений и определяет следующий ранг с учетом этого факта. В результате в нумерации рангов получается пропуск. Эти особенности работы двух ранжирующих функций можно увидеть в столбцах r и dr полученной выборки. В столбце r за рангом номер 3 следует ранг номер 5, то есть имеет место пропуск в нумерации рангов, а в столбце dr за рангом номер 3 следует ранг номер 4.

Функция percent_rank возвращает относительный ранг. Он вычисляется по формуле

$$\frac{rank - 1}{\textit{общее число строк в разделе} - 1}.$$

Возвращаемое значение лежит в диапазоне от 0 до 1 включительно. Обратите внимание на совпадающие значения в столбце pr.

Функция cume_dist формирует так называемое кумулятивное распределение. Она возвращает значение, вычисляемое по формуле

$$\frac{\textit{число строк раздела, предшествующих или родственных текущей строке}}{\textit{общее число строк раздела}}.$$

Возвращаемое значение лежит в диапазоне от $1/N$ до 1 включительно, где N — число строк в разделе. В применении к нашему примеру значение, возвращаемое этой функцией, означает долю (часть) моделей, дальность полета которых *не менее* той, что имеет текущая модель. Обратите внимание на совпадающие значения в столбце cd.

Функция ntile разбивает раздел на группы строк, назначая каждой строке из группы один и тот же номер. Диапазон номеров — от 1 до того значения, которое, являясь аргументом функции, задает число групп. При этом разбиение строится так, чтобы число строк в группах было приблизительно одинаковым. В нашем примере эта функция формирует два столбца: в столбце nt4 — четыре группы строк, а в столбце nt3 — три группы строк. Обратите внимание, что функция никак не учитывает логику предметной области. Например, в разных группах оказались модели «Туполев Ту-204» и «Аэробус A319-100», имеющие одинаковую дальность полета.

3.5. Гипотезирующие агрегатные функции

Иногда при упорядочении каких-либо объектов на основе некоторого показателя может потребоваться выяснить, какое место занял бы гипотетический объект, если бы имел конкретное значение того показателя, по которому производится ранжирование. Решить такую задачу можно с помощью так называемых *гипотезирующих агрегатных функций* (Hypothetical-Set Aggregate Functions), то есть функций, работающих с гипотетическими множествами строк. Они представлены в разделе документации 9.21 «Агрегатные функции» (см. таблицу 9.63 «Гипотезирующие агрегатные функции»).

Предположим, что руководство нашей авиакомпании решило на приобретении самолета «Туполев Ту-204» не останавливаться, а заняться подбором модели с дальностью полета ровно 6000 км. Возникает вопрос: какой ранг такая модель получила бы при использовании функций rank и dense_rank?

Отвечая на него, мы можем обойтись без включения предполагаемой модели в набор данных. Для этого нужно те же функции rank и dense_rank, которые в предыдущем разделе главы мы использовали в качестве оконных функций, применить в качестве гипотезирующих, то есть ввести их в запрос с помощью конструкции WITHIN GROUP (ORDER BY ...). Очевидно, что присвоить корректные ранги без упорядочения строк невозможно. Количество и типы данных аргументов функций rank и dense_rank должны соответствовать столбцам, указанным в предложении WITHIN GROUP. В нашем примере функции получают только один числовой параметр целого типа, поэтому в предложении WITHIN GROUP указан лишь один столбец — range, — также имеющий целый тип.

```
WITH aircrafts_added AS
( SELECT aircraft_code, model, range
  FROM aircrafts
  UNION ALL
  VALUES ( 'I20', 'Туполев Ту-204', 6700 )
)
SELECT
  rank( 6000 ) WITHIN GROUP ( ORDER BY range DESC ),
  dense_rank( 6000 ) WITHIN GROUP ( ORDER BY range DESC )
FROM aircrafts_added;
```

Обратите внимание, что в запросе мы написали конструкцию WITHIN GROUP (ORDER BY ...) для *каждой* из функций.

Получаем ожидаемые результаты:

```
 rank | dense_rank
------+------------
    5 |          4
(1 строка)
```

Важно учитывать, что гипотезирующие функции, в отличие от большинства агрегатных функций, не отбрасывают значения NULL, а располагают их в выборке в том месте, которое предписано предложением ORDER BY (то есть в начале или в конце). По умолчанию значения NULL считаются больше любых настоящих значений.

Существуют еще две гипотезирующие функции, percent_rank и cume_dist, представленные в разделе документации 9.21 «Агрегатные функции». Предлагаем читателю ознакомиться с ними самостоятельно.

3.6. Контрольные вопросы и задания

1 Битовые строки тоже можно агрегировать: функции bit_and и bit_or

В разделе 3.1 «Агрегатные функции» (с. 109) были рассмотрены различные функции, в частности bool_and и bool_or. Однако мы не коснулись функций bit_and, bit_or и bit_xor. Давайте рассмотрим первые две из них на примере следующей ситуации. Модели самолетов можно дополнительно охарактеризовать несколькими показателями (сервисами), каждый из которых может либо присутствовать, либо отсутствовать у конкретной модели. Таким образом, показатели могут иметь истинное или ложное значение.

Для решения задачи воспользуемся типом данных bit, а конкретно bit(5), поскольку у нас будет всего пять показателей. Каждая позиция в битовой строке будет отвечать за конкретный показатель.

В конструкции WITH подзапросы all_facilities и aircrafts_equipment подготавливают исходные данные, которые затем агрегируются в подзапросе aggregates. Функция bit_and формирует битовую строку, в которой единицы означают, что сервис доступен *на всех* моделях самолетов. Единицы в битовой строке,

сформированной функцией bit_or, означают, что сервис доступен *хотя бы на одной* модели. В главном запросе формируются дополнительные показатели: сервисы, которыми оборудованы не все модели (not_all_equipped), и те, которыми не оборудована ни одна из них (no_one_equipped).

```
WITH all_facilities( facility_code, facility_name ) AS
( VALUES ( B'00001', 'система развлечений' ),
         ( B'00010', 'перевозка животных' ),
         ( B'00100', 'USB-розетки' ),
         ( B'01000', 'теплые пледы' ),
         ( B'10000', 'Wi-Fi в полете' )
),
aircrafts_equipment( aircraft_code, facilities ) AS
( VALUES ( 'SU9', B'01110' ),
         ( '320', B'01110' ),
         ( '773', B'01111' ),
         ( 'CN1', B'01000' )
),
aggregates AS
( SELECT
    bit_and( facilities ) AS all_equipped,
    bit_or( facilities ) AS at_least_one_equipped
  FROM aircrafts_equipment
)
SELECT
  all_equipped,
  ~all_equipped AS not_all_equipped,
  at_least_one_equipped,
  ~at_least_one_equipped AS no_one_equipped
FROM aggregates \gx
```

Интегральные показатели представлены в виде битовых масок, в которых каждая позиция соответствует конкретному частному показателю:

```
-[ RECORD 1 ]---------+------
all_equipped          | 01000
not_all_equipped      | 10111
at_least_one_equipped | 01111
no_one_equipped       | 10000
```

Однако более наглядным было бы представление интегрального показателя в виде совокупности его компонентов. Давайте посмотрим, например, какие сервисы предлагаются всеми моделями самолетов. Обратите внимание, что имя интегрального показателя all_equipped задается в предложении ON главного запроса.

```
...
aggregates AS
( SELECT
    bit_and( facilities ) AS all_equipped,
    bit_or( facilities ) AS at_least_one_equipped
  FROM aircrafts_equipment
),
finals AS
( SELECT
    all_equipped,
    ~all_equipped AS not_all_equipped,
    at_least_one_equipped,
    ~at_least_one_equipped AS no_one_equipped
  FROM aggregates
)
SELECT af.facility_code, af.facility_name
FROM finals AS f
  JOIN all_facilities AS af ON ( af.facility_code & f.all_equipped )::int > 0;
```

```
 facility_code | facility_name
---------------+---------------
 01000         | теплые пледы
(1 строка)
```

Для получения состава всех интегральных показателей придется выполнить запрос четыре раза, изменяя имя показателя.

Задание 1. Решите эту задачу с помощью одного запроса, представив результат в таком виде:

```
   agg_name   |                              facilities
--------------+-------------------------------------------------------------------
 все модели   | теплые пледы
 не все       | система развлечений, перевозка животных, USB-розетки, Wi-Fi в полете
 ни одной     | Wi-Fi в полете
 хотя бы одна | система развлечений, перевозка животных, USB-розетки, теплые пледы
(4 строки)
```

Указание. В качестве одного из решений можно рассмотреть следующее. Измените подзапрос aggregates в конструкции WITH таким образом, чтобы в результате его выполнения каждому интегральному показателю соответствовала строка, а не столбец. Для этого можно воспользоваться оператором UNION ALL. В главном запросе соедините подзапросы aggregates и all_facilities аналогично тому, как это было сделано в тексте упражнения. Учтите также, что агрегатные функции могут использовать предложение ORDER BY.

Задание 2. Рассмотренную задачу можно решить и с использованием других средств, например типа данных JSON. Реализуйте такой вариант и сравните два решения с точки зрения сложности кода, удобства добавления новых показателей и т. д.

2 Битовые строки тоже можно агрегировать: функция bit_xor

Существует еще одна агрегатная функция для работы с битовыми строками — bit_xor. В документации в разделе 9.21 «Агрегатные функции» сказано, что эта функция может, например, использоваться для вычисления контрольной суммы неупорядоченного набора значений.

Задание. Вычислите значение этой функции для одной из таблиц базы данных «Авиаперевозки». Например:

```
SELECT bit_xor( range ) AS check_sum FROM aircrafts;
```

Подумайте, можно ли использовать эту функцию для обнаружения изменений в базе данных (случайных или преднамеренных). Учтите, что модификацию одной строки можно компенсировать модификацией другой, сохранив значение функции bit_xor неизменным.

Сравните возможности этой функции с другими методами, например вычислением кодов CRC.

3 В каких частях запроса можно использовать агрегатные выражения?

В приведенном ниже запросе агрегатные выражения присутствуют в предложениях HAVING и ORDER BY. Как вы думаете, будет ли запрос корректно выполняться? Сначала сделайте обоснованное предположение, а затем проверьте его, выполнив запрос. Найдите в документации объяснение полученным результатам.

```
SELECT a.aircraft_code, a.model, a.range, count( * )
FROM aircrafts a
  JOIN seats s ON s.aircraft_code = a.aircraft_code
GROUP BY a.aircraft_code, a.model, a.range
HAVING count( * ) FILTER ( WHERE s.fare_conditions = 'Business' ) > 25
ORDER BY count( * ) FILTER ( WHERE s.fare_conditions = 'Business' ) DESC;
```

4 Агрегирование в параллельном режиме

В подразделе 3.1.2 «Агрегирование в параллельном режиме» (с. 114) мы рассматривали выполнение запросов с агрегатными функциями в параллельном режиме. Однако в том случае, когда предполагается, что формируемых групп будет относительно много (в предельном случае их число равно числу исходных строк), планировщик может отказаться от создания параллельного плана (см. подраздел документации 15.3.3 «Параллельное агрегирование»).

Для иллюстрации сказанного давайте рассмотрим два запроса к таблице «Посадочные талоны» (boarding_passes). План первого запроса, подсчитывающего число пассажиров на каждом рейсе, будет параллельным:

```
EXPLAIN ( costs off )
SELECT flight_id, count( * )
FROM boarding_passes
GROUP BY flight_id;
                    QUERY PLAN
--------------------------------------------------
 Finalize HashAggregate
   Group Key: flight_id
   -> Gather
       Workers Planned: 2
       -> Partial HashAggregate
           Group Key: flight_id
           -> Parallel Seq Scan on boarding_passes
(7 строк)
```

Если добавить в предложение GROUP BY этого запроса еще один столбец, то план параллельным уже не будет, поскольку число формируемых групп будет относительно велико, а точнее говоря, равно числу исходных строк:

```
EXPLAIN ( costs off )
SELECT flight_id, boarding_no, count( * )
FROM boarding_passes
GROUP BY flight_id, boarding_no;
             QUERY PLAN
-----------------------------------
 HashAggregate
   Group Key: flight_id, boarding_no
   -> Seq Scan on boarding_passes
(3 строки)
```

Задание. Предложите свой вариант запросов, выполняемых в параллельном режиме и без распараллеливания.

5 Агрегирование числовых данных, содержащихся в JSON-объектах

В объектах типа JSON могут содержаться числовые данные, которые тоже можно агрегировать. Однако для получения корректного результата придется принять дополнительные меры.

В качестве примера выполним запрос к таблице «Самолеты» (aircrafts). При этом в дополнение к столбцам таблицы с помощью функции row_to_json сформируем JSON-объект из всех полей строки, а затем воспользуемся агрегатными функциями min и max для определения минимальной и максимальной дальности полета самолетов. Причем для сравнения определим эти показатели как на основе исходного столбца range, имеющего числовой тип, так и на основе поля range, содержащегося в JSON-объекте.

Обратите внимание, что параметром функции row_to_json является имя таблицы. Когда таблица создается, автоматически создается и одноименный *составной тип* данных, соответствующий строке этой таблицы. Поэтому функция row_to_json фактически получает значения *всех* полей текущей строки.

```
WITH aircrafts_tmp AS
( SELECT *, row_to_json( aircrafts )::jsonb AS info
  FROM aircrafts
)
SELECT
  max( range ),
  min( range),
  max( info->>'range' ) AS max_json,
  min( info->>'range' ) AS min_json
FROM aircrafts_tmp;
```

```
  max  | min  | max_json | min_json
-------+------+----------+----------
 11100 | 1200 | 7900     | 11100
(1 строка)
```

Вопрос. Как можно объяснить полученные, казалось бы, парадоксальные, результаты?

Указание. Обратите внимание на оператор ->>. Данные какого типа он возвращает? За справкой обратитесь к документации: раздел 9.16 «Функции и операторы JSON».

Для лучшего понимания того, что выдает подзапрос в предложении WITH, выполните его отдельно.

Задание 1. Модифицируйте вызовы функций max и min с полями JSON-объектов таким образом, чтобы получить корректные результаты. Попробуйте заменить оператор ->> на ->. Поможет ли это решить задачу? Внимательно изучите сообщения об ошибках, если они будут.

Задание 2. Попробуйте убрать приведение к типу jsonb результата функции row_to_json и повторите эксперименты.

6 Как первичный ключ влияет на выбор группируемых столбцов?

Если потребуется узнать общее число мест в каждой модели самолета, то решить эту задачу может такой запрос:

```
SELECT
  a.aircraft_code,
  a.model AS model,
  a.range,
  count( * ) AS seats_num
FROM aircrafts a
  JOIN seats s ON s.aircraft_code = a.aircraft_code
GROUP BY a.aircraft_code, a.model, a.range
ORDER BY seats_num DESC;
 aircraft_code |        model        | range | seats_num
---------------+---------------------+-------+-----------
 773           | Боинг 777-300       | 11100 |       402
 763           | Боинг 767-300       |  7900 |       222
 321           | Аэробус A321-200    |  5600 |       170
 320           | Аэробус A320-200    |  5700 |       140
 733           | Боинг 737-300       |  4200 |       130
 319           | Аэробус A319-100    |  6700 |       116
 SU9           | Сухой Суперджет-100 |  3000 |        97
 CR2           | Бомбардье CRJ-200   |  2700 |        50
 CN1           | Сессна 208 Караван  |  1200 |        12
(9 строк)
```

Объект «Самолеты» (aircrafts) является представлением. Оно создано на основе таблицы «Самолеты» (aircrafts_data). Узнать, как устроены таблица и представление, можно с помощью команд утилиты psql:

```
\d aircrafts
\sv aircrafts
\d aircrafts_data
```

Поставленную задачу можно решить и с помощью обращения к таблице:

```
SELECT
  a.aircraft_code,
  a.model ->> lang() AS model,
  a.range,
  count( * ) AS seats_num
FROM aircrafts_data a
  JOIN seats s ON s.aircraft_code = a.aircraft_code
GROUP BY a.aircraft_code
ORDER BY seats_num DESC;
```

Функция lang возвращает значение параметра локализации, отвечающего за языковые настройки системы (по умолчанию — ru). Это не встроенная функция PostgreSQL, она создана в базе данных «Авиаперевозки».

Попутно вспомните, что означает синтаксис:

```
a.model ->> lang()
```

Вопрос. Почему во втором запросе в предложении GROUP BY указаны не три столбца, как в первом запросе, а только один? Нет ли здесь какого-то противоречия с подразделом документации 7.2.3 «Предложения GROUP BY и HAVING», в котором говорится, что столбцы, не включенные в список GROUP BY, можно использовать только в агрегатных выражениях? Из этого следует, что указать их в предложении SELECT можно только в качестве параметров агрегатных функций, однако во втором запросе это не так. Можно ли и в предложении GROUP BY первого запроса также оставить только один столбец a.aircraft_code?

Указание. Добавьте в предложение GROUP BY второго запроса столбцы a.model и a.range и посмотрите планы выполнения обоих вариантов запроса. Обратите внимание на строку с меткой «Group Key». Вспомните о свойствах первичного ключа.

7 Эксперимент с процентилями

В разделе 3.2 «Статистические функции» (с. 126) мы вычисляли процентили (децили) для коротких задержек рейсов, то есть задержек длительностью не более одного часа. В запросе, приведенном в этом разделе, использовалась функция percentile_disc. При этом в предложении WITHIN GROUP (ORDER BY), которым сопровождается вызов функции, был задан порядок сортировки задержек по возрастанию длительности.

Если в указанном предложении задать порядок сортировки по убыванию (ключевое слово DESC), то будет ли запрос выполняться успешно? Давайте проведем этот эксперимент, расположив и элементы массива, являющегося параметром функции percentile_disc, по убыванию, то есть от 1,0 к 0,1. Обратите внимание, что если в запросе присутствует несколько вызовов этой функции, то порядки сортировки в предложениях WITHIN GROUP могут различаться.

```
WITH deciles AS
( SELECT
    percentile_disc( ARRAY[ 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0 ] )
      WITHIN GROUP ( ORDER BY delay ) AS deciles,
    percentile_disc( ARRAY[ 1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1 ] )
      WITHIN GROUP ( ORDER BY delay DESC ) AS deciles_invert
  FROM short_delays
)
SELECT
  unnest( ARRAY[ 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0 ] ) AS level,
  unnest( deciles ) AS decile,
  unnest( ARRAY[ 1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1 ] ) AS level_invert,
  unnest( deciles_invert ) AS decile_invert
FROM deciles;
```

```
 level |  decile  | level_invert | decile_invert
-------+----------+--------------+---------------
   0.1 | 00:01:00 |          1.0 | 00:01:00
   0.2 | 00:02:00 |          0.9 | 00:01:00
   0.3 | 00:02:00 |          0.8 | 00:02:00
   0.4 | 00:03:00 |          0.7 | 00:02:00
   0.5 | 00:03:00 |          0.6 | 00:03:00
   0.6 | 00:03:00 |          0.5 | 00:03:00
   0.7 | 00:04:00 |          0.4 | 00:03:00
   0.8 | 00:04:00 |          0.3 | 00:04:00
   0.9 | 00:05:00 |          0.2 | 00:04:00
   1.0 | 00:11:00 |          0.1 | 00:05:00
(10 строк)
```

Вопрос. Если подойти к делу формально, то мы видим, что такой запрос работает. Но как в таком случае можно интерпретировать полученный результат? Будет ли эта интерпретация математически строгой? Будет ли она иметь содержательный смысл? При интерпретации процентилей используют такую фразу: «Такая-то доля всех значений в выборке не превышает такой-то величины». А какую фразу можно было бы использовать в нашем эксперименте? И конкретный вопрос: почему в столбце decile присутствует значение `00:11:00`, а в столбце `decile_invert` его нет, но при этом повторяется значение `00:01:00`?

8 Аргумент-массив функции можно сгенерировать, но есть нюансы

Если массив границ, передаваемый функции `percentile_disc`, содержит небольшое число элементов, их можно перечислить в виде литерала. Но чтобы вычислить процентили с шагом, скажем, 5 %, хотелось бы иметь способ генерирования этого массива, избавляющий нас от ручного ввода всех значений. И такой способ есть — это конструктор `ARRAY` в сочетании с функцией `generate_series`:

```
WITH percentiles AS
( SELECT percentile_disc(
      ARRAY( SELECT generate_series( 0.05, 1.0, 0.05 ) )
    )
    WITHIN GROUP ( ORDER BY delay ) AS percentiles
  FROM short_delays
)
SELECT
  generate_series( 0.05, 1.0, 0.05 ) AS level,
  unnest( percentiles ) AS percentile
FROM percentiles;
```

```
 level | percentile
-------+------------
  0.05 | 00:01:00
  0.10 | 00:01:00
  0.15 | 00:02:00
  0.20 | 00:02:00
  0.25 | 00:02:00
 ...
  0.85 | 00:05:00
  0.90 | 00:05:00
  0.95 | 00:06:00
  1.00 | 00:11:00
(20 строк)
```

Вопрос. Почему нельзя написать просто:

```
...
SELECT percentile_disc( ARRAY[ ( generate_series( 0.05, 1.0, 0.05 ) ) ] )
...
```

Ведь такой запрос выполняется корректно:

```
SELECT ARRAY[ ( generate_series( 0.05, 1.0, 0.05 ) ) ];
```

9 Небольшое статистическое исследование

Когда число диапазонов при вычислении процентилей равно четырем, мы имеем дело с *квартилями*. Давайте вычислим квартили для общих сумм бронирования, поскольку руководство компании проявляет интерес к закономерностям продажи билетов.

```
WITH quartiles AS
( SELECT percentile_disc( ARRAY[ 0.25, 0.5, 0.75, 1.0 ] )
         WITHIN GROUP ( ORDER BY total_amount ) AS quartile
  FROM bookings
)
SELECT
  unnest( ARRAY[ 0.25, 0.5, 0.75, 1.0 ] ) AS level,
  unnest( quartile ) AS quartile
FROM quartiles;
```

```
 level |  quartile
-------+------------
  0.25 |   29000.00
   0.5 |   55900.00
  0.75 |   99200.00
   1.0 | 1204500.00
(4 строки)
```

Таким образом, для четверти всех операций бронирования полная стоимость каждой из них не превышает 29 000 рублей. Обратите внимание на большую разницу значений третьего и четвертого квартилей. На графике это отразилось бы в виде крутого роста значений в этой области.

Информативным показателем может быть *межквартильный размах*. Он определяется как разность между третьим и первым квартилями. В нашем примере

это 99 200 – 29 000 = 70 200. Таким образом, межквартильный размах представляет собой диапазон, в который попадает 50 % всех значений из выборки.

Теперь вычислим описательные статистики — минимальное, максимальное и среднее значения, медиану, дисперсию и среднеквадратическое отклонение — для всего множества операций бронирования:

```
SELECT
  count( * ),
  min( total_amount ),
  max( total_amount ),
  percentile_disc( 0.5 ) WITHIN GROUP ( ORDER BY total_amount ) AS median,
  round( avg( total_amount ), 2 ) AS average,
  round( stddev_pop( total_amount ), 2 ) AS stddev,
  round( var_pop( total_amount ), 2 ) AS variance
FROM bookings \gx
-[ RECORD 1 ]-----------
count    | 262788
min      | 3400.00
max      | 1204500.00
median   | 55900.00
average  | 79025.61
stddev   | 77621.78
variance | 6025139959.40
```

Часто принято сравнивать среднее значение со значением медианы, то есть процентиля уровня 0,5. Мы видим, что в этой выборке среднее значение выше медианы. Это объясняется тем, что у небольшой части операций бронирования полная стоимость весьма высока. Видно, что и значение дисперсии высокое. Это можно было предположить заранее, исходя из минимальной и максимальной полной стоимости бронирования.

Задание 1. Вычислите эти же статистики для случайной выборки объемом 10 % из исходного множества операций бронирования. Теперь нужно использовать функции `stddev_samp` и `var_samp` для вычисления выборочного среднеквадратического отклонения и выборочной дисперсии. В запросе также изменится предложение `FROM`. Именно здесь задается способ случайного отбора строк и их доля от всей доступной совокупности.

```
SELECT ...
...
FROM bookings TABLESAMPLE BERNOULLI( 10 ) \gx
```

Параметр BERNOULLI — это метод случайного отбора строк из таблицы. Есть еще один встроенный метод — SYSTEM. В чем заключаются различия между ними, предлагаем читателю разобраться самостоятельно с помощью описания команды SELECT, приведенного в документации.

Выполните этот запрос несколько раз: каждый раз будут получаться разные результаты, поскольку мы не включили в предложение TABLESAMPLE параметр REPEATABLE. Тем не менее получаемые результаты будут обладать определенной устойчивостью.

Задание 2. Выполните этот запрос еще несколько раз, отбирая из таблицы «Бронирования» (bookings) только 1 % строк.

```
...
FROM bookings TABLESAMPLE BERNOULLI( 1 ) \gx
```

Сохраняется ли устойчивость результатов при таком объеме выборки? Как на это влияет факт, что доля очень больших значений в таблице мала?

10 Сопоставление конструкции UNION с конструкцией GROUPING SETS

В подразделе 3.3.1 «Группировка с помощью GROUPING SETS» (с. 137) были показаны два варианта решения задачи подсчета количества маршрутов, обслуживаемых самолетами разных моделей в каждом аэропорту. В первом варианте использовалась конструкция UNION, во втором — GROUPING SETS. Причем второй работал гораздо быстрее первого.

Задание. Попытайтесь найти объяснение большой разнице во времени выполнения в пользу запроса с конструкцией GROUPING SETS. Посмотрите планы запросов.

Указание. Планы получаются очень громоздкими из-за того, что в запросах используется представление «Маршруты» (routes). Для упрощения планов можно создать временную таблицу на основе этого представления:

```
CREATE TEMP TABLE routes_t AS SELECT * FROM routes;
```

Конечно, в запросах нужно изменить имя routes на имя новой таблицы. Обратите внимание на число сканирований таблицы routes_t в каждом из планов.

11 Влияет ли порядок следования групп столбцов в конструкции GROUPING SETS на работу запроса?

В подразделе 3.3.1 «Группировка с помощью GROUPING SETS» (с. 137) был приведен запрос, вычисляющий количество рейсов, выполняемых самолетами разных моделей из каждого аэропорта. В том запросе группировка выполнялась по комбинации наименования аэропорта и наименования модели самолета.

Задание. Измените порядок группирования строк на обратный: первым должно идти наименование модели самолета, а затем наименование аэропорта. Модифицируйте запрос таким образом, чтобы он выводил отчет в следующем виде:

```
            model              |     airport_name     | routes_count
-------------------------------+----------------------+--------------
 Аэробус A319-100              | Астрахань            |            1
 Аэробус A319-100              | Байкал               |            1
 Аэробус A319-100              | Богашёво             |            1
 Аэробус A319-100              | Братск               |            1
 Аэробус A319-100              | Казань               |            1
 ...
 Аэробус A319-100              | Талаги               |            2
 Аэробус A319-100              | Хомутово             |            2
 Аэробус A319-100              | Внуково              |            3
 Аэробус A319-100              | Шереметьево          |            3
 Аэробус A319-100              | Анадырь              |            4
 Аэробус A319-100              | Чульман              |            4
 Аэробус A319-100              | Домодедово           |            5
   Всего по Аэробус A319-100   |                      |           46
 Аэробус A321-200              | Иркутск              |            1
 Аэробус A321-200              | Казань               |            1
 ...
 Сухой Суперджет-100           | Брянск               |           10
 Сухой Суперджет-100           | Домодедово           |           18
 Сухой Суперджет-100           | Шереметьево          |           18
   Всего по Сухой Суперджет-100 |                     |          158
                               | Братск               |            1
                               | Елизово              |            1
                               | Игнатьево            |            1
 ...
                               | Пулково              |           35
                               | Шереметьево          |           57
                               | Домодедово           |           62
 ИТОГО                         |                      |          710
(387 строк)
```

Обязательно ли порядок следования имен столбцов в конструкции GROUPING SETS должен совпадать с порядком их вывода в выборке?

```
GROUP BY GROUPING SETS
( ( a.model, r.departure_airport_name ),
  ( a.model ),
  ( r.departure_airport_name ),
  ( )
)
```

А если сделать так, это повлияет на выполнение запроса?

```
GROUP BY GROUPING SETS
( ( a.model ),
  ( a.model, r.departure_airport_name ),
  ( ),
  ( r.departure_airport_name )
)
```

12 Конструкция ROLLUP и ее отражение в плане запроса

В подразделе 3.3.2 «Группировка с помощью ROLLUP» (с. 142) был представлен запрос с конструкцией ROLLUP, который показывал динамику бронирований авиабилетов с разбиением по дням, декадам месяцев и месяцам. Давайте рассмотрим план его выполнения. При этом отключим вывод оценок затрат ресурсов с помощью параметра COSTS OFF, так как нас сейчас интересует только вид выполняемых операций, а не их стоимости.

```
EXPLAIN (costs off)
SELECT
  extract( month FROM book_date ) AS month,
  CASE
    WHEN extract( day FROM book_date ) <= 10 THEN 1
    WHEN extract( day FROM book_date ) <= 20 THEN 2
    ELSE 3
  END AS ten_days,
  extract( day FROM book_date ) AS day,
  count( * ) AS book_num,
  round( sum( total_amount ) / 1000000, 2 ) AS amount
FROM bookings
GROUP BY ROLLUP( month, ten_days, day )
ORDER BY month, ten_days, day;
```

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 GroupAggregate
   Group Key: (EXTRACT(month FROM book_date)),
              (CASE WHEN (EXTRACT(day FROM book_date) <= '10'::numeric) THEN ... END),
              (EXTRACT(day FROM book_date))
   Group Key: (EXTRACT(month FROM book_date)),
              (CASE WHEN (EXTRACT(day FROM book_date) <= '10'::numeric) THEN ...  END)
   Group Key: (EXTRACT(month FROM book_date))
   Group Key: ()
   -> Sort
       Sort Key: (EXTRACT(month FROM book_date)),
                 (CASE WHEN (EXTRACT(day FROM book_date) <= '10'::numeric) THEN ... END),
                 (EXTRACT(day FROM book_date))
       -> Seq Scan on bookings
(8 строк)
```

Обратите внимание, что в узле GroupAggregate присутствуют четыре группировки. Наборы группируемых столбцов соответствуют тем, какие мы получили бы, заменив конструкцию ROLLUP на GROUPING SETS.

В этом плане в верхнем узле нет операции сортировки строк, хотя в запросе присутствует предложение ORDER BY. Обратите внимание на порядок следования столбцов в предложении ORDER BY и в предложении ROLLUP. В приведенном случае агрегирование выполняется на основе сортировки строк, а не хеширования. А поскольку порядок сортировки в предложении ORDER BY указан такой же, как и порядок группирования в предложении ROLLUP, то планировщику достаточно только одной сортировки. Хотя предложение ROLLUP фактически заменяется несколькими группировками, но все они содержат подмножество столбцов из предложения ORDER BY, при этом порядок следования столбцов в этих подмножествах такой же, как и в предложении ORDER BY.

Задание. Посмотрите, какими будут планы запросов, если изменить столбцы в предложении ORDER BY на такие варианты, имеющие практический смысл:

```
...
ORDER BY month DESC, ten_days DESC, day DESC;
...
ORDER BY month, ten_days, amount DESC;
```

Конечно, не только посмотрите планы, но также и выполните сами запросы. Изучите их вывод. Попробуйте предложить правдоподобное обоснование практической целесообразности данных запросов.

13 Конструкция CUBE, предложение HAVING и функция GROUPING

В подразделе 3.3.3 «Группировка с помощью CUBE» (с. 145) был приведен запрос, формирующий детальный отчет о распределении продаж билетов по направлениям, моделям самолетов и классам обслуживания.

```
SELECT
  r.departure_airport AS da,
  -- GROUPING( r.departure_airport ) AS da_g,
  r.arrival_airport AS aa,
  -- GROUPING( r.arrival_airport ) AS aa_g,
  a.model,
  -- GROUPING( a.model ) AS m_g,
  left( tf.fare_conditions, 1 ) AS fc,
  -- GROUPING( tf.fare_conditions ) AS fc_g,
  count( * ),
  round( sum( tf.amount ) / 1000000, 2 ) AS t_amount,
  GROUPING( r.departure_airport, r.arrival_airport, a.model, tf.fare_conditions )::bit( 4 )
    AS mask,
  concat_ws( ',',
    CASE WHEN GROUPING( r.departure_airport ) = 0 THEN 'da' END,
    CASE WHEN GROUPING( r.arrival_airport ) = 0 THEN 'aa' END,
    CASE WHEN GROUPING( a.model ) = 0 THEN 'm' END,
    CASE WHEN GROUPING( tf.fare_conditions ) = 0 THEN 'fc' END
  ) AS grouped_cols
FROM routes r
  JOIN flights f ON f.flight_no = r.flight_no
  JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  JOIN aircrafts a ON a.aircraft_code = r.aircraft_code
GROUP BY CUBE
( ( da, aa ),
  a.model,
  tf.fare_conditions
)
ORDER BY da, aa, a.model, fc;
```

В предложении CUBE здесь используются как псевдонимы столбцов из списка SELECT (это da и aa), так и имена столбцов, а аргументами в вызовах функции GROUPING служат только имена столбцов.

Вопрос 1. Можно ли в предложении CUBE вместо имени столбца tf.fare_conditions написать псевдоним fc, то есть имя выходного столбца? Сначала сделайте обоснованное предположение, а потом проверьте его практически.

Вопрос 2. Будет ли работать запрос, если в предложении CUBE вместо имени столбца tf.fare_conditions написать псевдоним fc, а в списке SELECT вместо выражения left(tf.fare_conditions, 1) написать просто tf.fare_conditions?

Указание. Для объяснения полученных результатов проведите необходимые эксперименты. В дополнение к тем, что описаны в вопросах 1 и 2, проведите, например, такой: вместо имени столбца tf.fare_conditions во всех случаях его использования напишите left(tf.fare_conditions, 1), как это сделано в списке SELECT. Изучите полученные сообщения об ошибках.

Вопрос 3. Почему мы можем использовать псевдонимы столбцов в предложении CUBE, но не можем этого сделать в функции GROUPING?

```
...
  GROUPING( da, aa, a.model, tf.fare_conditions ) AS mask
...
```

Указание. В поиске ответа может помочь описание порядка обработки команды SELECT, приведенное в начале раздела документации, посвященного этой команде.

Задание 1. Пусть требуется выбрать, скажем, только итоги по классам обслуживания. Первый вариант модификации запроса может быть таким:

```
...
GROUP BY CUBE ( ( da, aa ), a.model, tf.fare_conditions )
HAVING tf.fare_conditions IS NOT NULL
   AND r.departure_airport IS NULL
   AND r.arrival_airport IS NULL
   AND a.model IS NULL
ORDER BY da, aa, a.model, fc;
 da | aa | model | fc | count  | t_amount | mask | grouped_cols
----+----+-------+----+--------+----------+------+--------------
    |    |       | B  | 107642 |  5505.18 | 1110 | fc
    |    |       | C  |  17291 |   566.12 | 1110 | fc
    |    |       | E  | 920793 | 14695.68 | 1110 | fc
(3 строки)
```

Такое решение в нашем случае работает корректно, поскольку в исходных таблицах столбцы, включенные в предложение GROUP BY, не содержат значений NULL. Однако более универсальным подходом было бы использование функции GROUPING.

Например:

```
...
GROUP BY CUBE ( ( da, aa ), a.model, tf.fare_conditions )
HAVING GROUPING(
    tf.fare_conditions, r.arrival_airport, r.departure_airport, a.model
  )::bit( 4 ) = '0111'::bit( 4 )
ORDER BY da, aa, a.model, fc;
```

Функция GROUPING возвращает целое число, но можно преобразовать его в битовую маску для удобства сравнения значений. Важно учитывать, что возвращаемое значение зависит от порядка следования имен столбцов, являющихся аргументами функции. Этот порядок не обязан совпадать с порядком в предложении GROUP BY.

Возможно, более удобным будет комбинирование отдельных вызовов функции GROUPING для каждого интересующего нас столбца с помощью логической операции AND. Выполните такую модификацию запроса самостоятельно.

Указание. Поскольку каждый вызов функции будет иметь только один параметр, нет необходимости приводить возвращаемое значение к типу bit(1) — можно использовать целочисленное сравнение.

Задание 2. Рассмотренный запрос формирует отчет, содержащий целый ряд составных частей. При необходимости можно выбрать конкретный фрагмент отчета с помощью предложения HAVING. Например, получить сведения о направлении Санкт-Петербург — Иркутск можно таким способом:

```
...
GROUP BY CUBE ( ( da, aa ), a.model, tf.fare_conditions )
HAVING r.departure_airport = 'LED'
   AND r.arrival_airport = 'IKT'
ORDER BY da, aa, a.model, fc;
 da  | aa  |      model       | fc | count | t_amount | mask | grouped_cols
-----+-----+------------------+----+-------+----------+------+--------------
 LED | IKT | Аэробус A321-200 | B  |  1142 |   151.77 | 0000 | da,aa,m,fc
 LED | IKT | Аэробус A321-200 | E  |  5816 |   259.83 | 0000 | da,aa,m,fc
 LED | IKT | Аэробус A321-200 |    |  6958 |   411.60 | 0001 | da,aa,m
 LED | IKT |                  | B  |  1142 |   151.77 | 0010 | da,aa,fc
 LED | IKT |                  | E  |  5816 |   259.83 | 0010 | da,aa,fc
 LED | IKT |                  |    |  6958 |   411.60 | 0011 | da,aa
(6 строк)
```

Обратите внимание, что использовать псевдонимы столбцов в предложении HAVING нельзя. Такой код вызовет ошибку:

```
...
HAVING da = 'LED'
   AND aa = 'IKT'
...
ОШИБКА:  столбец "da" не существует
СТРОКА 23: HAVING da = 'LED' AND aa = 'IKT'
                  ^
```

Учитывая, что единого решения на все случаи жизни не существует, сопоставьте два подхода:

1) модифицирование (например, с помощью предложения HAVING) подобного универсального запроса, содержащего конструкции GROUPING SETS, CUBE или ROLLUP;
2) создание специальных запросов, не использующих этих конструкций, для решения конкретных задач.

Учтите такие факторы, как сложность запроса, удобство его модифицирования для получения различных частных результатов, частоту возникновения потребности в них, быстродействие и т. д.

14 Сопоставление конструкции UNION с конструкцией CUBE

В подразделе 3.3.3 «Группировка с помощью CUBE» (с. 145) было показано решение задачи с помощью конструкции CUBE. Эту задачу — формирование детального отчета о распределении продаж билетов по направлениям, моделям самолетов и классам обслуживания — можно решить и с помощью оператора UNION ALL и восьми подзапросов (по числу комбинаций столбцов, формируемых при обработке предложения CUBE). Поскольку наборы группируемых столбцов будут различаться, списки SELECT в подзапросах также будут различаться, поэтому в ряде случаев придется вместо конкретного столбца использовать NULL, чтобы структура выборок была регулярной.

В этом запросе предложения FROM будут одинаковыми, а предложения GROUP BY будут различаться (в последнем подзапросе GROUP BY вовсе не будет):

```
SELECT
  r.departure_airport AS da,
  r.arrival_airport AS aa,
  a.model, left( tf.fare_conditions, 1 ) AS fc,
  count( * ),
  round( sum( tf.amount ) / 1000000, 2 ) AS t_amount,
  'da,aa,model,fc' AS grouped_ cols
FROM routes r
  JOIN flights f ON f.flight_no = r.flight_no
  JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  JOIN aircrafts a ON a.aircraft_code = r.aircraft_code
GROUP BY da, aa, a.model, tf.fare_conditions
UNION ALL
SELECT
  r.departure_airport AS da,
  r.arrival_airport AS aa,
  a.model,
  NULL AS fc,
  count( * ),
  round( sum( tf.amount ) / 1000000, 2 ) AS t_amount,
  'da,aa,model' AS grouped_ cols
FROM routes r
  JOIN flights f ON f.flight_no = r.flight_no
  JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  JOIN aircrafts a ON a.aircraft_code = r.aircraft_code
GROUP BY da, aa, a.model
UNION ALL
--
-- Здесь нужно добавить еще пять подзапросов
--
UNION ALL
SELECT
  NULL AS da,
  NULL AS aa,
  NULL AS model,
  NULL AS fc,
  count( * ),
  round( sum( tf.amount ) / 1000000, 2 ) AS t_amount,
  NULL AS grouped_cols
FROM routes r
  JOIN flights f ON f.flight_no = r.flight_no
  JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  JOIN aircrafts a ON a.aircraft_code = r.aircraft_code
ORDER BY da, aa, model, fc;
```

Задание 1. Допишите запрос, добавив в него пять подзапросов. Сравните время выполнения этого запроса с тем, что был рассмотрен в тексте главы.

Задание 2. Посмотрите планы запросов. Попытайтесь найти объяснение большой разнице во времени выполнения в пользу запроса с конструкцией CUBE.

Указание. Планы получаются очень громоздкими из-за того, что в запросах используется представление «Маршруты» (routes). Для их упрощения можно создать временную таблицу на основе этого представления:

```
CREATE TEMP TABLE routes_t AS SELECT * FROM routes;
```

15 Комбинирование конструкций GROUPING SETS, CUBE и ROLLUP

В подразделе документации 7.2.4 «GROUPING SETS, CUBE и ROLLUP» сказано, что конструкции CUBE и ROLLUP могут быть вложены в GROUPING SETS, а в предложении GROUP BY может быть записано несколько этих конструкций.

Для иллюстрации обратимся к запросу, приведенному в подразделе 3.3.3 «Группировка с помощью CUBE» (с. 145), который формировал детальный отчет о распределении продаж билетов по направлениям полетов, моделям самолетов и классам обслуживания. Давайте упростим его, исключив столбцы mask и grouped_cols, поскольку сейчас они нам не потребуются. Также мы откажемся от детальных группировок по направлениям полетов, но будем подсчитывать по каждому аэропорту отправления и прибытия общее число пассажиров и суммарную стоимость их билетов.

```
SELECT
  r.departure_airport AS da,
  r.arrival_airport AS aa,
  a.model,
  left( tf.fare_conditions, 1 ) AS fc,
  count( * ),
  round( sum( tf.amount ) / 1000000, 2 ) AS t_amount
FROM routes r
  JOIN flights f ON f.flight_no = r.flight_no
  JOIN ticket_flights tf ON tf.flight_id = f.flight_id
  JOIN aircrafts a ON a.aircraft_code = r.aircraft_code
GROUP BY /*DISTINCT*/ GROUPING SETS
( GROUPING SETS ( ( da ), ( aa ) ),
  CUBE ( a.model, fc )
)
ORDER BY
  coalesce( r.departure_airport, r.arrival_airport ),
  r.arrival_airport NULLS FIRST, a.model, fc;
```

```
 da  | aa  |        model         | fc |  count  | t_amount
-----+-----+----------------------+----+---------+----------
 AAQ |     |                      |    |    9502 |   110.09
     | AAQ |                      |    |    9734 |   112.57
 ABA |     |                      |    |    1903 |    48.55
     | ABA |                      |    |    1901 |    49.39
 ...
 YKS |     |                      |    |    2584 |    58.29
     | YKS |                      |    |    2572 |    56.49
     |     | Аэробус A319-100     | B  |    9055 |  1028.20
     |     | Аэробус A319-100     | E  |   43798 |  1677.96
     |     | Аэробус A319-100     |    |   52853 |  2706.16
 ...
     |     |                      | B  |  107642 |  5505.18
     |     |                      | C  |   17291 |   566.12
     |     |                      | E  |  920793 | 14695.68
     |     |                      |    | 1045726 | 20766.98
(215 строк)
```

Обратите внимание на предложение ORDER BY. Использование функции coalesce позволяет расположить строки, относящиеся к одному аэропорту, парами: в первой строке он выступает в роли аэропорта отправления, а во второй — аэропорта прибытия.

В конструкции GROUPING SETS находятся две вложенные конструкции: GROUPING SETS и CUBE. В том случае, когда при формировании эквивалентной конструкции GROUPING SETS возможно появление одинаковых наборов столбцов, можно устранить дубликаты с помощью предложения DISTINCT. В запросе оно закомментировано, поскольку в нашем случае дубликатов нет.

В подразделе документации 7.2.4 «GROUPING SETS, CUBE и ROLLUP» сказано, что когда одна конструкция GROUPING SETS вложена в другую такую же, это равносильно тому, что элементы вложенной конструкции были бы записаны непосредственно во внешнюю конструкцию GROUPING SETS.

Задание 1. Поскольку без вложенной конструкции GROUPING SETS можно обойтись, исключите ее из предложения GROUP BY.

Задание 2. Напишите конструкцию GROUPING SETS, эквивалентную приведенной в запросе, но не имеющую вложенных конструкций CUBE и GROUPING SETS. Если нужно, обратитесь к полученной выборке, чтобы понять, какие комбинации группируемых столбцов были сформированы фактически.

Задание 3. Если непосредственно в предложении GROUP BY записано несколько группировочных конструкций, итоговый список множеств группируемых столбцов будет являться декартовым произведением множеств, порождаемых каждой такой конструкцией.

Давайте еще раз модифицируем запрос:

```
...
GROUP BY
  GROUPING SETS ( ( da ), ( aa ) ),
  CUBE( a.model, fc )
ORDER BY
  coalesce( r.departure_airport, r.arrival_airport ),
  r.arrival_airport NULLS FIRST, a.model, fc;
 da  | aa  |        model        | fc | count  | t_amount
-----+-----+---------------------+----+--------+----------
 AAQ |     | Боинг 737-300       | B  |    495 |    18.12
 AAQ |     | Боинг 737-300       | E  |   4759 |    58.35
 AAQ |     | Боинг 737-300       |    |   5254 |    76.47
 ...
 AAQ |     |                     | B  |   1029 |    28.21
 AAQ |     |                     | E  |   8473 |    81.88
 AAQ |     |                     |    |   9502 |   110.09
     | AAQ | Боинг 737-300       | B  |    505 |    18.48
     | AAQ | Боинг 737-300       | E  |   4867 |    59.68
     | AAQ | Боинг 737-300       |    |   5372 |    78.17
 ...
     | AAQ |                     | B  |   1044 |    28.67
     | AAQ |                     | E  |   8690 |    83.90
     | AAQ |                     |    |   9734 |   112.57
 ...
 YKS |     |                     | B  |    118 |    17.15
 ...
     | YKS |                     |    |   2572 |    56.49
(1639 строк)
```

Если в конструкцию GROUPING SETS добавить пустое множество группируемых столбцов, то в выборке появятся дополнительные строки:

```
...
GROUP BY
  GROUPING SETS( ( da ), ( aa ), ( ) ),
  CUBE( a.model, fc )
ORDER BY
  coalesce( r.departure_airport, r.arrival_airport ),
  r.arrival_airport NULLS FIRST, a.model, fc;
```

```
 ...
 YKS |     |                     | B  |     118 |    17.15
 ...
     | YKS |                     |    |    2572 |    56.49
     |     | Аэробус A319-100    | B  |    9055 |  1028.20
     |     | Аэробус A319-100    | E  |   43798 |  1677.96
     |     | Аэробус A319-100    |    |   52853 |  2706.16
 ...
     |     | Сухой Суперджет-100 | B  |   45415 |  1520.85
     |     | Сухой Суперджет-100 | E  |  320283 |  3593.63
     |     | Сухой Суперджет-100 |    |  365698 |  5114.48
     |     |                     | B  |  107642 |  5505.18
     |     |                     | C  |   17291 |   566.12
     |     |                     | E  |  920793 | 14695.68
     |     |                     |    | 1045726 | 20766.98
(1666 строк)
```

Попробуйте объяснить происхождение строк в выборке, полученных в результате модификации конструкции GROUPING SETS. Напишите конструкцию GROUPING SETS, эквивалентную использованной в запросе комбинации двух группировочных конструкций.

Задание 4. Предложите модификацию запроса с использованием конструкции ROLLUP. Например, замените конструкцию CUBE на ROLLUP, выполните запрос и объясните различия в получаемых выборках.

16 Оконная функция в предложении ORDER BY: можно ли обойтись без нее?

В документации сказано, что использование оконных функций допускается только в списке вывода команды SELECT и в предложении ORDER BY (см. разделы 4.2.8 «Вызовы оконных функций» и 3.5 «Оконные функции»). В качестве примера оконной функции в предложении ORDER BY покажем запрос, который выводит модели самолетов по убыванию числа кресел в салоне.

```
SELECT aircraft_code, model, range
FROM aircrafts AS a
WINDOW seats_win AS
( ORDER BY (
     SELECT count( * ) FROM seats AS s WHERE s.aircraft_code = a.aircraft_code
  ) DESC
)
ORDER BY rank() OVER ( seats_win );
```

```
 aircraft_code |        model        | range
---------------+---------------------+-------
 773           | Боинг 777-300       | 11100
 763           | Боинг 767-300       |  7900
 321           | Аэробус A321-200    |  5600
 320           | Аэробус A320-200    |  5700
 733           | Боинг 737-300       |  4200
 319           | Аэробус A319-100    |  6700
 SU9           | Сухой Суперджет-100 |  3000
 CR2           | Бомбардье CRJ-200   |  2700
 CN1           | Сессна 208 Караван  |  1200
(9 строк)
```

Недостатком этого запроса является отсутствие в выборке столбца, содержащего число кресел, то есть того показателя, по которому и производится ранжирование.

Можно обойтись без оконной функции, соединив таблицы «Самолеты» (aircrafts) и «Места» (seats):

```
SELECT a.aircraft_code, a.model, a.range, count( * ) AS seats_count
FROM aircrafts AS a
  JOIN seats AS s ON s.aircraft_code = a.aircraft_code
GROUP BY a.aircraft_code, a.model, a.range
ORDER BY count( * ) DESC;
```

```
 aircraft_code |        model        | range | seats_count
---------------+---------------------+-------+-------------
 773           | Боинг 777-300       | 11100 |         402
 763           | Боинг 767-300       |  7900 |         222
 321           | Аэробус A321-200    |  5600 |         170
 320           | Аэробус A320-200    |  5700 |         140
 733           | Боинг 737-300       |  4200 |         130
 319           | Аэробус A319-100    |  6700 |         116
 SU9           | Сухой Суперджет-100 |  3000 |          97
 CR2           | Бомбардье CRJ-200   |  2700 |          50
 CN1           | Сессна 208 Караван  |  1200 |          12
(9 строк)
```

Хотя оба запроса выполняются в пределах нескольких миллисекунд, второй запрос все же работает в несколько раз дольше первого.

Задание. При разработке эквивалентного запроса без оконной функции имеет смысл рассмотреть разные варианты. Реализуйте вариант запроса с предварительным подсчетом числа мест в каждой модели самолета с использованием

общего табличного выражения (конструкция WITH). Сравните быстродействие трех вариантов запроса и попытайтесь объяснить разницу, посмотрев планы их выполнения.

Вопрос. Как вы думаете, почему использовать оконные функции в предложениях WHERE и HAVING нельзя, а в предложении ORDER BY можно? Вспомните, на каком этапе выполнения запроса вычисляются оконные функции (см. раздел документации 3.5 «Оконные функции»).

17 В запросе может быть несколько разных определений окна

В документации сказано, что в запросе можно использовать более одной оконной функции (см. разделы документации 3.5 «Оконные функции» и 7.2.5 «Обработка оконных функций»). При этом функции могут работать с окнами, имеющими разные определения.

Если нам нужно показать в одной выборке упорядочение моделей самолетов как по дальности полета, так и по числу мест в салонах, а также разбить модели на группы в зависимости от числа этих мест, то одним из вариантов решения задачи может быть такой:

```
SELECT
  aircraft_code AS a_code,
  model,
  range,
  rank() OVER ( range_win ) AS r_rank,
  ( SELECT count( * ) FROM seats AS s WHERE s.aircraft_code = a.aircraft_code
  ) AS seats_num,
  rank() OVER ( seats_win ) AS s_rank,
  ntile( 4 ) OVER ( seats_win ) AS s_ntile
FROM aircrafts AS a
WINDOW range_win AS
( ORDER BY range DESC
),
seats_win AS
( ORDER BY (
     SELECT count( * ) FROM seats AS s WHERE s.aircraft_code = a.aircraft_code
  ) DESC
)
ORDER BY model;
```

```
 a_code |        model        | range | r_rank | seats_num | s_rank | s_ntile
--------+---------------------+-------+--------+-----------+--------+---------
 319    | Аэробус A319-100    |  6700 |      3 |       116 |      6 |       3
 320    | Аэробус A320-200    |  5700 |      4 |       140 |      4 |       2
 321    | Аэробус A321-200    |  5600 |      5 |       170 |      3 |       1
 733    | Боинг 737-300       |  4200 |      6 |       130 |      5 |       2
 763    | Боинг 767-300       |  7900 |      2 |       222 |      2 |       1
 773    | Боинг 777-300       | 11100 |      1 |       402 |      1 |       1
 CR2    | Бомбардье CRJ-200   |  2700 |      8 |        50 |      8 |       4
 CN1    | Сессна 208 Караван  |  1200 |      9 |        12 |      9 |       4
 SU9    | Сухой Суперджет-100 |  3000 |      7 |        97 |      7 |       3
(9 строк)
```

Обратите внимание, что подзапрос можно использовать в определении окна.

Недостатком приведенного запроса является повторение одного и того же подзапроса: он фигурирует не только в определении окна, но еще и в списке SELECT. Дублирования можно избежать, поместив подзапрос в общее табличное выражение:

```
WITH seats_counts AS
( SELECT
    aircraft_code AS a_code,
    model,
    range,
    ( SELECT count( * ) FROM seats AS s WHERE s.aircraft_code = a.aircraft_code
    ) AS seats_num
  FROM aircrafts AS a
)
SELECT
  a_code,
  model,
  range,
  rank() OVER ( range_win ) AS r_rank,
  seats_num,
  rank() OVER ( seats_win ) AS s_rank,
  ntile( 4 ) OVER ( seats_win ) AS s_ntile
FROM seats_counts
WINDOW range_win AS
( ORDER BY range DESC
),
seats_win AS
( ORDER BY seats_num DESC
)
ORDER BY model;
```

Таким образом, мы продемонстрировали, что одна и та же оконная функция (в нашем случае — rank) может использоваться с разными окнами, а одно и то же окно (в нашем случае — seats_win) — с разными функциями.

Вопрос. Как вы думаете, можно ли модифицировать первый вариант запроса таким образом: в определении окна seats_win вместо подзапроса воспользоваться псевдонимом столбца seats_num, определенным в списке SELECT? Ведь аналогичная конструкция в определении окна range_win с именем столбца range работает корректно:

```
...
WINDOW range_win AS
( ORDER BY range DESC
),
seats_win AS
( ORDER BY seats_num DESC
)
ORDER BY model;
```

Проверьте вашу гипотезу и объясните полученный результат.

Указание. За разъяснениями можно обратиться к подразделу документации 4.2.8 «Вызовы оконных функций» и подразделу «Предложение WINDOW» в описании команды SELECT.

Задание. Предложите еще одну ситуацию, в которой было бы логичным создать в запросе два окна (или более). Например, проранжируйте модели самолетов по числу выполненных ими рейсов и перевезенных пассажиров.

18 Использование условия в определении раздела

В тексте подраздела 3.4.3 «Совместное использование оконных и агрегатных функций» (с. 173) шла речь об оценке загрузки электронной системы бронирования билетов в выходные дни по сравнению с рабочими днями.

Средние значения, вычисленные для рабочих дней, выводились в столбце avg_5_days и для выходных дней (в строках, где в столбце dow стоят значения 6 и 7). Аналогично средние значения, вычисленные для выходных дней, выводились в столбце avg_7_days и для рабочих дней (в тех строках, где в столбце dow стоят значения от 1 до 5).

Запрос был таким:

```
SELECT
  to_char( date_trunc( 'day', b.book_date ), 'YYYY-MM-DD' ) AS b_date,
  -- b.book_date::date AS b_date,  -- можно сделать и так
  extract( week FROM b.book_date ) AS week,
  extract( isodow FROM b.book_date ) AS dow,
  count( * ) AS day_tf_count,
  round(
    avg( count( * ) )
      FILTER ( WHERE extract( isodow FROM book_date ) BETWEEN 1 AND 5 )
      OVER week_win,
    0
  ) AS avg_5_days,
  round(
    avg( count( * ) )
      FILTER ( WHERE extract( isodow FROM book_date ) IN ( 6, 7 ) )
      OVER week_win,
    0
  ) AS avg_67 -- выходные дни
FROM bookings b
  JOIN tickets t ON t.book_ref = b.book_ref
  JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
GROUP BY b_date, week, dow
WINDOW week_win AS ( PARTITION BY extract( week FROM b.book_date ) )
ORDER BY b_date;
```

```
   b_date   | week | dow | day_tf_count | avg_5_days | avg_67
------------+------+-----+--------------+------------+--------
 2017-06-21 |   25 |   3 |           20 |         41 |    494
 2017-06-22 |   25 |   4 |           22 |         41 |    494
 2017-06-23 |   25 |   5 |           80 |         41 |    494
 2017-06-24 |   25 |   6 |          284 |         41 |    494
 2017-06-25 |   25 |   7 |          704 |         41 |    494
 2017-06-26 |   26 |   1 |         1323 |       5187 |  14684
 2017-06-27 |   26 |   2 |         2639 |       5187 |  14684
 2017-06-28 |   26 |   3 |         4295 |       5187 |  14684
 2017-06-29 |   26 |   4 |         7408 |       5187 |  14684
 2017-06-30 |   26 |   5 |        10269 |       5187 |  14684
 ...
 2017-08-10 |   32 |   4 |        24656 |      23944 |  25666
 2017-08-11 |   32 |   5 |        25344 |      23944 |  25666
 2017-08-12 |   32 |   6 |        25109 |      23944 |  25666
 2017-08-13 |   32 |   7 |        26222 |      23944 |  25666
 2017-08-14 |   33 |   1 |        27459 |      26499 |
 2017-08-15 |   33 |   2 |        25538 |      26499 |
(56 строк)
```

Можно изменить эту ситуацию, добавив в предложение PARTITION BY, отвечающее за определение разделов, еще один вычисляемый столбец — проверку условия:

```
...
WINDOW week_win AS
( PARTITION BY
    extract( week FROM book_date ),
    extract( isodow FROM book_date ) IN ( 6, 7 )
)
...
   b_date   | week | dow | day_tf_count | avg_5_days | avg_67
------------+------+-----+--------------+------------+--------
 2017-06-21 |   25 |   3 |           20 |         41 |
 2017-06-22 |   25 |   4 |           22 |         41 |
 2017-06-23 |   25 |   5 |           80 |         41 |
 2017-06-24 |   25 |   6 |          284 |            |    494
 2017-06-25 |   25 |   7 |          704 |            |    494
 2017-06-26 |   26 |   1 |         1323 |       5187 |
 2017-06-27 |   26 |   2 |         2639 |       5187 |
 2017-06-28 |   26 |   3 |         4295 |       5187 |
 2017-06-29 |   26 |   4 |         7408 |       5187 |
 2017-06-30 |   26 |   5 |        10269 |       5187 |
 2017-07-01 |   26 |   6 |        13398 |            |  14684
 2017-07-02 |   26 |   7 |        15969 |            |  14684
 ...
(56 строк)
```

Задание 1. Попробуйте объяснить механизм получения такого результата.

Задание 2. Поскольку теперь разделы формируются с учетом вида дней недели, нельзя ли обойтись без использования предложений FILTER? Сначала сделайте обоснованное предположение, а затем проверьте его практически и объясните полученные результаты.

19 Эксперименты с определениями окон и оконными функциями

В тексте подраздела 3.4.2 «Способы формирования оконного кадра» (с. 164) шла речь об использовании так называемого скользящего среднего значения для анализа поступления денег за счет продажи авиабилетов. При использовании этого подхода важно правильно выбрать длину интервала сглаживания.

Давайте добавим в запрос, рассмотренный в тексте главы, обработку еще одного интервала сглаживания длительностью семь дней. Это позволит нам сравнить результаты для пятидневного и семидневного интервалов. Теперь в запросе будет два определения окон в предложении WINDOW, соответственно увеличится и число вызовов оконных функций. Для упрощения запроса перенесем вычисление оконных функций в общее табличное выражение. При этом время выполнения останется примерно таким же.

```
WITH day_amounts ( b_date, day_sum ) AS
( SELECT date_trunc( 'day', book_date ),
         round( sum( total_amount ) / 1000000, 2 )
  FROM bookings
  GROUP BY 1
),
win_funcs AS
( SELECT
    to_char( b_date, 'YYYY-MM-DD' ) AS date,
    day_sum,
    avg( day_sum ) OVER moving_win_5 AS mv_avg_5,
    avg( day_sum ) OVER moving_win_7 AS mv_avg_7
  FROM day_amounts
  WINDOW moving_win_5 AS
  ( ORDER BY b_date ROWS BETWEEN 4 PRECEDING AND CURRENT ROW
  ),
  moving_win_7 AS
  ( ORDER BY b_date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
  )
)
SELECT
  date,
  day_sum,
  round( mv_avg_5, 2 ) AS mv_avg_5,
  round( day_sum - mv_avg_5, 2 ) AS delta_5,
  round( ( day_sum - mv_avg_5 ) / day_sum * 100, 2 )
  AS percent_5,
  round( mv_avg_7, 2 ) AS mv_avg_7,
  round( day_sum - mv_avg_7, 2 ) AS delta_7,
  round( ( day_sum - mv_avg_7 ) / day_sum * 100, 2 ) AS percent_7
FROM win_funcs
ORDER BY date;
```

```
    date    | day_sum | mv_avg_5 | delta_5 | percent_5 | mv_avg_7 | delta_7 | percent_7
------------+---------+----------+---------+-----------+----------+---------+-----------
 2017-06-21 |    0.44 |     0.44 |    0.00 |      0.00 |     0.44 |    0.00 |      0.00
 2017-06-22 |    0.56 |     0.50 |    0.06 |     10.71 |     0.50 |    0.06 |     10.71
 ...
```

```
 2017-08-03 |  430.93 |   438.67 |   -7.74 |     -1.80 |   437.93 |   -7.00 |     -1.62
 2017-08-04 |  451.32 |   443.46 |    7.86 |      1.74 |   438.56 |   12.76 |      2.83
 2017-08-05 |  452.65 |   445.16 |    7.49 |      1.66 |   442.47 |   10.18 |      2.25
 2017-08-06 |  453.36 |   446.92 |    6.44 |      1.42 |   446.19 |    7.17 |      1.58
 2017-08-07 |  447.97 |   447.25 |    0.72 |      0.16 |   446.73 |    1.24 |      0.28
 2017-08-08 |  471.32 |   455.32 |   16.00 |      3.39 |   450.56 |   20.76 |      4.41
 2017-08-09 |  471.70 |   459.40 |   12.30 |      2.61 |   454.18 |   17.52 |      3.71
 2017-08-10 |  487.51 |   466.37 |   21.14 |      4.34 |   462.26 |   25.25 |      5.18
 2017-08-11 |  497.21 |   475.14 |   22.07 |      4.44 |   468.82 |   28.39 |      5.71
 2017-08-12 |  480.76 |   481.70 |   -0.94 |     -0.20 |   472.83 |    7.93 |      1.65
 2017-08-13 |  502.46 |   487.93 |   14.53 |      2.89 |   479.85 |   22.61 |      4.50
 2017-08-14 |  528.43 |   499.27 |   29.16 |      5.52 |   491.34 |   37.09 |      7.02
 2017-08-15 |  483.56 |   498.48 |  -14.92 |     -3.09 |   493.09 |   -9.53 |     -1.97
(56 строк)
```

В данном случае нас интересует не столько то, какой из двух интервалов предпочтительнее, сколько порядок выполнения оконных функций в этом запросе. Команда EXPLAIN с параметром ANALYZE покажет примерно такую картину:

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 ...
   -> Subquery Scan on win_funcs
       (cost=27048.43..33140.56 rows=56672 width=256)
       (actual time=118.910..119.228 rows=56 loops=1)
       -> WindowAgg
           (cost=27048.43..30590.32 rows=56672 width=136)
           (actual time=118.905..119.137 rows=56 loops=1)
           -> WindowAgg
               (cost=27048.37..29598.56 rows=56672 width=72)
               (actual time=118.889..119.055 rows=56 loops=1)
 ...
```

Задание 1. Попробуйте выяснить, какому из вызовов оконной функции соответствует каждый из двух узлов плана WindowAgg? Обратите внимание на значения width и на тот факт, что временные интервалы actual time перекрываются. Как вы думаете, может ли эта информация помочь в решении задачи?

Указание. Для того чтобы планы были более компактными, добавьте в общее табличное выражение запроса параметр MATERIALIZED. Подключите дополнительно параметр VERBOSE в команде EXPLAIN и проведите ряд экспериментов, обращая внимание на число узлов WindowAgg, а также на элементы плана с пометкой Output.

Например:

- добавьте еще один вызов оконной функции avg и окно, скажем, для шестидневного интервала сглаживания;
- замените одну или две оконные функции avg на другие функции, например min или max, так чтобы все оконные функции были различными;
- задайте определения окон для всех оконных функций совершенно одинаковыми, например:

```
...
WINDOW moving_win_5 AS
  ( ORDER BY b_date ROWS BETWEEN 4 PRECEDING AND CURRENT ROW
  ),
  moving_win_7 AS
  ( ORDER BY b_date ROWS BETWEEN 4 PRECEDING AND CURRENT ROW
  )
...
```

- сделайте все определения окон различными;
- добавьте еще одно определение окна для вызова функции, скажем count, и посмотрите, в каких узлах WindowAgg окажется ее вызов:

```
...
count( * ) OVER moving_win_10, 2 AS count_10
...
  moving_win_10 AS
  ( ORDER BY b_date ROWS BETWEEN 9 PRECEDING AND CURRENT ROW
  )
...
```

Задание 2. В подразделе документации 4.2.8 «Вызовы оконных функций» говорится, что можно с помощью предложения WINDOW задать определение окна, а затем при вызове оконной функции воспользоваться им, внеся необходимые изменения. Это может быть удобным, когда в запросе присутствует более одной оконной функции. Тогда можно общую часть определений нескольких окон поместить в предложение WINDOW, а при вызове оконной функции дописать в предложение OVER специфическую часть определения конкретного окна. Однако при этом нужно учитывать некоторые важные ограничения, перечисленные в упомянутом подразделе документации.

Модифицируйте запрос, воспользовавшись представленной возможностью. Вместо символов многоточия напишите необходимый код.

Обратите внимание, что в данном случае в предложении OVER требуются скобки, но когда мы использовали готовое определение окна без модификации, скобки не были нужны:

```
...
win_funcs AS
( SELECT
    to_char( b_date, 'YYYY-MM-DD' ) AS date,
    day_sum,
    avg( day_sum ) OVER ( base_win ... ) AS move_avg_5,
    avg( day_sum ) OVER ( base_win ... ) AS move_avg_7
  FROM day_amounts
  WINDOW base_win AS ( ORDER BY b_date )
)
...
```

20 Функции lead и lag

В пункте «Функции lead и lag» подраздела 3.4.4 «Оконные функции общего назначения» (с. 180) были рассмотрены функции lead и lag. В запросе, который был приведен для их иллюстрации, в определении окна отсутствовало предложение PARTITION BY, поэтому в качестве раздела выступала вся выборка. Характерной чертой полученной выборки было наличие строк, в которых в столбцах week_ago и delta оказались значения NULL.

Задание 1. Модифицируйте запрос, добавив в определение окна предложение PARTITION BY таким образом, чтобы в каждом разделе были собраны строки, соответствующие операциям только одного месяца. До выполнения запроса предположите, появятся ли в таком случае в выборке строки со значениями NULL в столбцах week_ago и delta. Обоснуйте ваши предположения.

Задание 2. Предложите другие ситуации, в которых использование функций lead и lag помогло бы решить задачу, и напишите запросы. В качестве примера можно обратиться к подсчету количества оформленных перелетов и сопоставлению результатов, полученных в текущий день, с днем, скажем, неделей ранее.

Задание 3. Попробуйте предложить такую ситуацию, в которой эти функции не были бы взаимозаменяемыми. Например, можно ли в следующем запросе заменить функцию lag на функцию lead, возможно, изменив порядок сортировки в определении окна?

```
SELECT
  aircraft_code,
  CASE
    WHEN substring( seat_no FROM '^\d+' ) =
         lag( substring( seat_no FROM '^\d+' ), 1 ) OVER all_rows_win
      THEN NULL
    ELSE substring( seat_no FROM '^\d+' )
  END AS row_num,
  seat_no,
  fare_conditions
FROM seats
WINDOW all_rows_win AS
( PARTITION BY aircraft_code
  ORDER BY substring( seat_no FROM '^\d+' )::smallint, right( seat_no, 1 )
)
ORDER BY
  aircraft_code,
  substring( seat_no FROM '^\d+' )::smallint,
  right( seat_no, 1 );
```

```
 aircraft_code | row_num | seat_no | fare_conditions
---------------+---------+---------+-----------------
 319           | 1       | 1A      | Business
 319           |         | 1C      | Business
 319           |         | 1D      | Business
 319           |         | 1F      | Business
 319           | 2       | 2A      | Business
 319           |         | 2C      | Business
 319           |         | 2D      | Business
 319           |         | 2F      | Business
 319           | 3       | 3A      | Business
 319           |         | 3C      | Business
 319           |         | 3D      | Business
 319           |         | 3F      | Business
 319           | 4       | 4A      | Business
 319           |         | 4C      | Business
 319           |         | 4D      | Business
 319           |         | 4F      | Business
 319           | 5       | 5A      | Business
 ...
(1339 строк)
```

21 Оконные функции и родственные строки

В пункте «Остальные оконные функции» подраздела 3.4.4 «Оконные функции общего назначения» (с. 183) мы в одном запросе показали использование сразу нескольких оконных функций: `row_number`, `rank`, `dense_rank` и других. Для иллюстрации различий между функциями `rank` и `dense_rank` мы добавляли в выборку из таблицы «Самолеты» (`aircrafts`) еще одну модель самолета — «Туполев Ту-204», при этом дальность полета этой модели назначали такой же, как и у модели «Аэробус A319-100».

Задание. Добавьте в выборку из таблицы «Самолеты» (`aircrafts`) не только модель «Туполев Ту-204», но и еще одну (возможно, вымышленную) модель. У нее должна быть такая же дальность полета, как у модели «Сессна 208 Караван», чтобы группа родственных (peer) строк образовалась не только в середине, но и в конце выборки. Сделайте обоснованное предположение о том, какие значения будут теперь в столбцах выборки. В частности, будут ли равны единице значения в последней строке в столбцах `pr` и `cd`, которые формируются функциями `percent_rank` и `cume_dist`? Может ли функция `cume_dist` возвратить значение 1,0 не только для последней строки выборки? Почему? Проверьте ваши гипотезы, выполнив запрос.

22 Сравнение режимов формирования оконного кадра RANGE, ROWS и GROUPS

В подразделе 3.4.2 «Способы формирования оконного кадра» (с. 164) представлены режимы его формирования. При прочих равных условиях выбор того или иного режима может повлиять на полученный результат.

Давайте проведем ряд экспериментов. Для этого создадим таблицу и заполним ее данными. Обратите внимание, что даты повторяются (есть родственные строки) и в них есть пропуски.

```
CREATE TABLE dates ( dt date );
CREATE TABLE
COPY dates FROM STDIN;
Вводите данные для копирования, разделяя строки переводом строки.
Закончите ввод строкой '\.' или сигналом EOF.
```

```
>> 2017-07-01
2017-07-01
2017-07-01
2017-07-03
2017-07-03
2017-07-05
2017-07-05
2017-07-06
2017-07-06
2017-07-09
2017-07-09
2017-07-09
2017-07-10
\.
COPY 13
```

Нам потребуется такой запрос:

```
SELECT dt, count( * ) OVER dates_win
FROM dates
WINDOW dates_win AS
( ORDER BY dt
--  RANGE BETWEEN interval '2 days' PRECEDING AND CURRENT ROW
--  RANGE BETWEEN CURRENT ROW AND interval '2 days' FOLLOWING
--  ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
--  ROWS BETWEEN CURRENT ROW AND 2 FOLLOWING
--  GROUPS BETWEEN 2 PRECEDING AND CURRENT ROW
--  GROUPS BETWEEN CURRENT ROW AND 2 FOLLOWING
)
ORDER BY dt;
```

Задание. Проведите эксперименты, выбирая конкретный режим формирования оконного кадра и изменяя значение смещения, как предлагается в закомментированных фрагментах запроса. Проанализируйте значения в столбце count для каждой строки выборки.

Глава 4
Конструкция LATERAL команды SELECT

Конструкция LATERAL значительно расширяет возможности команды SELECT. Она позволяет организовать обработку подзапроса в предложении FROM таким образом, как будто она производится в цикле. С помощью конструкции LATERAL в ряде случаев удается значительно упростить запрос, облегчить его интерпретацию. Бывают ситуации, когда без этой конструкции трудно обойтись.

4.1. Подзапросы в предложении FROM

Предположим, что отдел маркетинга нашей авиакомпании с целью привлечения пассажиров на рейсы, выполняемые между Москвой и регионами Дальнего Востока, решил поощрять каждого десятого пассажира, например предоставляя скидку на последующие полеты. В этой акции принимают участие только пассажиры уже выполненных рейсов. Такие счастливчики должны определяться случайным образом. Очевидно, что их число на каждом рейсе в общем случае будет различаться. При этом было решено, что даже если число пассажиров на рейсе окажется меньше десяти, все равно хотя бы один из них должен получить поощрение. Каким образом можно произвести такой отбор?

Алгоритм решения задачи может быть таким: для каждого рейса, имеющего статус Arrived, нужно случайным образом выбрать из таблицы «Посадочные талоны» (boarding_passes) десятую часть строк, соответствующих этому рейсу, а затем объединить полученные множества строк в единую выборку. Конечно, реализовать этот алгоритм можно, используя процедурное расширение PostgreSQL — язык PL/pgSQL, — но можно обойтись и одной командой SELECT. Поможет в этом конструкция LATERAL, представленная в подразделе документации 7.2.1.5 «Подзапросы LATERAL» и в разделе «Предложение FROM» описания команды SELECT.

Чтобы не загромождать запрос, предварительно получим коды аэропортов:

```
SELECT city, airport_code
FROM airports
WHERE city IN ('Москва', 'Владивосток', 'Хабаровск', 'Южно-Сахалинск', 'Благовещенск')
ORDER BY city;
```

```
      city      | airport_code
----------------+--------------
 Благовещенск   | BQS
 Владивосток    | VVO
 Москва         | SVO
 Москва         | VKO
 Москва         | DME
 Хабаровск      | KHV
 Южно-Сахалинск | UUS
(7 строк)
```

Теперь приведем запрос и результат его работы, а затем дадим пояснения.

```
SELECT f.flight_id, bp2.ticket_no
FROM flights f
  CROSS JOIN LATERAL
    ( SELECT ticket_no
      FROM boarding_passes bp
      WHERE bp.flight_id = f.flight_id
      ORDER BY random()
      LIMIT ceiling(
              ( SELECT count( * )
                FROM boarding_passes bp
                WHERE bp.flight_id = f.flight_id
              ) * 0.1
            )
    ) AS bp2
WHERE f.status = 'Arrived'
  AND ( ( f.departure_airport IN ( 'DME', 'SVO', 'VKO' )
          AND
          f.arrival_airport IN ( 'VVO', 'KHV', 'UUS', 'BOS' )
        )
        OR
        ( f.departure_airport IN ( 'VVO', 'KHV', 'UUS', 'BOS' )
          AND
          f.arrival_airport IN ( 'DME', 'SVO', 'VKO' )
        )
      )
ORDER BY f.flight_id, bp2.ticket_no;
```

Здесь мы использовали таблицу «Посадочные талоны» (boarding_passes) для определения пассажиров, фактически вылетевших конкретными рейсами. В нашей базе данных нет ни одного случая, когда пассажир приобрел билет, но не явился на регистрацию, поэтому мы могли бы воспользоваться и таблицей «Перелеты» (ticket_flights), но такое решение было бы менее строгим.

```
 flight_id |   ticket_no
-----------+---------------
      1249 | 0005433415429
      1249 | 0005435824343
      1249 | 0005435824347
      1249 | 0005435824354
      1249 | 0005435824367
      1250 | 0005435824712
      1250 | 0005435824713
      1250 | 0005435824714
      1250 | 0005435824724
  ...
     26616 | 0005432571283
     26616 | 0005435947094
     26616 | 0005435947099
(1858 строк)
```

Известно, что в предложении FROM команды SELECT могут использоваться подзапросы. По умолчанию (если не принять специальных мер) все они обрабатываются независимо друг от друга и не могут ссылаться на другие элементы списка FROM. При этом порядок следования элементов в данном списке не влияет на порядок выполнения соединений, который выберет планировщик. Однако можно сделать так, чтобы подзапрос мог ссылаться на столбцы таблиц или подзапросов, находящихся *перед* ним в списке FROM. Для этого необходимо дополнительно поместить ключевое слово LATERAL *перед* подзапросом, который должен иметь такую возможность. Наличие этого ключевого слова вынуждает планировщик сначала выполнить соединение всех элементов списка FROM, находящихся левее этого слова.

При наличии ключевого слова LATERAL запрос выполняется следующим образом. Если слева от слова LATERAL стоит только одна таблица (или подзапрос), то для ее *текущей строки* выполняется подзапрос, стоящий справа. Если в нем есть ссылки на столбцы левой таблицы, то их значения берутся из ее текущей строки. Такие ссылки могут быть, например, в предложениях WHERE или LIMIT.

Затем строки, порожденные подзапросом, соединяются с текущей строкой левой таблицы в соответствии с предписанием JOIN. Эта процедура повторяется для *всех строк* левой таблицы. Все сформированные подмножества строк объединяются в единое множество. В том случае, когда слева от ключевого слова LATERAL стоят несколько таблиц (или подзапросов), текущая строка берется из *результата их соединения*.

Приведенное описание говорит о том, что обработка конструкции LATERAL представляет собой параметризованный цикл.

Проецируя это правило на приведенный запрос, получим такую схему. Берется строка из таблицы «Рейсы» (flights), удовлетворяющая условию WHERE главного запроса, то есть конкретный рейс, прибывший из Москвы в один из дальневосточных регионов или обратно, и выполняется подзапрос bp2 с условием WHERE bp.flight_id = f.flight_id. Таким образом, из таблицы «Посадочные талоны» (boarding_passes) выбираются только номера билетов, оформленных на данный рейс. В результирующий набор попадает лишь случайная выборка из этих строк. После этого выполняется декартово произведение (CROSS JOIN) текущей строки из таблицы «Рейсы» (flights) на все строки, которые возвратил подзапрос bp2.

Затем берется следующая строка из таблицы «Рейсы» (flights), удовлетворяющая условию WHERE главного запроса, для нее выполняется подзапрос bp2, и эта строка соединяется со строками, которые возвратил подзапрос. Этот процесс повторяется до исчерпания строк таблицы «Рейсы» (flights). Все сформированные наборы строк объединяются в единую выборку, которую мы и получаем.

Обратите внимание, что в предложении LIMIT тоже находится подзапрос. В нем определяется общее число пассажиров конкретного рейса, для которого производятся вычисления. А от этого числа берется десятая часть, как и было решено на этапе постановки задачи. Здесь используется функция ceiling, которая возвращает наименьшее целое число, большее или равное ее аргументу. Это позволяет выбрать пассажира для поощрения даже на тех рейсах, которыми летели менее десяти человек.

Случайная выборка строк из таблицы «Посадочные талоны» (boarding_passes) достигается сортировкой на основе значений функции random, вычисленных для каждой строки. Из документации известно, что элементами предложения

ORDER BY могут быть и произвольные выражения, вычисленные на основе значений исходных строк (см. описание команды SELECT). Поэтому использование функции random в данном случае вполне правомерно.

Напомним также, что вместо выражения CROSS JOIN можно было просто поставить запятую.

План запроса будет таким:

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Incremental Sort
   Sort Key: f.flight_id, bp.ticket_no
   Presorted Key: f.flight_id
   -> Nested Loop
       -> Index Scan using flights_pkey on flights f
           Filter: (((status)::text = 'Arrived'::text) AND (((departure_airport = ANY...
       -> Limit
           InitPlan 1
             -> Aggregate
                 -> Index Only Scan using boarding_passes_flight_id_seat_no_key on bo...
                     Index Cond: (flight_id = f.flight_id)
           -> Sort
               Sort Key: (random())
               -> Bitmap Heap Scan on boarding_passes bp
                   Recheck Cond: (flight_id = f.flight_id)
                   -> Bitmap Index Scan on boarding_passes_flight_id_seat_no_key
                       Index Cond: (flight_id = f.flight_id)
(17 строк)
```

Обратите внимание, что для соединения строк используется метод вложенного цикла. Это вполне объяснимо. Метод хеширования здесь неприменим, поскольку он последовательно читает сначала один набор строк (строя на его основе хеш-таблицу), а затем второй (сопоставляя прочитанные строки с готовой хеш-таблицей). Метод слияния также неприменим, поскольку перед слиянием оба набора строк должны быть полностью отсортированы. И только метод вложенного цикла просматривает строки итеративно, позволяя формировать один из наборов строк частями, что и подразумевается конструкцией LATERAL.

В узле Nested Loop для каждой строки, выбранной из таблицы «Рейсы» (flights), выполняется весь набор операций, представленных в узле Limit и его подузлах. Условие flight_id = f.flight_id в узле Index Only Scan говорит о том, что из таблицы «Посадочные талоны» (boarding_passes) выбираются строки, в которых

значение поля flight_id совпадает со значением f.flight_id в текущей строке таблицы «Рейсы» (flights).

Если в команду EXPLAIN добавить параметр ANALYZE, то запрос будет выполнен и в плане появятся фактические значения показателей rows и loops. Предлагаем читателю самостоятельно проинтерпретировать эти значения, учитывая, что мы отбираем десятую часть строк, полученных из таблицы «Посадочные талоны» (boarding_passes).

Для именования подзапроса мы использовали псевдоним bp2 только для наглядности, чтобы показать, что таблица «Посадочные талоны» (boarding_passes) внутри подзапроса — это одна сущность, а результат выполнения самого подзапроса — другая. Поэтому нельзя, используя псевдоним bp2 в списке SELECT главного запроса, обратиться к тем столбцам таблицы boarding_passes, которых нет в списке SELECT в подзапросе. Например, если бы мы захотели выводить еще и номера мест в салонах самолетов, но добавили столбец seat_no только в главный запрос, не добавив его в список SELECT подзапроса, то получили бы сообщение об ошибке:

```
SELECT f.flight_id, bp2.ticket_no, bp2.seat_no
FROM flights f
  CROSS JOIN LATERAL
    ( SELECT ticket_no
      FROM boarding_passes bp
...
    ) AS bp2
...
ОШИБКА:  столбец bp2.seat_no не существует
СТРОКА 1: SELECT f.flight_id, bp2.ticket_no, bp2.seat_no
                                             ^
```

Точно так же, если изменить псевдоним таблицы для столбца ticket_no в списке SELECT главного запроса с bp2 на bp, не изменив его для самого подзапроса, снова получим сообщение об ошибке:

```
SELECT f.flight_id, bp.ticket_no
FROM flights f
  CROSS JOIN LATERAL
    ( SELECT ticket_no
      FROM boarding_passes bp
...
    ) AS bp2   -- здесь не изменили на bp!
...
```

```
ОШИБКА:  таблица "bp" отсутствует в предложении FROM
СТРОКА 1: SELECT f.flight_id, bp.ticket_no
                              ^
```

В сообщении имеется в виду предложение FROM, ближайшее в смысле уровней вложенности к списку SELECT, но в этом предложении есть только псевдоним bp2, а bp нет.

Поэтому вполне можно давать и таблице, обрабатываемой в подзапросе, и самому подзапросу один и тот же псевдоним, в данном случае — bp, а PostgreSQL разберется. В дальнейшем мы такой подход тоже будем использовать.

Давайте для каждого рейса подсчитаем число пассажиров, получивших «счастливый» билет, и общее число пассажиров. Для этого можно модифицировать и дополнить рассматриваемый запрос.

Поскольку условие отбора дальневосточных рейсов будет использоваться дважды, представим его в виде общего табличного выражения far_east_flights. В общем табличном выражении happy_passengers откажемся от коррелированного подзапроса в предложении LIMIT, а вместо этого поместим в предложение FROM еще один подзапрос. В нем будет выполняться подсчет числа поощряемых пассажиров сразу для всех рейсов, а не для каждого рейса в отдельности, как было прежде. Приведет ли такая модификация к ускорению запроса, предлагаем читателю проверить самостоятельно. Для наглядности в запросе оставлен закомментированный вариант общего табличного выражения happy_passengers.

```
WITH far_east_flights AS
( SELECT flight_id
  FROM flights
  WHERE status = 'Arrived'
    AND ( ( departure_airport IN ( 'DME', 'SVO', 'VKO' )
            AND
            arrival_airport IN ( 'VVO', 'KHV', 'UUS', 'BOS' )
          )
          OR
          ( departure_airport IN ( 'VVO', 'KHV', 'UUS', 'BOS' )
            AND
            arrival_airport IN ( 'DME', 'SVO', 'VKO' )
          )
        )
),
```

```
all_passengers AS
( SELECT f.flight_id, count( * ) AS all_pass
  FROM boarding_passes AS bp
    JOIN far_east_flights AS f ON f.flight_id = bp.flight_id
  GROUP BY f.flight_id
),
/*
happy_passengers AS
( SELECT f.flight_id, bp2.ticket_no
  FROM far_east_flights AS f
    CROSS JOIN LATERAL
      ( SELECT ticket_no FROM boarding_passes bp
        WHERE bp.flight_id = f.flight_id
        ORDER BY random()
        LIMIT ceiling(
                ( SELECT count(*)
                  FROM boarding_passes bp
                  WHERE bp.flight_id = f.flight_id
                ) * 0.1
              )
      ) AS bp2
), */
happy_passengers AS
( SELECT cnt.flight_id, bp2.ticket_no
  FROM ( SELECT f.flight_id, ceiling( count( * ) * 0.1 ) AS happy_pass_count
         FROM far_east_flights AS f
           JOIN boarding_passes AS bp ON bp.flight_id = f.flight_id
         GROUP BY f.flight_id
       ) AS cnt
    CROSS JOIN LATERAL
       ( SELECT ticket_no
         FROM boarding_passes bp
         WHERE bp.flight_id = cnt.flight_id
         ORDER BY random()
         LIMIT cnt.happy_pass_count
       ) AS bp2
),
happy_passengers_2 AS
( SELECT flight_id, count( * ) AS happy_pass
  FROM happy_passengers
  GROUP BY flight_id
)
SELECT hp.flight_id, all_pass, happy_pass
FROM happy_passengers_2 AS hp
  JOIN all_passengers AS ap ON ap.flight_id = hp.flight_id
ORDER BY all_pass DESC, flight_id;
```

```
 flight_id | all_pass | happy_pass
-----------+----------+------------
      2354 |      218 |         22
      2329 |      216 |         22
 ...
     26192 |      120 |         12
     26207 |      115 |         12
     26183 |       89 |          9
     26197 |       85 |          9
 ...
     17990 |       15 |          2
      4204 |       12 |          2
      4199 |        9 |          1
      4201 |        6 |          1
      4255 |        5 |          1
      4241 |        3 |          1
(240 строк)
```

Надо добавить, что при выполнении этого — контрольного — запроса в общем случае будут отобраны другие номера билетов, а не те же самые, которые были отобраны при выполнении первого запроса. Нас это устраивает, поскольку цель проверки — подсчитать *количество* «счастливых» билетов, а не получить повторно их номера. Если бы требовалось провести проверку именно на той выборке, которая была получена в первом запросе, можно было бы сохранить ее результаты в таблице, а затем проводить дальнейшие эксперименты уже с ней.

Обратите внимание, что на некоторых рейсах оказалось всего по одному пассажиру, получившему «счастливый» билет. Посмотрев на общее число пассажиров на этих рейсах, мы увидим, что принцип «с каждого рейса должен быть отобран хотя бы один человек» выполняется.

Вернемся к обсуждению конструкции LATERAL. Поскольку она предполагает использование метода вложенного цикла для соединения наборов строк, то на больших выборках время выполнения таких запросов может быть значительным. Вообще, при увеличении размера выборки в принципе может быть выбран другой план выполнения запроса, в результате *относительное* ускорение модифицированного запроса, достигнутое на небольших выборках, не будет иметь места на большой выборке. В упражнении 1 (с. 244) предлагается исследовать поведение запросов, рассмотренных в этом разделе, на выборках различных размеров.

4.2. Вызовы функций в предложении FROM

В конструкции LATERAL можно использовать не только подзапросы, но и *вызовы функций*, возвращающих множества строк. Покажем это на примере.

Согласованность данных является важным критерием их качества и пригодности для принятия управленческих решений. Предположим, что специалисты нашей авиакомпании периодически анализируют согласованность данных в базе «Авиаперевозки», в частности отсутствие пропусков в нумерации посадочных талонов. Эти талоны получают последовательные порядковые номера, начиная с единицы, в рамках каждого рейса. Каким образом можно выявить возможные пропуски?

Идея алгоритма такова. Для каждого состоявшегося рейса (статус Departed или Arrived) нужно сформировать «эталонный» список номеров посадочных талонов без пропусков, начиная с единицы и заканчивая максимальным номером оформленного на этот рейс талона. Затем, используя, например, внешнее соединение, нужно сопоставить этот полный список номеров талонов со списком номеров талонов, фактически оформленных на этот рейс. Те номера из полного списка, для которых не найдется пары в списке фактически оформленных талонов, и будут являться пропущенными номерами.

Теперь приведем запрос.

```
SELECT f.flight_id, nums.boarding_no
FROM flights f
  CROSS JOIN LATERAL generate_series(
      1, ( SELECT max( boarding_no )
           FROM boarding_passes AS bp
           WHERE bp.flight_id = f.flight_id
         )
    ) AS nums( boarding_no )
  LEFT OUTER JOIN boarding_passes AS bp
    ON bp.flight_id = f.flight_id AND bp.boarding_no = nums.boarding_no
WHERE f.status IN ( 'Departed', 'Arrived' )
  AND bp.boarding_no IS NULL
ORDER BY f.flight_id, nums.boarding_no;
```

Запрос работает следующим образом. Для каждой строки из таблицы «Рейсы» (flights) вызывается функция generate_series. В качестве ее аргумента указан коррелированный подзапрос, определяющий максимальное значение номера

посадочного талона для данного рейса. Число строк, формируемых функцией, будет равно этому значению. Мы используем соединение CROSS JOIN и получаем для каждого рейса столько комбинированных строк, сколько номеров было сгенерировано функцией generate_series. Здесь присутствует ключевое слово LATERAL, однако для функций его можно опускать. В наших примерах мы приводим это ключевое слово только для ясности.

Затем выполняется соединение LEFT OUTER JOIN. Каждая комбинированная строка, полученная на предыдущем этапе, сопоставляется с таблицей «Посадочные талоны» (boarding_passes). Цель — найти для данного рейса строку с номером посадочного талона, равным «эталонному» номеру. Если нужной строки в таблице «Посадочные талоны» (boarding_passes) не нашлось, то благодаря левому внешнему соединению комбинированная строка все равно сформируется, но в ней в качестве значения столбца bp.boarding_no будет стоять NULL.

Наконец, в предложении WHERE главного запроса отбираются лишь те комбинированные строки, в которых в столбце bp.boarding_no стоит NULL.

Этот запрос не выявляет ни одного пропуска в нумерации посадочных талонов. Для того чтобы убедиться в его работоспособности, давайте удалим несколько строк из таблицы «Посадочные талоны» (boarding_passes), а чтобы база данных в итоге не пришла в рассогласованное состояние, воспользуемся транзакцией, которую затем отменим.

```
BEGIN;
BEGIN
DELETE FROM boarding_passes
WHERE ( flight_id, boarding_no ) IN ( ( 25, 3 ), ( 25, 6 ), ( 26, 3 ), ( 27, 5 ) );
DELETE 4
SELECT f.flight_id, nums.boarding_no
FROM flights f
  CROSS JOIN LATERAL
...
 flight_id | boarding_no
-----------+-------------
        25 |           3
        25 |           6
        26 |           3
        27 |           5
(4 строки)
```

Да, запрос работает корректно.

```
ROLLBACK;
ROLLBACK
```

Посмотрев план выполнения этого запроса, увидим, что соединение CROSS JOIN LATERAL выполняется методом вложенного цикла. Для получения более полной картины самостоятельно выполните запрос с помощью команды EXPLAIN с параметром ANALYZE. Обратите внимание на значения rows и loops.

Таким образом, мы показали, что применение конструкции LATERAL полезно, когда нужно передать значение параметра функции, возвращающей набор строк. В рассмотренном примере это была функция generate_series. В упражнении 5 (с. 250) она используется для выявления пропусков в нумерации.

4.3. Тип JSON и конструкция LATERAL

Могут возникать ситуации, когда на основе структурированного описания данных требуется сформировать строки для вставки в таблицу. В таких случаях использование в конструкции LATERAL функций, возвращающих множества строк, также поможет решить задачу.

В качестве иллюстрации рассмотрим следующую ситуацию. Предположим, что мы только приступаем к работе с базой данных «Авиаперевозки», и для начала требуется заполнить таблицу «Места» (seats). В этой таблице три столбца: aircraft_code, seat_no и fare_conditions. Конфигурации планировок салонов представлены в виде JSON-массива. Элементами массива являются JSON-объекты, представляющие класс обслуживания. Объект включает название класса обслуживания и два массива: один содержит номера первого и последнего рядов в этом классе, а другой — буквенные обозначения мест в этих рядах. Для примера возьмем только две модели самолетов.

Надо сказать, что не всегда салоны самолетов имеют регулярную структуру, как в нашем примере. Для планировок, в которых количество кресел одного класса обслуживания отличается от ряда к ряду, можно добавить в массив несколько элементов для одного класса.

Вначале для наглядности приведем упрощенный запрос и результат его выполнения.

```
WITH seats_confs( aircraft_code, seats_conf ) AS
( VALUES
  ( 'SU9',
    '[ { "fare conditions": "Business",
         "rows": [ 1, 3 ],
         "letters": [ "A", "C", "D", "F" ] },
       { "fare conditions": "Economy",
         "rows": [ 4, 20 ],
         "letters": [ "A", "C", "D", "E", "F" ] }
     ]'::jsonb
  ),
  ( 'CN1',
    '[ { "fare conditions": "Economy",
         "rows": [ 1, 6 ],
         "letters": [ "A", "B" ] }
     ]'::jsonb
  )
),
fare_cond_confs AS
( SELECT aircraft_code, conf
  FROM seats_confs sc
    CROSS JOIN LATERAL jsonb_array_elements( sc.seats_conf ) AS conf
)
SELECT aircraft_code, param.*
FROM fare_cond_confs fcc
  CROSS JOIN LATERAL jsonb_each( fcc.conf ) AS param;
```

```
 aircraft_code |       key       |           value
---------------+-----------------+---------------------------
 SU9           | rows            | [1, 3]
 SU9           | letters         | ["A", "C", "D", "F"]
 SU9           | fare conditions | "Business"
 SU9           | rows            | [4, 20]
 SU9           | letters         | ["A", "C", "D", "E", "F"]
 SU9           | fare conditions | "Economy"
 CN1           | rows            | [1, 6]
 CN1           | letters         | ["A", "B"]
 CN1           | fare conditions | "Economy"
(9 строк)
```

В подзапросе `fare_cond_confs` функция `jsonb_array_elements` вызывается для *каждой* строки исходной временной таблицы `seats_confs`, созданной в конструкции `WITH`. Поскольку эта функция преобразует JSON-массив во *множество* JSON-объектов, то для каждой модели самолета формируется столько строк,

сколько элементов содержится в массиве. Для модели с кодом SU9 будет сформировано две строки (классы Business и Economy), а для модели CN1 — только одна (класс Economy).

Можно сказать, что применение функций в предложении FROM позволяет для *каждой* строки таблицы, стоящей слева от ключевого слова JOIN, создать *требуемое* число строк в зависимости от информации, содержащейся в этой строке.

В основном запросе элементы JSON-массива, являющиеся JSON-объектами, выводятся в виде пар ключ—значение с помощью функции jsonb_each. Эта функция вызывается для *каждой* строки временной таблицы fare_cond_confs и формирует столько строк, сколько ключей у объекта в этой строке. В нашем примере каждый JSON-объект состоит ровно из трех строк, но в общем случае количество может различаться.

Теперь получим итоговые результаты в виде, пригодном для вставки в таблицу «Места» (seats):

```
WITH seats_confs( aircraft_code, seats_conf ) AS
...
fare_cond_confs AS
( SELECT aircraft_code, conf
  FROM seats_confs sc
    CROSS JOIN LATERAL jsonb_array_elements( sc.seats_conf ) AS conf
)
SELECT
  aircraft_code,
  fcc.conf->>'fare conditions' AS fare_conditions,
  row || letter AS seat_no
FROM fare_cond_confs fcc
  CROSS JOIN LATERAL generate_series(
      ( fcc.conf->'rows'->>0 )::integer, ( fcc.conf->'rows'->>1 )::integer
    ) AS rows( row )
  CROSS JOIN LATERAL jsonb_array_elements_text( fcc.conf->'letters' ) AS letters( letter )
ORDER BY aircraft_code, fare_conditions, row, letter;
```

В основном запросе для *каждой* строки временной таблицы fare_cond_confs вызывается функция generate_series, которая формирует список номеров рядов кресел на основе массива, доступного по ключу rows. Обратите внимание на оператор ->>, представляющий элемент JSON-массива в виде текстового значения. Поскольку функция generate_series требует числовых параметров, приходится эти текстовые значения привести к типу integer. Как мы

уже говорили ранее, декартово произведение CROSS JOIN тиражирует строку таблицы fare_cond_confs столько раз, сколько номеров сгенерирует функция generate_series. Нужно назначить результату работы функции псевдоним таблицы и имя столбца, если назначенные по умолчанию имена нас не устраивают.

Теперь *каждый* номер ряда, сгенерированный на предыдущем этапе, нужно скомбинировать с *каждым* значением из списка буквенных обозначений кресел в этом ряду. Поэтому опять мы видим конструкцию CROSS JOIN, в которой вызывается уже функция jsonb_array_elements_text. Она представляет JSON-массив в виде множества строк (таблицы с одним столбцом). Опять нужно назначить результату работы функции псевдоним таблицы и имя столбца.

Как ранее уже было сказано, при вызове функций в предложении FROM ключевое слово LATERAL можно не использовать.

```
 aircraft_code | fare_conditions | seat_no
---------------+-----------------+---------
 CN1           | Economy         | 1A
 CN1           | Economy         | 1B
 ...
 CN1           | Economy         | 6A
 CN1           | Economy         | 6B
 SU9           | Business        | 1A
 SU9           | Business        | 1C
 SU9           | Business        | 1D
 SU9           | Business        | 1F
 ...
 SU9           | Business        | 3A
 SU9           | Business        | 3C
 SU9           | Business        | 3D
 SU9           | Business        | 3F
 SU9           | Economy         | 4A
 SU9           | Economy         | 4C
 SU9           | Economy         | 4D
 SU9           | Economy         | 4E
 SU9           | Economy         | 4F
 ...
(109 строк)
```

В рассмотренном запросе нам помогли функции, обрабатывающие значения JSON. PostgreSQL предлагает большое количество разнообразных функций этого класса. Они представлены в разделе документации 9.16 «Функции и операторы JSON».

В 17-й версии PostgreSQL представлена новая функция `JSON_TABLE` (см. подраздел документации 9.16.4 «JSON_TABLE»). С ее помощью также можно решить поставленную задачу. Однако сначала нужно сказать о языке путей SQL/JSON, представленном в подразделе документации 9.16.2 «Язык путей SQL/JSON», который позволяет эффективно выполнять запросы к данным JSON. На нем можно записать путь к извлекаемому элементу (элементам) значения JSON, задав, если нужно, условия отбора. Путь формируется в виде значения типа `jsonpath`, который описан в подразделе документации 8.14.7 «Тип jsonpath».

Для обращения к обрабатываемому значению JSON (контекстному элементу) используется переменная с именем `$`. За ней может скрываться как все исходное значение JSON, так и его составной элемент, полученный с помощью операторов обращения (см. таблицу 8.25 «Операторы обращения в jsonpath»). Они позволяют углубляться в структуру значения JSON и извлекать элементы, вложенные в текущий контекстный элемент. При этом каждый последующий оператор имеет дело с результатом, полученным на предыдущем шаге. Для обращения к элементу по его ключу служит оператор *.ключ*, значения всех элементов верхнего уровня возвращает оператор `.*`. Один элемент массива выдает оператор [*индекс*], а все элементы массива — оператор [`*`].

Теперь вернемся к задаче формирования номеров кресел в салоне самолета. Сначала выполним совсем простой запрос, который лишь представит описание каждого салона самолета в виде одной строки выборки.

```
WITH seats_confs( aircraft_code, seats_conf ) AS
( VALUES
  ( 'SU9',
    '[ { "fare conditions": "Business",
         "rows": [ 1, 3 ],
         "letters": [ "A", "C", "D", "F" ] },
       { "fare conditions": "Economy",
         "rows": [ 4, 20 ],
         "letters": [ "A", "C", "D", "E", "F" ] }
    ]'::jsonb
  ),
  ( 'CN1',
    '[ { "fare conditions": "Economy",
         "rows": [ 1, 6 ],
         "letters": [ "A", "B" ] }
    ]'::jsonb
  )
)
```

```
SELECT sc.aircraft_code, jt.*
FROM seats_confs AS sc
  CROSS JOIN LATERAL JSON_TABLE(
      seats_conf,
      '$[ * ]'
      COLUMNS
      ( fare_conditions text PATH '$."fare conditions"',
        row_from integer PATH '$.rows[ 0 ]',
        row_to integer PATH '$.rows[ 1 ]',
        letters text[] PATH '$.letters'
      )
    ) AS jt
ORDER BY aircraft_code, row_from;
 aircraft_code | fare_conditions | row_from | row_to |   letters
---------------+-----------------+----------+--------+-------------
 CN1           | Economy         |        1 |      6 | {A,B}
 SU9           | Business        |        1 |      3 | {A,C,D,F}
 SU9           | Economy         |        4 |     20 | {A,C,D,E,F}
(3 строки)
```

Имя функции JSON_TABLE в запросе написано заглавными буквами, как в документации, хотя можно использовать и строчные.

Первым параметром функции является JSON-значение, взятое из текущей строки таблицы seats_confs. Второй параметр — путь к извлекаемым элементам этого значения. В этом пути переменная $ представляет собой все исходное JSON-значение, являющееся JSON-массивом, поэтому оператор [*] извлекает все его элементы, которые, в свою очередь, представляют собой JSON-объекты. Каждый такой объект содержит описание одного салона с конкретным классом обслуживания.

Предложение COLUMNS предназначено для определения столбцов формируемой выборки. Для каждого из них задается имя, тип данных и путь для получения его значения. В этом пути переменная $ представляет собой JSON-объект, выбранный на предыдущем этапе и содержащий описание одного салона самолета. Применение операторов обращения к этой переменной позволяет получить номера первого и последнего рядов салона, а также массив буквенных обозначений кресел. Обратите внимание, что типом данных столбца letters является массив значений типа text.

Поскольку ключ fare conditions содержит пробел, приходится использовать кавычки в определении пути для столбца fare_conditions.

Строки, порожденные функцией JSON_TABLE, соединяются со строкой, на основе которой они были сформированы, в обычной конструкции LATERAL.

Давайте усовершенствуем запрос: пусть он теперь разворачивает массивы буквенных обозначений кресел в виде множества строк.

```
WITH seats_confs( aircraft_code, seats_conf ) AS
...
SELECT sc.aircraft_code, jt.*
FROM seats_confs AS sc
  CROSS JOIN LATERAL JSON_TABLE(
      seats_conf,
      '$[ * ]'
      COLUMNS
      ( fare_conditions text PATH '$."fare conditions"',
        row_from integer PATH '$.rows[ 0 ]',
        row_to integer PATH '$.rows[ 1 ]',
        NESTED PATH '$.letters[ * ]' COLUMNS
        ( letter text PATH '$'
        )
      )
    ) AS jt
ORDER BY aircraft_code, row_from, letter;
```

```
 aircraft_code | fare_conditions | row_from | row_to | letter
---------------+-----------------+----------+--------+--------
 CN1           | Economy         |        1 |      6 | A
 CN1           | Economy         |        1 |      6 | B
 SU9           | Business        |        1 |      3 | A
 SU9           | Business        |        1 |      3 | C
 SU9           | Business        |        1 |      3 | D
 SU9           | Business        |        1 |      3 | F
 SU9           | Economy         |        4 |     20 | A
 SU9           | Economy         |        4 |     20 | C
 SU9           | Economy         |        4 |     20 | D
 SU9           | Economy         |        4 |     20 | E
 SU9           | Economy         |        4 |     20 | F
(11 строк)
```

Для получения значений вложенных элементов служит предложение NESTED PATH. Путь $.letters дополнен оператором [*] для получения всех элементов массива в виде отдельных сущностей. Теперь в определении пути для столбца letter находится только переменная $, которая содержит контекстный элемент, а в данном контексте он представляет собой одну букву. Важно, что предложение NESTED PATH является рекурсивным, что позволяет разобрать всю иерархию

JSON-значения в одном вызове функции JSON_TABLE. В нашем примере всего один столбец, соответствующий вложенным элементам JSON-значения, — letters, но в общем случае таких столбцов может быть несколько.

Для получения окончательного результата придется дополнить запрос обращением к функции generate_series.

```
WITH seats_confs( aircraft_code, seats_conf ) AS
...
SELECT sc.aircraft_code, jt.fare_conditions, row || letter AS seat_no
FROM seats_confs AS sc
  CROSS JOIN LATERAL JSON_TABLE(
      seats_conf,
      '$[ * ]'
      COLUMNS
      ( fare_conditions text PATH '$."fare conditions"',
        row_from integer PATH '$.rows[ 0 ]',
        row_to integer PATH '$.rows[ 1 ]',
        NESTED PATH '$.letters[ * ]' COLUMNS
        ( letter text PATH '$'
        )
      )
    ) AS jt
  CROSS JOIN LATERAL generate_series( row_from, row_to ) AS rows( row )
ORDER BY aircraft_code, row, letter;
```

```
 aircraft_code | fare_conditions | seat_no
---------------+-----------------+---------
 CN1           | Economy         | 1A
 CN1           | Economy         | 1B
 CN1           | Economy         | 2A
 ...
 SU9           | Economy         | 20D
 SU9           | Economy         | 20E
 SU9           | Economy         | 20F
(109 строк)
```

Таким образом, функция JSON_TABLE предоставляет декларативный способ преобразования иерархического JSON-объекта в табличную форму, пригодную для выполнения обычных реляционных операций.

В рассмотренном примере решение с помощью функции JSON_TABLE получилось лишь немного проще, чем «традиционное» решение. Однако по мере усложнения иерархической структуры JSON-объекта выигрыш от использования этой функции может быть весьма значительным, поскольку один ее вызов заменяет вызовы целого ряда других функций.

4.4. Контрольные вопросы и задания

1 Оконная функция вместо конструкции LATERAL при отборе пассажиров для поощрения

В разделе 4.1 «Подзапросы в предложении FROM» (с. 225) решалась задача отбора десятой части пассажиров дальневосточных рейсов для поощрения. Была использована конструкция LATERAL. В связи с этим возникает вопрос: нельзя ли в данном случае обойтись без нее? Можно, и вот одно из решений. Будет ли оно выдавать результат быстрее, чем исходный запрос, покажут эксперименты.

```
WITH row_nums AS
( SELECT
    f.flight_id,
    bp.ticket_no,
    row_number() OVER ( PARTITION BY f.flight_id ORDER BY random() ) AS row_num
  FROM flights AS f
    JOIN boarding_passes AS bp ON bp.flight_id = f.flight_id
  WHERE f.status = 'Arrived'
    AND ( ( f.departure_airport IN ( 'DME', 'SVO', 'VKO' )
            AND
            f.arrival_airport IN ( 'VVO', 'KHV', 'UUS', 'BOS' )
          )
          OR
          ( f.departure_airport IN ( 'VVO', 'KHV', 'UUS', 'BOS' )
            AND
            f.arrival_airport IN ( 'DME', 'SVO', 'VKO' )
          )
        )
)
SELECT flight_id, ticket_no, row_num
FROM row_nums
WHERE row_num % 10 = 1
ORDER BY flight_id, row_num;
```

```
 flight_id |   ticket_no   | row_num
-----------+---------------+---------
      1249 | 0005435824364 |       1
      1249 | 0005432284530 |      11
      1249 | 0005435824363 |      21
      1249 | 0005435824354 |      31
      1249 | 0005435824359 |      41
      1250 | 0005435824709 |       1
 ...
```

```
 26614 | 0005435947237 |      11
 26614 | 0005435856199 |      21
 26614 | 0005435787354 |      31
 26614 | 0005435986694 |      41
 26616 | 0005432571281 |       1
 26616 | 0005435856158 |      11
 26616 | 0005435856159 |      21
(1858 строк)
```

Нам удалось отказаться от конструкции LATERAL за счет изменения способа отбора пассажиров. Вместо предложения LIMIT мы пронумеровали строки таблицы. Столбец row_num добавлен для демонстрации того, что условие отбора каждого десятого пассажира (row_num % 10 = 1) работает корректно.

Этот запрос также можно модифицировать, например, перенеся условие отбора строк и операцию соединения в главный запрос, оставив в общем табличном выражении только операцию нумерования строк таблицы «Посадочные талоны» (boarding_passes).

Задание 1. Сравните быстродействие всех вариантов решения задачи отбора «счастливых» билетов, приведенных в тексте главы и в этом упражнении. Попытайтесь объяснить различия в затратах времени. При необходимости изучите планы выполнения запросов.

Задание 2. Предположим, что наша авиакомпания решила поощрять пассажиров всех рейсов, а не только дальневосточных. Измените соответствующим образом условия в запросах и повторите эксперименты. Посмотрите, как изменились соотношения времени выполнения запросов на выборке большего размера. Почему это произошло?

2 Еще один вариант решения задачи отбора пассажиров для поощрения

Рассмотрим еще один вариант решения задачи о выборе номеров билетов для поощрения пассажиров. В предложении FROM команды SELECT есть возможность указать способ случайного отбора строк с помощью ключевого слова TABLESAMPLE. Запрос может стать таким:

```
SELECT f.flight_id, bp.ticket_no
FROM flights f
  CROSS JOIN LATERAL
    ( SELECT ticket_no
      FROM boarding_passes bp TABLESAMPLE BERNOULLI ( 10.0 )
      WHERE bp.flight_id = f.flight_id
    ) AS bp
WHERE f.status = 'Arrived'
  AND ( ( f.departure_airport IN ( 'DME', 'SVO', 'VKO' )
          AND
          f.arrival_airport IN ( 'VVO', 'KHV', 'UUS', 'BOS' )
        )
        OR
        ( f.departure_airport IN ( 'VVO', 'KHV', 'UUS', 'BOS' )
          AND
          f.arrival_airport IN ( 'DME', 'SVO', 'VKO' )
        )
      )
ORDER BY f.flight_id, bp.ticket_no;
```

Этот запрос выглядит очень привлекательно. Как вы думаете, выдает ли он корректный результат?

Задание. Для проверки своей гипотезы сравните результат работы этого запроса с другими вариантами, которые были рассмотрены в тексте главы и в предыдущем упражнении. Обратите внимание на те рейсы, численность пассажиров на которых не превышала девяти человек (мы ведь отбираем десять процентов строк). Учтите следующее важное замечание. В описании команды SELECT, приведенном в документации, сказано, что отбор строк из таблицы при использовании предложения TABLESAMPLE выполняется до наложения любых других фильтров, в том числе условий из предложения WHERE.

Указание. В качестве примера использования предложения TABLESAMPLE можно выполнить запрос, выбирающий десять процентов строк из таблицы, в которой всего девять строк.

```
SELECT * FROM aircrafts_data TABLESAMPLE BERNOULLI ( 10.0 );
```

Проведя значительное количество экспериментов с этим запросом, можно получить в результате одну, две, три и даже четыре строки. Можно получить и большее число строк, но вероятность такого исхода крайне мала.

```
 aircraft_code |                                model                                 | range
---------------+----------------------------------------------------------------------+-------
 321           | {"en": "Airbus A321-200", "ru": "Аэробус A321-200"}                  |  5600
 733           | {"en": "Boeing 737-300", "ru": "Боинг 737-300"}                      |  4200
 CN1           | {"en": "Cessna 208 Caravan", "ru": "Сессна 208 Караван"}             |  1200
 CR2           | {"en": "Bombardier CRJ-200", "ru": "Бомбардье CRJ-200"}              |  2700
(4 строки)
```

А можно получить и пустую выборку (в данном случае это происходит довольно часто):

```
 aircraft_code | model | range
---------------+-------+-------
(0 строк)
```

Обратите внимание, что в запросе используется не представление aircrafts, а таблица aircrafts_data, на которой оно основано. В разделе «Совместимость» описания команды SELECT, приведенного в документации, сказано, что механизм TABLESAMPLE работает только с обычными таблицами и материализованными представлениями.

3 Наглядное представление планировок салонов

Пассажир имеет право выбрать место в салоне самолета (конечно, из числа свободных). Причем за возможность выбора некоторых мест авиакомпании взимают плату. Предположим, что маркетологи нашей авиакомпании решили выяснить степень популярности различных мест в салонах самолетов. Для этого они попросили нас представить в удобной форме конфигурации салонов, а именно: для каждого класса обслуживания нужно привести номера первого и последнего рядов, а также список буквенных обозначений кресел в ряду. Конечно, это упрощение реальной ситуации, поскольку в салоне одного класса обслуживания могут находиться ряды с разным числом кресел, и каждая группа рядов потребует отдельного описания.

Необходимую информацию можно получить из таблицы «Места» (seats). Рассмотрим решение этой задачи только для класса обслуживания Business, причем представим два варианта. В каждом из них будем исходить из того, что все ряды салона этого класса в конкретной модели самолета имеют одно и то же число кресел.

В первом варианте не используется конструкция LATERAL:

```
SELECT
  a.aircraft_code AS a_code,
  a.model,
  sc.first_row AS f_row,
  sc.last_row AS l_row,
  sc.seats_config
FROM aircrafts AS a
  LEFT OUTER JOIN
    ( SELECT
        aircraft_code,
        min( ( left( seat_no, -1 ) )::int ) AS first_row,
        max( ( left( seat_no, -1 ) )::int ) AS last_row,
        array_agg( DISTINCT right( seat_no, 1 ) ORDER BY right( seat_no, 1 ) )
          AS seats_config
      FROM seats
      WHERE fare_conditions = 'Business'
      GROUP BY aircraft_code
    ) AS sc -- seats_config
    ON sc.aircraft_code = a.aircraft_code
ORDER BY a.model;
```

Обратите внимание, что левое внешнее соединение необходимо, поскольку у ряда моделей нет салона бизнес-класса. Все функции, использованные в запросе, уже известны читателю. Добавим только, что отрицательное значение второго параметра функции left позволяет отбросить часть символов строки, начиная с ее правого конца. В нашем случае отбрасывается только один символ — буквенное обозначение места в ряду, и в результате остается лишь номер ряда. И еще важно, что в вызове функции array_agg присутствует ключевое слово DISTINCT. Без него мы получили бы многократное дублирование букв.

```
 a_code |        model        | f_row | l_row | seats_config
--------+---------------------+-------+-------+--------------
 319    | Аэробус A319-100    |     1 |     5 | {A,C,D,F}
 320    | Аэробус A320-200    |     1 |     5 | {A,C,D,F}
 321    | Аэробус A321-200    |     1 |     7 | {A,C,D,F}
 733    | Боинг 737-300       |     1 |     3 | {A,C,D,F}
 763    | Боинг 767-300       |     1 |     5 | {A,B,C,F,G,H}
 773    | Боинг 777-300       |     1 |     5 | {A,C,D,G,H,K}
 CR2    | Бомбардье CRJ-200   |       |       |
 CN1    | Сессна 208 Караван  |       |       |
 SU9    | Сухой Суперджет-100 |     1 |     3 | {A,C,D,F}
(9 строк)
```

Заметим, что возможность указывать и DISTINCT, и ORDER BY в агрегатном выражении — это расширение PostgreSQL, как сказано в подразделе документации 4.2.7 «Агрегатные выражения».

Тот же результат можно получить и с помощью запроса, использующего конструкцию LATERAL.

```
SELECT
  a.aircraft_code AS a_code,
  a.model,
  s.first_row AS f_row,
  s.last_row AS l_row,
  s.seats_config
FROM aircrafts AS a,
  LATERAL
    ( SELECT
        min( ( left( s.seat_no, -1 ) )::int ) AS first_row,
        max( ( left( s.seat_no, -1 ) )::int ) AS last_row,
        array_agg( DISTINCT right( s.seat_no, 1 ) ORDER BY right( s.seat_no, 1 ) )
          AS seats_config
      FROM seats s
      WHERE s.aircraft_code = a.aircraft_code
        AND s.fare_conditions = 'Business'
    ) AS s
ORDER BY a.model;
```

Напомним, что запятая между элементами в предложении FROM равнозначна ключевым словам CROSS JOIN.

Вопрос. В запросе с конструкцией LATERAL не использовано левое внешнее соединение. Тем не менее те модели, у которых нет салона бизнес-класса, также вошли в выборку. Как это можно объяснить?

Указание. Обратите внимание на то, сколько строк выводит следующий упрощенный запрос и какое значение содержится в столбце first_row:

```
SELECT min( ( left( seat_no, -1 ) )::int ) AS first_row
FROM seats
WHERE aircraft_code = 'CN1'
  AND fare_conditions = 'Business';
 first_row
-----------

(1 строка)
```

Задание. Напишите запрос, формирующий конфигурацию салонов с учетом всех трех существующих классов обслуживания. Найдите решение как с использованием конструкции LATERAL, так и без нее. Сравните сложность этих вариантов запроса.

4 Напоминает ли конструкция LATERAL коррелированный подзапрос?

Это упражнение является небольшим эскизом, который позволит лучше разобраться в механизме работы конструкции LATERAL. Для подсчета числа кресел в каждой модели самолета можно воспользоваться, например, таким запросом:

```
SELECT a.model, count( * ) AS seats_cnt
FROM aircrafts a
  JOIN seats s ON s.aircraft_code = a.aircraft_code
GROUP BY a.model
ORDER BY seats_cnt DESC;
```

```
        model        | seats_cnt
---------------------+-----------
 Боинг 777-300       |       402
 Боинг 767-300       |       222
 Аэробус A321-200    |       170
 Аэробус A320-200    |       140
 Боинг 737-300       |       130
 Аэробус A319-100    |       116
 Сухой Суперджет-100 |        97
 Бомбардье CRJ-200   |        50
 Сессна 208 Караван  |        12
(9 строк)
```

Задание. Напишите запрос, решающий эту же задачу с использованием конструкции LATERAL. Не напоминает ли такое решение использование коррелированного подзапроса?

5 Выявление пропусков в нумерации

Предположим, что мы анализируем базу «Авиаперевозки» на предмет возможного рассогласования данных. В таблице «Посадочные талоны» (boarding_passes) есть столбец boarding_no, содержащий значения порядковых номеров посадочных талонов. Требуется выявить пропуски в этой нумерации, если они

присутствуют. В разделе 4.2 «Вызовы функций в предложении FROM» (с. 234) такая задача уже решалась, но с применением другого алгоритма.

Давайте сначала решим задачу для более простой — абстрактной — ситуации. Создадим таблицу с одним столбцом, заполним ее данными, а затем удалим несколько строк, чтобы образовались пропуски в нумерации.

```
CREATE TABLE numbers ( a int );
CREATE TABLE
INSERT INTO numbers SELECT g FROM generate_series( 1, 100 ) AS g;
INSERT 0 100
DELETE FROM numbers WHERE a IN ( 19, 50, 51, 52 );
DELETE 4
```

Идея алгоритма такова: для каждого значения нужно найти наименьшее значение, превышающее его. Если разность этой пары значений превышает единицу, значит, имеет место пропуск в нумерации. Надо отобрать такие пары и для каждой из них сгенерировать пропущенные значения.

```
WITH gaps AS
( SELECT a1, a2
  FROM numbers AS n1
    CROSS JOIN LATERAL
      ( SELECT n1.a AS a1, n2.a AS a2
        FROM numbers AS n2
        WHERE n2.a > n1.a
        ORDER BY n2.a
        LIMIT 1
      ) AS n2
  WHERE a2 - a1 > 1
)
SELECT missing
FROM gaps
  CROSS JOIN generate_series( a1 + 1, a2 - 1 ) AS missing
ORDER BY missing;
 missing
---------
      19
      50
      51
      52
(4 строки)
```

Теперь воспользуемся предложенным алгоритмом применительно к таблице «Посадочные талоны» (boarding_passes).

```
BEGIN;
BEGIN
DELETE FROM boarding_passes
WHERE flight_id = 1969
  AND boarding_no IN ( 1, 2 );
DELETE 2
DELETE FROM boarding_passes
WHERE flight_id = 2023
  AND boarding_no IN ( 13, 14, 15 );
DELETE 3
WITH gaps AS
( SELECT
    bp2.flight_id,
    boarding_no1,
    boarding_no2
  FROM flights AS f
    CROSS JOIN LATERAL
      ( SELECT 0 AS boarding_no -- если пропуски в самом начале нумерации
        UNION ALL
        SELECT boarding_no
        FROM boarding_passes AS bp
        WHERE bp.flight_id = f.flight_id
      ) AS bp1
    CROSS JOIN LATERAL
      ( SELECT
          f.flight_id,
          bp1.boarding_no AS boarding_no1,
          bp2.boarding_no AS boarding_no2
        FROM boarding_passes AS bp2
        WHERE bp2.flight_id = f.flight_id
          AND bp2.boarding_no > bp1.boarding_no
        ORDER BY bp2.boarding_no
        LIMIT 1
      ) AS bp2
  WHERE boarding_no2 - boarding_no1 > 1
)
SELECT
  flight_id,
  missing
FROM gaps
  CROSS JOIN generate_series( boarding_no1 + 1, boarding_no2 - 1 ) AS missing
ORDER BY flight_id, missing;
```

```
 flight_id | missing
-----------+---------
      1969 |       1
      1969 |       2
      2023 |      13
      2023 |      14
      2023 |      15
(5 строк)
ROLLBACK;
ROLLBACK
```

Задание 1. Самостоятельно разберитесь, как работает этот запрос. В нем в предложении FROM на один элемент больше, чем в абстрактном примере, поскольку нумерация посадочных талонов ведется в рамках каждого рейса. За счет чего запрос выявляет пропуски в самом начале нумерации? Обратите внимание, что подзапрос bp2 ссылается не только на ближайший слева подзапрос bp1, но также и на таблицу flights, которая находится в предложении FROM еще левее. Попробуйте проследить по шагам, как происходит соединение наборов строк в подзапросах конструкции WITH.

Задание 2. Смоделируйте еще какую-либо ситуацию в базе данных «Авиаперевозки», в которой можно было бы использовать подобный алгоритм выявления пропусков в нумерации (возможно, с модификациями алгоритма).

6 Конструкция LATERAL вместо оконных функций

В ряде ситуаций подзапросы с ключевым словом LATERAL могут заменить оконные функции. В качестве примера воспользуемся таблицей «Самолеты» (aircrafts). Для моделей самолетов, производимых компаниями Airbus и Boeing, определим максимальную дальность полета среди моделей каждой из компаний. Затем для каждой модели вычислим абсолютные отклонения дальности ее полета от максимального значения среди моделей этой же компании. Рассчитаем также и отношения этих показателей.

Сначала покажем запрос, в котором задача решается с помощью оконной функции. Поскольку одно и то же определение окна используется в запросе неоднократно, зададим его в предложении WINDOW. Обратите внимание, что для вычисления значений столбцов delta и coeff нельзя использовать имя max_range в качестве готового значения, а приходится повторять конструкцию OVER.

```
SELECT
 aircraft_code AS a_code,
 model,
 range,
 max( range ) OVER company_win AS max_range,
 max( range ) OVER company_win - range AS delta,
 round( range::numeric / max( range ) OVER company_win, 2 ) AS coeff
FROM aircrafts
WHERE left( model, 5 ) IN ( 'Аэроб', 'Боинг' )
WINDOW company_win AS ( PARTITION BY left( model, 5 ) )
ORDER BY left( model, 5 ), range DESC;
```

Теперь представим эквивалентный запрос, построенный на основе конструкции LATERAL. В первом подзапросе определяется максимальная дальность полета для моделей каждой из двух компаний, поэтому здесь формируется всего две строки. Каждая из этих строк затем соединяется только со строками таблицы «Самолеты» (aircrafts), описывающими модели соответствующей компании.

```
SELECT
  aircraft_code AS a_code,
  model,
  range,
  max_range,
  delta,
  coeff
FROM
    ( SELECT
        left( model, 5 ) AS company,
        max( range ) AS max_range
      FROM aircrafts
      WHERE left( model, 5 ) IN ( 'Аэроб', 'Боинг' )
      GROUP BY company
    ) AS mr  -- maximal ranges
  JOIN LATERAL
    ( SELECT
        aircraft_code,
        model,
        range,
        mr.max_range - range AS delta,
        round( range::numeric / mr.max_range, 2 ) AS coeff
      FROM aircrafts
    ) AS deltas
    ON left( model, 5 ) = mr.company
ORDER BY left( model, 5 ), range DESC;
```

```
 a_code |      model       | range | max_range | delta | coeff
--------+------------------+-------+-----------+-------+-------
 319    | Аэробус A319-100 |  6700 |      6700 |     0 |  1.00
 320    | Аэробус A320-200 |  5700 |      6700 |  1000 |  0.85
 321    | Аэробус A321-200 |  5600 |      6700 |  1100 |  0.84
 773    | Боинг 777-300    | 11100 |     11100 |     0 |  1.00
 763    | Боинг 767-300    |  7900 |     11100 |  3200 |  0.71
 733    | Боинг 737-300    |  4200 |     11100 |  6900 |  0.38
(6 строк)
```

Задание 1. Сопоставьте два приведенных запроса с точки зрения простоты интерпретации. Поскольку выборка совсем небольшая, то говорить о сравнении быстродействия не имеет смысла.

Задание 2. Предложите в рамках базы данных «Авиаперевозки» аналогичную пару запросов. Можно обратиться, например, к таблице «Аэропорты» (airports). Аэропорты распределены по часовым поясам, каждый аэропорт имеет координаты — долготу и широту.

Вопрос. Как вы думаете, любой ли запрос, в котором используется оконная функция, можно заменить эквивалентным запросом с конструкцией LATERAL?

Указание. Вспомните, что существуют различные способы формирования оконного кадра.

Задание 3. Модифицируйте запрос с конструкцией LATERAL таким образом, чтобы можно было обойтись без нее. Для этого можно перенести вычисления значений delta и coeff из подзапроса deltas в список SELECT главного запроса. Без конструкции LATERAL ссылаться на столбцы подзапроса mr непосредственно в подзапросе deltas будет нельзя, а в предложении ON — можно.

Вопрос. Как вы думаете, любой ли запрос, в котором используется конструкция LATERAL, можно заменить эквивалентным запросом с «обычным» соединением таблиц или подзапросов? Попробуйте дать ответ в общем случае, а не только приведя примеры запросов.

7 Использование конструкций LATERAL и WITH для многошаговых вычислений

Бывают ситуации, когда необходимо выполнить довольно громоздкие многошаговые вычисления. В таких случаях конструкция LATERAL может избавить нас

от необходимости многократно повторять в списке SELECT одни и те же выражения. Рассмотрим абстрактный пример. В таблице содержатся некоторые данные и ряд коэффициентов, используемых в вычислениях.

```
CREATE TABLE some_data
( name text PRIMARY KEY,  -- наименование объекта
  data int,               -- полезные данные
  a numeric( 5, 2 ),      -- коэффициент a
  b numeric( 5, 2 ),      -- коэффициент b
  c numeric( 5, 2 )       -- коэффициент c
);
CREATE TABLE
INSERT INTO some_data VALUES
( 'obj1', 27, 1.25, 3.0,  2.5 ),
( 'obj2', 63, 1.75, 3.5,  2.75 ),
( 'obj3', 48, 1.5,  4.5,  1.5 ),
( 'obj4', 19, 0.75, 4.25, 2.25 ),
( 'obj5', 35, 2.25, 2.75, 1.75 );
INSERT 0 5
```

Сначала приведем запрос без конструкции LATERAL:

```
SELECT
  name,
  data,
  round( a * b, 2 ) AS ab,
  round( data * a * b, 2 ) AS data_ab,
  round( a * b / c, 2 ) AS abc,
  round( data * a * b / c, 2 ) AS data_abc
FROM some_data;
 name | data |  ab  | data_ab | abc  | data_abc
------+------+------+---------+------+----------
 obj1 |   27 | 3.75 |  101.25 | 1.50 |    40.50
 obj2 |   63 | 6.13 |  385.88 | 2.23 |   140.32
 obj3 |   48 | 6.75 |  324.00 | 4.50 |   216.00
 obj4 |   19 | 3.19 |   60.56 | 1.42 |    26.92
 obj5 |   35 | 6.19 |  216.56 | 3.54 |   123.75
(5 строк)
```

Расчетные формулы здесь не имеют какого-либо содержательного наполнения, их назначение — проиллюстрировать пошаговый характер вычислений. Обратите внимание, что математические выражения в списке SELECT повторяются несколько раз. Например, выражение (a * b) повторяется четырежды. Если бы

на месте этого простого выражения находилось что-то значительно более сложное, то запрос было бы трудно читать, а при необходимости внесения изменений риск ошибки или описки (при этом не сразу замеченной) значительно возрос бы.

Нельзя, с целью избежать повторов одних и тех же выражений, переписать запрос таким образом:

```
SELECT
  name,
  data,
  a * b AS source_ab,
  source_ab / c AS source_abc,
  round( source_ab, 2 ) AS ab,
  round( data * source_ab, 2 ) AS data_ab,
  round( source_abc, 2 ) AS abc,
  round( data * source_abc, 2 ) AS data_abc
FROM some_data;
```

```
ОШИБКА:  столбец "source_ab" не существует
СТРОКА 5:   source_ab / c AS source_abc,
            ^
```

А что можно сделать с помощью конструкции LATERAL? Можно перенести вычисления коэффициентов в подзапросы, оставив в списке SELECT главного запроса только финальные вычисления. Тем самым удастся избежать повторного включения в текст запроса одних и тех же выражений.

```
SELECT
  abs.name,
  abs.data,
  round( abs.ab, 2 ) AS ab,
  round( abs.data * abs.ab, 2 ) AS data_ab,
  round( abcs.abc, 2 ) AS abc,
  round( abs.data * abcs.abc, 2 ) AS data_abc
FROM
    ( SELECT name, data, a * b AS ab
      FROM some_data
    ) AS abs -- s означает множественное число
  JOIN LATERAL
    ( SELECT name, abs.ab / c AS abc
      FROM some_data
    ) AS abcs
    ON abcs.name = abs.name;
```

Вопрос. Как вы думаете, будет ли планировщик фактически выполнять промежуточные вычисления в узлах плана, соответствующих подзапросам, или они так же, как и в предыдущем запросе, будут выполняться в списке SELECT главного запроса? Сделайте обоснованное предположение, а затем проверьте его на практике. Можно добавить в команду EXPLAIN параметр VERBOSE.

Задание 1. Перепишите запрос, используя общие табличные выражения, например так:

```
WITH abs AS -- MATERIALIZED
( SELECT name, data, a * b AS ab, c
  FROM some_data
),
abcs AS -- MATERIALIZED
( SELECT name, data, ab, ab / c AS abc
  FROM abs
)
SELECT
  name,
  data,
  round( ab, 2 ) AS ab,
  round( data * ab, 2 ) AS data_ab,
  round( abc, 2 ) AS abc,
  round( data * abc, 2 ) AS data_abc
FROM abcs;
```

Выполните все три варианта запроса, причем последний как без материализации общих табличных выражений, так и с нею. Посмотрите планы запросов и время их выполнения.

Задание 2. Вставьте в таблицу some_data значительно большее число строк и повторите эксперименты. Для заполнения таблицы можно воспользоваться функциями generate_series и random, например:

```
INSERT INTO some_data
SELECT
  'obj' || g, 1 + ( random() * 99 )::int,
  0.1 + ( random() * 9.9 )::numeric( 5,2 ),
  0.1 + ( random() * 9.9 )::numeric( 5,2 ),
  0.1 + ( random() * 9.9 )::numeric( 5,2 )
FROM generate_series( 1, 100000 ) AS g;
```

Изменилось ли соотношение времени выполнения запросов? Изменился ли план запроса с конструкцией LATERAL?

Задание 3. В качестве имитации сложных вычислений замените обращения к столбцам a, b, c вызовами какой-нибудь функции, например sqrt:

```
SELECT
  name,
  data,
  round( sqrt( a ) * sqrt( b ), 2 ) AS ab,
  round( data * sqrt( a ) * sqrt( b ), 2 ) AS data_ab,
  round( sqrt( a ) * sqrt( b ) / sqrt( c ), 2 ) AS abc,
  round( data * sqrt( a ) * sqrt( b ) / sqrt( c ), 2 ) AS data_abc
FROM some_data;
```

Повторите эксперименты. Как изменилось соотношение времен выполнения запросов? Можно ли сказать, что при наличии сложных вычислений вариант с материализацией общих табличных выражений оказывается самым быстрым? Уступает ли ему вариант с конструкцией LATERAL? Если да, то почему?

Задание 4. Предложите ситуацию в предметной области «Авиаперевозки», которая требовала бы пошаговых вычислений. Напишите аналогичную пару запросов и проведите эксперименты с ними. В качестве примера можно рассмотреть предполетное обслуживание самолетов. Выберите количество нормочасов, требующееся для проведения обслуживания одного самолета каждой модели, и соотнесите время, затраченное на обслуживание, с общим временем полетов. На основе этих значений можно рассчитать интегральные показатели, характеризующие эффективность работы авиакомпании. Например, если большие самолеты, требующие длительного предполетного обслуживания, совершают много сравнительно непродолжительных перелетов, это, скорее всего, снижает экономическую эффективность компании.

8 Использование LATERAL во вложенных подзапросах

В документации (см. подраздел 7.2.1.5 «Подзапросы LATERAL») сказано: «Элемент LATERAL может находиться на верхнем уровне списка FROM или в дереве JOIN. В последнем случае он может также ссылаться на любые элементы в левой части JOIN, справа от которого он находится».

В качестве примера можно модифицировать запрос, рассмотренный в предыдущем упражнении. Здесь добавлены столбец ab_c и вложенный подзапрос ab_cs, ссылающийся на подзапрос abs, стоящий левее него в списке FROM.

```
SELECT
  abs.name,
  abs.data,
  round( abs.ab, 2 ) AS ab,
  round( abs.data * abs.ab,2 ) AS data_ab,
  round( abcs.abc, 2 ) AS abc,
  round( abs.data * abcs.abc, 2 ) AS data_abc,
  round( abcs.ab_c, 2 ) AS ab_c
FROM
    ( SELECT name, data, a * b AS ab
      FROM some_data
    ) AS abs
  JOIN LATERAL
    ( SELECT some_data.name, abs.ab / c AS abc, ab_c
      FROM some_data
        JOIN -- здесь можно без LATERAL
          ( SELECT name, abs.ab - c AS ab_c
            FROM some_data
          ) AS ab_cs
          ON ab_cs.name = abs.name
     ) AS abcs
     ON abcs.name = abs.name;
```

```
 name | data |  ab  | data_ab | abc  | data_abc | ab_c
------+------+------+---------+------+----------+------
 obj1 |   27 | 3.75 |  101.25 | 1.50 |    40.50 | 1.25
 obj2 |   63 | 6.13 |  385.88 | 2.23 |   140.32 | 3.38
 obj3 |   48 | 6.75 |  324.00 | 4.50 |   216.00 | 5.25
 obj4 |   19 | 3.19 |   60.56 | 1.42 |    26.92 | 0.94
 obj5 |   35 | 6.19 |  216.56 | 3.54 |   123.75 | 4.44
(5 строк)
```

Задание. Попробуйте проследить ход вычислений при выполнении этого запроса. Посмотрите план запроса при небольшом числе строк в таблице some_data и при значительно большем, как делали это в предыдущих экспериментах.

9 Функция JSON_TABLE

В разделе 4.3 «Тип JSON и конструкция LATERAL» (с. 236) решалась задача формирования номеров кресел для салонов самолетов с применением функции JSON_TABLE. В исходных данных для каждой модели самолета предусматривалась одна строка, содержащая два поля: код модели и описания планировок ее салонов, представленные в виде JSON-массива.

```
WITH seats_confs( aircraft_code, seats_conf ) AS
( VALUES
  ( 'SU9',
    '[ { "fare conditions": "Business",
         "rows": [ 1, 3 ],
         "letters": [ "A", "C", "D", "F" ] },
       { "fare conditions": "Economy",
         "rows": [ 4, 20 ],
         "letters": [ "A", "C", "D", "E", "F" ] }
    ]'::jsonb
  ),
  ( 'CN1',
    '[ { "fare conditions": "Economy",
         "rows": [ 1, 6 ],
         "letters": [ "A", "B" ] }
    ]'::jsonb
  )
)
...
```

Структуру исходных данных можно изменить, объединив два поля в единый JSON-объект. В него нужно поместить код модели самолета в качестве скалярного значения с ключом aircraft_code и JSON-массив описаний планировок с ключом confs:

```
WITH seats_confs( seats_conf ) AS
( VALUES
  ( '{ "aircraft code": "SU9",
       "confs": [ { "fare conditions": "Business",
                    "rows": [ 1, 3 ],
                    "letters": [ "A", "C", "D", "F" ] },
                  { "fare conditions": "Economy",
                    "rows": [ 4, 20 ],
                    "letters": [ "A", "C", "D", "E", "F" ] }
                ]
    }'::jsonb
  ),
  ( '{ "aircraft code": "CN1",
       "confs": [ { "fare conditions": "Economy",
                    "rows": [ 1, 6 ],
                    "letters": [ "A", "B" ] }
                ]
    }'::jsonb
  )
)
```

```
SELECT
  aircraft_code,
  fare_conditions,
  row || letter AS seat_no
FROM seats_confs AS sc,
     ...
     generate_series( row_from, row_to ) AS rows( row )
ORDER BY aircraft_code, row, letter;
```

Задание. Модифицируйте запрос с учетом изменившейся структуры данных.

Глава 5
Подпрограммы

SQL — декларативный язык, но функции и процедуры (объединяемые термином «подпрограммы», routines) дополняют его элементами императивного стиля программирования. PostgreSQL позволяет разрабатывать подпрограммы на различных языках. В этой главе используется только SQL, однако рассматриваемые концепции применимы и для других языков.

5.1. Базовые сведения о функциях

PostgreSQL изначально проектировался так, чтобы пользователь мог расширять функциональность СУБД своими собственными объектами, в том числе типами данных, операторами, агрегатами, пользовательскими функциями. При этом создаваемые объекты подключаются к серверу на лету. Такая гибкость обеспечивается тем, что в PostgreSQL значительная часть метаинформации не закодирована жестко, а хранится в специальных системных таблицах.

Функции можно разрабатывать на таких языках, как Python, Perl или PL/pgSQL, который является процедурным расширением PostgreSQL. Но можно создавать их и на языке SQL, возможностей которого зачастую вполне достаточно для решения весьма сложных задач. В этой главе мы покажем, как создавать пользовательские функции именно на SQL. Надо заметить, что многие детали правил объявления функций, передачи им параметров, формирования возвращаемых значений не зависят от языка, на котором функция написана. Поэтому материал данной главы очень важен и для понимания того, как вообще работают пользовательские функции.

Пользовательские функции создаются на стороне сервера и позволяют разработчику:

- декомпозировать задачу (наряду с подзапросами и общими табличными выражениями);

- повторно использовать серверный код в нескольких приложениях;
- упрощать программный код приложения, заменяя несколько запросов, выполняющих общую задачу, вызовом функции;
- избегать модификации приложения при изменении схемы базы данных или алгоритмов выполнения операций за счет сохранения программного интерфейса (API), предоставляемого функцией;
- расширять возможности сервера, создавая триггеры, операторы и другие объекты.

5.1.1. Создание функций

Начнем с простой задачи. Пусть необходимо подсчитать число мест в салоне конкретной модели самолета, соответствующих указанному классу обслуживания. Следующий запрос решает эту задачу для модели «Сухой Суперджет-100» и класса обслуживания Economy:

```
SELECT count( * )
FROM seats s
WHERE s.aircraft_code = 'SU9'
  AND s.fare_conditions = 'Economy';
 count
-------
    85
(1 строка)
```

Очевидно, что достаточно подставлять в этот запрос код модели и требуемый класс обслуживания, чтобы получать необходимую информацию. Но вместо этого можно написать функцию на языке SQL и передавать ей в качестве параметров те значения, которые нам пришлось бы каждый раз менять в запросе:

```
CREATE FUNCTION count_seats( a_code char( 3 ), fare_cond text )
RETURNS bigint AS
$$
  SELECT count( * )
  FROM seats s
  WHERE s.aircraft_code = a_code
    AND s.fare_conditions = fare_cond;
$$ LANGUAGE sql;
```

PostgreSQL ответит, что функция успешно создана:

```
CREATE FUNCTION
```

В приведенном примере в команде CREATE FUNCTION с помощью конструкции RETURNS задается тип возвращаемого значения, в данном случае — bigint. Мы использовали его, потому что функция count возвращает значение именно такого типа. Однако число мест заведомо не превысит 1000, поэтому вполне можно было бы использовать тип smallint.

Тело функции, представляющее собой символьную строку, предваряется ключевым словом AS. Строка заключается между символами-ограничителями, которыми могут служить одинарные кавычки или удвоенный символ доллара. Выбирая первый вариант, нужно учитывать, что если в теле функции присутствуют одинарные кавычки или символы обратной косой черты, их придется удваивать. Более детально этот вопрос рассмотрен в упражнении 4 (с. 373). Выбор ограничителей — дело вкуса, мы использовали $$.

Начиная с версии 14 PostgreSQL позволяет задать тело функции не только в виде строки, но и в виде блока SQL-команд или выражения в операторе RETURN. Примеры использования такого стиля и его особенности будут рассмотрены далее в этой главе.

В нашей функции всего один оператор. Она возвращает результат его выполнения, в данном случае — одну строку. Функция может содержать несколько операторов и возвращать множество строк, и такие ситуации мы также рассмотрим.

Завершается команда создания функции ключевым словом LANGUAGE с указанием языка, на котором она написана. В нашем случае это язык SQL.

Давайте проверим функцию в работе, вызвав ее с разными значениями параметров.

```
SELECT count_seats( '773', 'Business' );
 count_seats
-------------
          30
(1 строка)
```

```
SELECT count_seats( '763', 'Economy' );
 count_seats
-------------
         192
(1 строка)
```

Можно вызвать функцию не только в списке SELECT, но и в предложении FROM. Вообще, как заявлено в подразделе документации 36.5.8 «Функции SQL, порождающие таблицы», *все* функции, написанные на языке SQL, можно вызывать в предложении FROM. При этом фактически порождается таблица, к которой можно обращаться в запросе, как к обычной таблице. Если в объявлении функции сказано, что она возвращает базовый тип, например text или integer, то таблица, порождаемая ею, будет иметь всего один столбец. Однако функция может возвращать и таблицу с несколькими столбцами. Вскоре мы это покажем.

Поскольку мы не назначили функции никакого псевдонима, то заголовком столбца будет само имя функции:

```
SELECT * FROM count_seats( '763', 'Economy' );
 count_seats
-------------
         192
(1 строка)
```

Давайте назначим псевдоним и укажем при этом еще имя столбца:

```
SELECT * FROM count_seats( '763', 'Economy' ) AS c_seats( cs );
 cs
-----
 192
(1 строка)
```

А если задать только псевдоним, то столбец — он будет единственным — получит то же имя, что и таблица:

```
SELECT * FROM count_seats( '763', 'Economy' ) AS cs;
 cs
-----
 192
(1 строка)
```

Если число параметров функции велико, их порядок будет трудно запомнить. В таком случае можно воспользоваться нотацией с *именованными* аргументами. До этого мы применяли нотацию с *позиционными* аргументами.

Хотя наша функция имеет всего два параметра, воспользуемся для ее вызова нотацией с именованными аргументами, при этом переставим их местами:

```
SELECT count_seats( fare_cond => 'Business', a_code => 'SU9' );
 count_seats
-------------
          12
(1 строка)
```

Выше мы говорили, что PostgreSQL хранит в базе данных сведения обо всех объектах, созданных пользователем. Как увидеть информацию о функции count_seats? Для этого есть команда \df утилиты psql:

```
\df count_seats
Список функций
-[ RECORD 1 ]----------+---------------------------------
Схема                  | bookings
Имя                    | count_seats
Тип данных результата  | bigint
Типы данных аргументов | a_code character, fare_cond text
Тип                    | функ.
```

Обратите внимание, что в этом описании приведены не только типы данных, которые имеют параметры функции, но и имена этих параметров. При этом для типа character не указана длина, хотя мы ее задавали при создании функции. Как сказано в описании команды CREATE FUNCTION (см. подраздел «Замечания»), в объявлениях параметров функции и ее возвращаемого значения не учитываются модификаторы типа, задаваемые в скобках (например, поле точности для типа numeric).

В поле «Тип» указано функ., что означает функцию. Это поле может также иметь значение проц., но речь об этом пойдет в разделе 5.9 «Процедуры» (с. 367).

Чтобы увидеть описания всех функций, созданных пользователями в схеме bookings, нужно выполнить такую команду:

```
\df bookings.*
Список функций
-[ RECORD 1 ]----------+---------------------------------
Схема                  | bookings
Имя                    | count_seats
Тип данных результата  | bigint
Типы данных аргументов | a_code character, fare_cond text
Тип                    | функ.
-[ RECORD 2 ]----------+---------------------------------
Схема                  | bookings
Имя                    | lang
Тип данных результата  | text
Типы данных аргументов |
Тип                    | функ.
-[ RECORD 3 ]----------+---------------------------------
Схема                  | bookings
Имя                    | now
Тип данных результата  | timestamp with time zone
Типы данных аргументов |
Тип                    | функ.
```

Команда `\df` без параметра `bookings.*` выведет тот же список функций за исключением `now`. Это объясняется тем, что есть одноименная системная функция.

Можно увидеть и определение созданной функции, сохраненное в базе данных. Заметьте, что в нем произошел ряд изменений: тип `character` потерял модификатор — длину (поскольку при создании функций такие модификаторы отбрасываются), ограничитель строки `$$` был заменен на `$function$`, предложение `LANGUAGE` перенесено ближе к началу команды, а в самом ее начале добавлено предложение `OR REPLACE`. О нем мы вскоре расскажем.

```
\sf count_seats
CREATE OR REPLACE FUNCTION bookings.count_seats(a_code character, fare_cond text)
 RETURNS bigint
 LANGUAGE sql
AS $function$
  SELECT count( * )
  FROM seats s
  WHERE s.aircraft_code = a_code
    AND s.fare_conditions = fare_cond;
$function$
```

Параметры, которые мы передавали в функцию, имели имена, по которым к ним можно обращаться внутри тела функции. Существует еще один, более

старый способ обозначения параметров — по их порядковым номерам. Номеру должен предшествовать знак $. Давайте немного изменим команду создания функции. Обратите внимание, что в этой версии функции count_seats мы указали только типы данных для обоих параметров, но не задали их имен, поскольку они не будут нужны.

```
CREATE FUNCTION count_seats_2( char, text )
RETURNS bigint AS
$$
  SELECT count( * )
  FROM seats s
  WHERE s.aircraft_code = $1
    AND s.fare_conditions = $2;
$$ LANGUAGE sql;
CREATE FUNCTION
```

Эта функция работает так же, как и ее предшественница:

```
SELECT * FROM count_seats_2( 'SU9', 'Business' );
 count_seats_2
---------------
            12
(1 строка)
```

Конечно, теперь при вызове функции можно использовать только нотацию с позиционными аргументами, а применять именованные аргументы стало в принципе невозможно.

Параметры функций могут иметь модификаторы IN, OUT и INOUT. По умолчанию подразумевается IN, поэтому мы его и не указывали в нашей первой функции. Параметр с таким модификатором является *входным*, то есть служит для передачи внутрь функции какого-то значения, существующего вне ее.

Модификатор OUT означает, что параметр является *выходным*. Таких параметров у функции тоже может быть несколько. В случае отсутствия предложения RETURNS именно параметры с модификаторами OUT определят выходные значения, сформированные внутри функции. Таким способом она сможет вернуть несколько значений. Если же наряду с такими параметрами присутствует и предложение RETURNS, то типы данных этих параметров должны совпадать с типами, заданными в этом предложении. Это объясняется тем, что параметры с модификаторами OUT и предложение RETURNS не дополняют друг друга,

а по-разному определяют одну и ту же информацию. И в традиционных языках программирования, и в таких языках СУБД, как PL/SQL или Transact-SQL, возвращаемое значение и выходные параметры действуют независимо друг от друга. Но в PostgreSQL `OUT`-параметры и `RETURNS` — просто разные способы указать тип возвращаемого значения.

Модификатор `INOUT` дает возможность сочетать свойства входных и выходных параметров. Параметр с таким модификатором позволяет передать значение в функцию, а затем через него же вернуть значение (возможно, другое) вовне.

Все три типа модификаторов показаны на рис. 5.1.

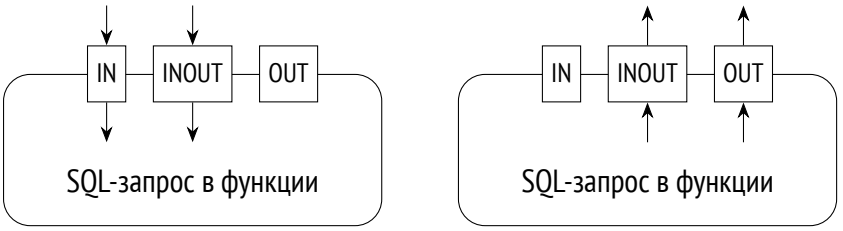


Рис. 5.1. *Входные и выходные параметры*

Давайте несколько модифицируем исходную задачу и напишем функцию, которая подсчитывает число мест для *каждого* класса обслуживания в салоне выбранной модели самолета. У первой версии нашей функции было два входных параметра: код модели самолета и класс обслуживания, а возвращала она одно числовое значение — количество мест. Новая версия функции будет иметь только один входной параметр.

Функция должна возвратить название модели самолета и число мест для каждого из трех классов обслуживания, поэтому нужно предусмотреть четыре параметра с модификатором `OUT`: предложение `RETURNS` позволяет указать тип только одного возвращаемого значения. Кроме этого, сформированный результат будет более полным и наглядным, если мы выведем не только название модели самолета, но и ее код, тот же самый, который передали в функцию с помощью параметра «код модели самолета».

Решить эту задачу можно было бы, предусмотрев еще один выходной параметр. Тогда заголовок функции стал бы таким:

```
CREATE FUNCTION count_seats_3
( IN a_code_in char DEFAULT 'SU9',
  OUT a_code_out char,
  OUT a_model text,
  OUT seats_business bigint,
  OUT seats_comfort bigint,
  OUT seats_economy bigint
) AS
...
```

Однако есть более простое решение: добавить к параметру для кода модели самолета модификатор INOUT.

```
CREATE FUNCTION count_seats_3
( INOUT a_code char DEFAULT 'SU9',
  OUT a_model text,
  OUT seats_business bigint,
  OUT seats_comfort bigint,
  OUT seats_economy bigint
) AS
$$
  SELECT
    a.aircraft_code,
    a.model,
    count( * ) FILTER ( WHERE s.fare_conditions = 'Business' ) AS business,
    count( * ) FILTER ( WHERE s.fare_conditions = 'Comfort' ) AS comfort,
    count( * ) FILTER ( WHERE s.fare_conditions = 'Economy' ) AS economy
  FROM aircrafts a
    JOIN seats s ON s.aircraft_code = a.aircraft_code
  WHERE a.aircraft_code = a_code
  GROUP BY a.aircraft_code, a.model
$$ LANGUAGE sql;
CREATE FUNCTION
```

Теперь проверим функцию в работе:

```
SELECT * FROM count_seats_3( '319' );
 a_code |     a_model      | seats_business | seats_comfort | seats_economy
--------+------------------+----------------+---------------+---------------
 319    | Аэробус A319-100 |             20 |             0 |            96
(1 строка)
```

Обратите внимание, что в качестве имен столбцов служат имена выходных параметров функции, то есть тех, которые имеют модификаторы OUT или INOUT,

а не те имена или псевдонимы столбцов, которые используются внутри функции в SQL-запросе. При этом параметры с модификатором OUT при вызове функции ей передавать не нужно, а параметр, имеющий модификатор INOUT, если только для него не задано значение по умолчанию, — нужно.

Для параметра a_code предусмотрено значение по умолчанию, которое мы задали с помощью ключевого слова DEFAULT. Вместо него можно было просто поставить знак равенства =. Таким образом, можно вызвать эту функцию вообще без аргументов:

```
SELECT * FROM count_seats_3();
 a_code |       a_model       | seats_business | seats_comfort | seats_economy
--------+---------------------+----------------+---------------+---------------
 SU9    | Сухой Суперджет-100 |             12 |             0 |            85
(1 строка)
```

Можно использовать функцию и так:

```
SELECT seats_business + seats_comfort + seats_economy AS total_seats
FROM count_seats_3( '319' );
 total_seats
-------------
         116
(1 строка)
```

В этом запросе мы не указали псевдонимы для столбцов, которые возвращает функция. В таком случае нам нужно знать те имена выходных параметров, которые были заданы при создании функции, чтобы воспользоваться ими.

При необходимости можно присвоить столбцам, возвращаемым функцией, другие имена (псевдонимы). В таком случае необходимо также задать псевдоним для таблицы, возвращаемой функцией:

```
SELECT model, code, seats_e, seats_b, seats_c
FROM count_seats_3( '319' ) AS cnt_seats( code, model, seats_b, seats_c, seats_e );
      model       | code | seats_e | seats_b | seats_c
------------------+------+---------+---------+---------
 Аэробус A319-100 | 319  |      96 |      20 |       0
(1 строка)
```

Можно получить результат выполнения функции и в виде значения так называемого *составного типа* (composite type), которое представлено как группа полей, заключенных в скобки:

```
SELECT cnt_seats
FROM count_seats_3( '319' ) AS cnt_seats;
            cnt_seats
---------------------------------
 (319,"Аэробус A319-100",20,0,96)
(1 строка)
```

В этой книге составные типы подробно не рассматриваются. В документации им посвящен раздел 8.16 «Составные типы». А сейчас мы только расскажем об их роли в качестве возвращаемого значения функции.

В PostgreSQL система типов данных устроена так, что когда создается таблица, вместе с ней создается и новый составной тип данных, имя которого совпадает с ее именем, а поля этого типа соответствуют столбцам таблицы. Составной тип, по сути, является списком имен полей и их типов данных; таким образом, для таблицы он является описателем структуры ее строк. Для возвращаемого функцией результата тоже автоматически создается составной тип, но анонимный.

Давайте реализуем вариант с возвратом значения (предварительно созданного) составного типа, взяв за основу функцию `count_seats_3` и модифицировав ее соответствующим образом. Такой способ с созданием специального составного типа может иметь смысл, если этот тип может понадобиться и в других ситуациях.

```
CREATE TYPE cnt_seats AS
( aircraft_code char( 3 ),
  model text,
  seats_business bigint,
  seats_comfort bigint,
  seats_economy bigint
);
CREATE TYPE
```

Тело функции останется без изменений, изменятся только список параметров и возвращаемое значение.

```
CREATE OR REPLACE FUNCTION count_seats_4( a_code char DEFAULT 'SU9' )
RETURNS cnt_seats AS
$$
  SELECT
    a.aircraft_code,
    a.model,
    count( * ) FILTER ( WHERE s.fare_conditions = 'Business' ) AS business,
    count( * ) FILTER ( WHERE s.fare_conditions = 'Comfort' ) AS comfort,
    count( * ) FILTER ( WHERE s.fare_conditions = 'Economy' ) AS economy
  FROM aircrafts a
    JOIN seats s ON s.aircraft_code = a.aircraft_code
  WHERE a.aircraft_code = a_code
  GROUP BY a.aircraft_code, a.model
$$ LANGUAGE sql;
CREATE FUNCTION
```

Обратите внимание, что в этой версии функции параметр a_code стал входным. Поскольку код модели самолета теперь входит в состав созданного нами типа, возвращать этот код в качестве отдельного элемента данных не требуется. Новая версия функции показывает такой же результат, что и предыдущая:

```
SELECT cnt_seats FROM count_seats_4( '319' ) AS cnt_seats;
            cnt_seats
---------------------------------
 (319,"Аэробус A319-100",20,0,96)
(1 строка)
```

Точно так же можно получить результат и в виде отдельных столбцов. Обратите внимание, что их имена определяются именами полей составного типа.

```
SELECT * FROM count_seats_4( '773' );
 aircraft_code |     model     | seats_business | seats_comfort | seats_economy
---------------+---------------+----------------+---------------+---------------
 773           | Боинг 777-300 |             30 |            48 |           324
(1 строка)
```

Давайте поместим вызов функции непосредственно в список SELECT:

```
SELECT count_seats_4();
            count_seats_4
-----------------------------------
 (SU9,"Сухой Суперджет-100",12,0,85)
(1 строка)
```

Функция возвращает значение составного типа; отдельное его поле можно получить, используя любую из двух нотаций, описанных в подразделе документации 36.5.3 «Функции SQL с составными типами». Первая из них требует наличия дополнительных скобок во избежание неоднозначности при разборе запроса:

```
SELECT ( count_seats_4( '319' ) ).seats_business;
 seats_business
----------------
             20
(1 строка)
```

Другая нотация называется *функциональной*. Обратите внимание на взаимное расположение имени функции и имени поля составного значения:

```
SELECT seats_business( count_seats_4( '319' ) );
 seats_business
----------------
             20
(1 строка)
```

Как мы увидели в ходе экспериментов, при вызове функции в предложении FROM результат можно вывести как в виде значения составного типа, так и в виде отдельных столбцов. Это показывает, что строка таблицы, по сути, и является значением составного типа.

Таким образом, чтобы возвратить из функции более одного значения, можно либо воспользоваться модификаторами OUT, либо создать составной тип и указать его в качестве типа возвращаемого значения. Можно также использовать в этом качестве тип record. О нем мы расскажем в подразделе 5.1.2 «Перегрузка функций» (с. 275), а его использование покажем в разделе 5.3 «Функции, возвращающие множества строк» (с. 298).

5.1.2. Перегрузка функций

Иногда бывает удобно, чтобы новая функция имела *то же имя*, что и уже существующая. Тогда конкретная версия функции будет вызываться по одному и тому же имени, но в зависимости от числа и типов данных переданных аргументов.

Реализовать это пожелание можно с помощью *перегрузки функций*. Функции, написанные на языке SQL, можно перегружать аналогично тому, как это делается в других языках программирования (например, C++). У перегруженной функции будет такое же имя, но другой набор входных параметров, то есть параметров с модификатором IN, INOUT или вовсе без модификатора. При этом имена параметров не играют роли. Дополнительные сведения по этому вопросу можно найти в разделе документации 36.6 «Перегрузка функций».

Как сказано в подразделе документации 36.5.4 «Функции SQL с выходными параметрами», сервер определяет, какую функцию вызвать, на основе ее *сигнатуры*, то есть имени функции и типов данных ее входных параметров, а параметры с модификатором OUT в расчет не принимаются. Сигнатура должна быть уникальной. Поэтому важен порядок, в котором типы данных следуют в списке входных параметров. Например, следующие две перегруженные функции будут успешно созданы, поскольку у них разные сигнатуры: хотя типы данных их параметров одинаковые, но порядок их следования различается.

```
CREATE OR REPLACE FUNCTION test( op text, op2 int )
RETURNS text AS
$$
  SELECT op || op2 || ' (1)';
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

```
CREATE OR REPLACE FUNCTION test( op int, op2 text )
RETURNS text AS
$$
  SELECT op || op2 || ' (2)';
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

В следующем запросе будет вызвана первая из этих функций:

```
SELECT test( 'SU', 9 );
```

```
  test
---------
 SU9 (1)
(1 строка)
```

Давайте вернемся к функции, подсчитывающей число кресел, и перегрузим ее самую первую версию. Для этого сделаем копию функции count_seats_3 и изменим имя на count_seats:

```
CREATE FUNCTION count_seats
( INOUT a_code char DEFAULT 'SU9',
  OUT a_model text,
  OUT seats_business bigint,
  OUT seats_comfort bigint,
  OUT seats_economy bigint
) AS
$$
  SELECT
    a.aircraft_code,
    a.model,
    count( * ) FILTER ( WHERE s.fare_conditions = 'Business' ) AS business,
    count( * ) FILTER ( WHERE s.fare_conditions = 'Comfort' ) AS comfort,
    count( * ) FILTER ( WHERE s.fare_conditions = 'Economy' ) AS economy
  FROM aircrafts a
    JOIN seats s ON s.aircraft_code = a.aircraft_code
  WHERE a.aircraft_code = a_code
  GROUP BY a.aircraft_code, a.model
$$ LANGUAGE sql;
CREATE FUNCTION
```

Передав функции два аргумента, мы увидим, что была вызвана первая версия:

```
SELECT * FROM count_seats( '773', 'Business' );
 count_seats
-------------
          30
(1 строка)
```

С одним аргументом отрабатывает вторая версия функции:

```
SELECT * FROM count_seats( '773' );
 a_code |    a_model    | seats_business | seats_comfort | seats_economy
--------+---------------+----------------+---------------+---------------
 773    | Боинг 777-300 |             30 |            48 |           324
(1 строка)
```

При вызове функции без аргументов также выполняется ее вторая версия:

```
SELECT * FROM count_seats();
 a_code |       a_model       | seats_business | seats_comfort | seats_economy
--------+---------------------+----------------+---------------+---------------
 SU9    | Сухой Суперджет-100 |             12 |             0 |            85
(1 строка)
```

Правила, которыми руководствуется сервер для определения вызываемой функции (не обязательно перегруженной), подробно описаны в разделе документации 10.3 «Функции».

Если параметры имеют значения по умолчанию, могут возникать проблемы с определением вызываемой перегруженной функции. Этот вопрос рассмотрен в упражнении 5 (с. 374).

Давайте посмотрим, какие сведения о созданных нами перегруженных функциях есть в базе данных:

```
\df count_seats
Список функций
-[ RECORD 1 ]----------+---------------------------------------------------------------
Схема                  | bookings
Имя                    | count_seats
Тип данных результата  | bigint
Типы данных аргументов | a_code character, fare_cond text
Тип                    | функ.
-[ RECORD 2 ]----------+---------------------------------------------------------------
Схема                  | bookings
Имя                    | count_seats
Тип данных результата  | record
Типы данных аргументов | INOUT a_code character DEFAULT 'SU9'::bpchar, OUT a_model tex...
Тип                    | функ.
```

Обратите внимание, что типом возвращаемого результата новой — перегруженной — версии функции `count_seats` будет `record` (запись), хотя мы не задавали этого явно при ее создании. Тип `record` относится к так называемым *псевдотипам*, которые представлены в разделе документации 8.21 «Псевдотипы». Использовать его в качестве типа данных столбца таблицы нельзя, но он может служить в качестве типа данных результата функции.

В этом разделе документации сказано: тип `record` указывает, что функция принимает или возвращает неопределенный тип строки (таблицы). Можно сказать, что `record` — это составной тип с неопределенным составом полей. Для придания ему определенности нужно либо предусматривать у функции выходные параметры (что мы уже показали), либо при ее вызове в запросе задавать не только псевдоним формируемой таблицы, но и определения (имена и типы данных) ее столбцов. Использование `record` в качестве типа возвращаемого значения мы покажем в разделе 5.3 «Функции, возвращающие множества строк» (с. 298).

Надо сказать, что SQL-функция может возвращать значение типа record, а вот принимать параметр такого типа не может.

В упомянутом разделе документации представлено много других псевдотипов. Далее в этой главе мы покажем еще использование псевдотипа void.

5.1.3. Удаление функций

Как и другие объекты базы данных, функции можно удалять. Для этого служит команда DROP FUNCTION. В ней достаточно указать только имя функции и типы данных, которые имеют ее параметры, а их имена можно не указывать. При этом можно опускать модификаторы типов (например, не задавать длину для параметра типа char), поскольку они отбрасываются при создании функций.

Попробуем удалить вторую из двух перегруженных функций count_seats:

```
DROP FUNCTION count_seats( char, text, bigint, bigint, bigint );
ОШИБКА:  функция count_seats(character, text, bigint, bigint, bigint) не существует
```

Ошибка объясняется тем, что в команде удаления функции нужно задать только входные параметры (в том числе и имеющие модификатор INOUT), а параметры, имеющие модификатор OUT, указывать не нужно. Включив в команду выходные параметры, мы фактически указали функцию с другой сигнатурой, а такой функции в нашей базе данных нет.

Правильная команда будет такой:

```
DROP FUNCTION count_seats( char );
DROP FUNCTION
```

Действительно, осталась только первая из двух функций count_seats:

```
\df count_seats
Список функций
-[ RECORD 1 ]----------+---------------------------------
Схема                  | bookings
Имя                    | count_seats
Тип данных результата  | bigint
Типы данных аргументов | a_code character, fare_cond text
Тип                    | функ.
```

Имя функции count_seats теперь стало уникальным в базе данных, и ее можно удалить, не указывая параметры в команде удаления. При этом имя функции нужно написать без круглых скобок, поскольку иначе PostgreSQL будет искать функцию без параметров, а наша функция их имеет:

```
DROP FUNCTION count_seats;
DROP FUNCTION
```

Как и в командах удаления других объектов базы данных, в команде DROP FUNCTION можно использовать необязательное предложение IF EXISTS. Оно позволяет избежать вывода сообщения об ошибке, если удаляемая функция не существует. Сравните:

```
DROP FUNCTION count_seats( char );
ОШИБКА:  функция count_seats(character) не существует

DROP FUNCTION IF EXISTS count_seats( char );
ЗАМЕЧАНИЕ:  функция count_seats(pg_catalog.bpchar) не существует, пропускается
DROP FUNCTION
```

Замена сообщения об ошибке на замечание может быть полезной в скриптах, выполняемых в среде утилиты psql. В случае, когда ее встроенная переменная ON_ERROR_STOP имеет истинное значение (ON), выполнение скрипта прерывается после первой ошибки. Предложение IF EXISTS позволяет и в таких условиях продолжить выполнение скрипта. Подробно использование скриптов рассмотрено в описании утилиты psql, приведенном в документации (см. часть VI «Справочное руководство», раздел II «Клиентские приложения PostgreSQL»).

5.1.4. Функции, включающие несколько SQL-команд

Все созданные нами функции содержали только одну SQL-команду, однако в общем случае их может быть несколько. Давайте напишем функцию, записывающую в базу данных сведения о продаже билета. Поскольку билет (и перелеты) оформляется в операции бронирования, функция должна добавить строки в таблицы «Бронирования» (bookings), «Билеты» (tickets) и «Перелеты» (ticket_flights).

У функции будет целый ряд параметров. Возвращать она будет полную сумму бронирования. Конечно, эта функция предлагает упрощенное решение, тем не менее она позволяет при многократном вызове оформить в одном бронировании несколько билетов с несколькими перелетами в каждом.

```
CREATE FUNCTION make_or_update_booking
( b_ref char,      -- номер бронирования
  t_no char,       -- номер билета
  p_id varchar,    -- идентификатор пассажира
  p_name text,     -- имя пассажира
  c_data jsonb,    -- контактные данные пассажира
  f_id integer,    -- идентификатор рейса
  f_cond varchar, -- класс обслуживания
  amt numeric      -- стоимость перелета
)
RETURNS numeric AS
$$
  INSERT INTO bookings ( book_ref, book_date, total_amount )
  VALUES ( b_ref, bookings.now(), 0 )
    ON CONFLICT DO NOTHING;

  INSERT INTO tickets
  ( ticket_no, book_ref, passenger_id, passenger_name, contact_data )
  VALUES ( t_no, b_ref, p_id, p_name, c_data )
    ON CONFLICT DO NOTHING;

  INSERT INTO ticket_flights ( ticket_no, flight_id, fare_conditions, amount )
  VALUES ( t_no, f_id, f_cond, amt );

  UPDATE bookings
  SET total_amount = total_amount + amt
  WHERE book_ref = b_ref
  RETURNING total_amount;
$$
LANGUAGE sql;
```

```
CREATE FUNCTION
```

Обратите внимание на предложение RETURNING команды UPDATE. Именно благодаря ему функция возвращает полную сумму бронирования.

Давайте оформим билет на рейс Нижний Новгород — Пермь. Операции будем выполнять в транзакции.

```
BEGIN;
```

```
BEGIN
```

```
SELECT make_or_update_booking
( b_ref => 'ABC123',
  t_no => '1234567890123',
  p_id => '0000 123456',
  p_name => 'IVAN PETROV',
  c_data => '{"phone": "+7(123)4567890" }'::jsonb,
  f_id => 20502, f_cond => 'Economy',
  amt => 10000::numeric
) AS total_amount;
 total_amount
--------------
     10000.00
(1 строка)

SELECT * FROM bookings
WHERE book_ref = 'ABC123';
 book_ref |       book_date        | total_amount
----------+------------------------+--------------
 ABC123   | 2017-08-15 22:00:00+07 |     10000.00
(1 строка)

SELECT * FROM tickets
WHERE ticket_no = '1234567890123';
   ticket_no   | book_ref | passenger_id | passenger_name |        contact_data
---------------+----------+--------------+----------------+----------------------------
 1234567890123 | ABC123   | 0000 123456  | IVAN PETROV    | {"phone": "+7(123)4567890"}
(1 строка)

SELECT * FROM ticket_flights
WHERE ticket_no = '1234567890123'
  AND flight_id = 20502;
   ticket_no   | flight_id | fare_conditions |  amount
---------------+-----------+-----------------+----------
 1234567890123 |     20502 | Economy         | 10000.00
(1 строка)

ROLLBACK;
ROLLBACK
```

Таким образом, функция возвращает результат выполнения своего последнего оператора. Если он порождает только одну строку, она и будет возвращена в качестве результата. Если же оператор порождает множество строк, то будет возвращена первая из них. Нужно учитывать, что без использования предложения ORDER BY первой строкой станет случайная строка.

Для того чтобы функция могла возвратить множество строк, нужно предусмотреть специальные меры. Это будет показано в разделе 5.3 «Функции, возвращающие множества строк» (с. 298).

Если последний запрос функции не сформирует ни одной строки, она возвратит NULL.

5.1.5. Функции в стиле стандарта SQL

Тело каждой функции, созданной нами до сих пор, представляло собой строковую константу. Однако стандарт языка SQL предлагает другой способ, который можно назвать «функция в стиле стандарта SQL». При использовании этого способа тело функции представляет собой блок операторов, входящих в состав функции, или одно выражение, возвращаемое оператором RETURN.

Между этими способами есть ряд принципиальных различий. При представлении тела функции в виде строки разбор производится в процессе выполнения функции, а при использовании SQL-стиля — в процессе ее определения. В результате становится возможным отслеживание *зависимостей* между функцией и другими объектами базы данных. Речь об этом пойдет в разделе 5.2 «Функции и зависимости между объектами базы данных» (с. 288). В описании команды CREATE FUNCTION сказано, что SQL-стиль более совместим со стандартом языка SQL и другими его реализациями. Однако этот стиль применим только к функциям на языке SQL, а функции с телом в виде строковой константы можно писать на всех поддерживаемых языках.

Давайте создадим версию нашей первой функции count_seats в новом стиле. Добавим к ее имени суффикс _sql:

```
CREATE FUNCTION count_seats_sql( a_code char, fare_cond text )
RETURNS bigint
RETURN
( SELECT count( * )
  FROM seats s
  WHERE s.aircraft_code = a_code
    AND s.fare_conditions = fare_cond
);
CREATE FUNCTION
```

Язык указывать необязательно, ведь поддерживается только SQL.

Функция сохранилась в базе данных:

```
\sf count_seats_sql
CREATE OR REPLACE FUNCTION bookings.count_seats_sql(a_code character, fare_cond text)
 RETURNS bigint
 LANGUAGE sql
RETURN (SELECT count(*) AS count FROM seats s WHERE ((s.aircraft_code =
count_seats_sql.a_code) AND ((s.fare_conditions)::text = count_seats_sql.fare_cond)))
```

Вызвать эту функцию можно так же, как и прежде. Результат будет ожидаемым:

```
SELECT count_seats_sql( 'SU9', 'Economy' );
 count_seats_sql
-----------------
              85
(1 строка)
```

Теперь давайте создадим и версию функции make_or_update_booking в стиле стандарта SQL. Обратите внимание на конструкцию BEGIN ATOMIC ... END:

```
CREATE FUNCTION make_or_update_booking_sql
( b_ref char,
  t_no char,
  p_id varchar,
  p_name text,
  c_data jsonb,
  f_id integer,
  f_cond varchar,
  amt numeric
)
RETURNS numeric
BEGIN ATOMIC
  INSERT INTO bookings ( book_ref, book_date, total_amount )
  VALUES ( b_ref, bookings.now(), 0 )
    ON CONFLICT DO NOTHING;

  INSERT INTO tickets
  ( ticket_no, book_ref, passenger_id, passenger_name, contact_data )
  VALUES ( t_no, b_ref, p_id, p_name, c_data )
    ON CONFLICT DO NOTHING;

  INSERT INTO ticket_flights ( ticket_no, flight_id, fare_conditions, amount )
  VALUES ( t_no, f_id, f_cond, amt );
```

```
  UPDATE bookings
  SET total_amount = total_amount + amt
  WHERE book_ref = b_ref
  RETURNING total_amount;
END;
CREATE FUNCTION
```

Выше мы говорили, что разбор такой функции производится еще на стадии ее определения. Результат разбора можно увидеть в системном каталоге pg_proc. При этом для функции с телом в виде строковой константы в столбце prosqlbody (в выборке это псевдоним parsed_code) будет стоять NULL, а для функции в SQL-стиле NULL будет в столбце prosrc (в выборке — source_code).

```
SELECT
  proname,
  left( prosrc, 74 ) AS source_code,
  left( prosqlbody, 74 ) AS parsed_code
FROM pg_proc
WHERE proname ~ 'make_or_update_booking' \gx
-[ RECORD 1 ]-------------------------------------------------------------------
proname     | make_or_update_booking
source_code |                                                                  +
            |   INSERT INTO bookings ( book_ref, book_date, total_amount )     +
            |   VALUES ( b
parsed_code |
-[ RECORD 2 ]-------------------------------------------------------------------
proname     | make_or_update_booking_sql
source_code |
parsed_code | (({QUERY :commandType 3 :querySource 0 :canSetTag true :utilityStmt <> :re
```

Читатель может самостоятельно проверить работу этой версии функции.

5.1.6. Значения NULL в качестве аргументов функции

Функция может получить в качестве аргумента значение NULL, и разработчик должен позаботиться о том, чтобы это не приводило к сбоям. По умолчанию обработка таких значений возлагается на саму функцию. Однако если функция заведомо должна возвратить NULL при получении хотя бы одного неопределенного значения, можно явно указать это в команде CREATE FUNCTION или ALTER

FUNCTION с помощью предложения STRICT. Тогда при выполнении запроса функция не исполняется вовсе, а ее результат автоматически принимает значение NULL, позволяя разработчику упростить код.

В качестве примера воспользуемся функцией count_seats_sql, разработанной в подразделе 5.1.5 «Функции в стиле стандарта SQL» (с. 283). Давайте вызовем ее с аргументом NULL. Мы должны явно указать аргументы: если написать просто count_seats_sql(), PostgreSQL выдаст ошибку, поскольку будет искать одноименную перегруженную функцию без параметров и, естественно, не найдет:

```
SELECT count_seats_sql( 'SU9', NULL );
 count_seats_sql
-----------------
               0
(1 строка)
```

Посмотрим план запроса:

```
                QUERY PLAN
-----------------------------------
 Result (actual rows=1 loops=1)
 Planning Time: 0.051 ms
 Execution Time: 0.164 ms
(3 строки)
```

Давайте удалим функцию и создадим ее вновь, дополнив код предложением STRICT:

```
DROP FUNCTION count_seats_sql;
DROP FUNCTION
CREATE FUNCTION count_seats_sql( a_code char, fare_cond text )
RETURNS bigint
STRICT
RETURN
( SELECT count( * )
  FROM seats s
  WHERE s.aircraft_code = a_code
    AND s.fare_conditions = fare_cond
);
CREATE FUNCTION
```

Теперь функция возвращает NULL, получив неопределенный аргумент:

```
SELECT count_seats_sql( 'SU9', NULL );
 count_seats_sql
-----------------

(1 строка)
```

План покажет, что время исполнения сократилось на порядок, и даже планироваться запрос стал быстрее, ведь теперь результат вычисляется без вызова функции:

```
           QUERY PLAN
--------------------------------
 Result (actual rows=1 loops=1)
 Planning Time: 0.012 ms
 Execution Time: 0.007 ms
(3 строки)
```

При необходимости можно вернуть способ обработки, принятый по умолчанию (при создании функции его указывать не обязательно):

```
ALTER FUNCTION count_seats_sql
  CALLED ON NULL INPUT;
ALTER FUNCTION
```

Существует альтернативный вариант предложения STRICT. Давайте покажем его на примере команды ALTER FUNCTION. Обратите внимание, что здесь ключевое слово RETURNS не имеет ничего общего с одноименным предложением, задающим тип возвращаемого значения функции.

```
ALTER FUNCTION count_seats_sql
  RETURNS NULL ON NULL INPUT;
ALTER FUNCTION
```

Проверить, изменились ли характеристики функции, можно в уже известном системном каталоге pg_proc:

```
SELECT proisstrict
FROM  pg_proc
WHERE proname = 'count_seats_sql';
 proisstrict
-------------
 t
(1 строка)
```

5.2. Функции и зависимости между объектами базы данных

Для существования и успешного использования одних объектов базы данных необходимо наличие других, то есть объекты в базе данных *зависят* друг от друга. При удалении объектов связи между ними разрываются, что может приводить к ошибкам в процессе выполнения запросов. Функции также участвуют в связях с другими объектами, и нам необходимо рассмотреть два вопроса:

- что происходит с другими объектами, когда удаляется функция, на которую они ссылаются (например, функция используется в ограничении `CHECK` таблицы);
- что происходит с функцией, когда удаляются объекты, которые используются в ней.

Сведения об объектах базы данных хранятся в системных каталогах. Их описания приведены в документации в главе 51 «Системные каталоги». Каждый объект в базе данных имеет так называемый *идентификатор объекта* (Object Identifier, OID), который представляется значением типа `oid` и используется в качестве первичного ключа соответствующего системного каталога.

В PostgreSQL насчитывается несколько десятков системных каталогов, из которых нас будут интересовать следующие:

- `pg_class` — описывает таблицы, индексы, последовательности, представления (в том числе материализованные), составные типы и другие объекты, объединяемые термином «отношение»;
- `pg_proc` — хранит сведения о функциях (в том числе оконных и агрегатных) и процедурах, то есть о подпрограммах;
- `pg_constraint` — содержит ограничения первичных, уникальных и внешних ключей, ограничения `CHECK` и ряд других ограничений;
- `pg_depend` — описывает зависимости между объектами базы данных.

Каталог pg_depend содержит ряд столбцов, в том числе:

- classid (ссылается на pg_class.oid) — OID системного каталога, в котором находится зависимый объект;
- objid (ссылается на какой-либо столбец OID) — OID конкретного зависимого объекта;
- refclassid (ссылается на pg_class.oid) — OID системного каталога, в котором находится вышестоящий объект;
- refobjid (ссылается на какой-либо столбец OID) — OID конкретного вышестоящего объекта;
- deptype — код, определяющий вид зависимости.

Факт наличия записи в каталоге pg_depend говорит о том, что нельзя удалить вышестоящий объект, не удалив также и зависимый (подчиненный). Есть несколько вариантов зависимости, задаваемых в столбце deptype. Они представлены в документации в подразделе 51.18 «pg_depend». Мы будем учитывать только два из них:

- Обычная зависимость (n) — зависимость между объектами, создаваемыми отдельно. Зависимый объект можно удалить, не затрагивая вышестоящий. Однако вышестоящий объект (при наличии зависимого) можно удалить, только добавив предложение CASCADE в команду удаления. При этом зависимый объект также будет удален. В качестве примера можно привести зависимость столбца таблицы от его типа данных или внешнего ключа от таблицы, на которую он ссылается.
- Автоматическая зависимость (a) — зависимый объект, как и в предыдущем варианте зависимости, можно удалить, не затрагивая вышестоящий. Однако при удалении вышестоящего объекта зависимый должен быть удален автоматически, независимо от наличия предложения CASCADE в команде удаления. В качестве примера может служить именованное ограничение, наложенное на таблицу, которое автоматически удаляется при ее удалении.

Давайте создадим представление для расшифровки информации из системного каталога pg_depend и последующие запросы будем строить на его основе:

```
CREATE VIEW pg_depend_v AS
( SELECT
    objid,
    classid::regclass::text AS classname,
    CASE classid
      WHEN 'pg_proc'::regclass THEN objid::regproc::text
      WHEN 'pg_class'::regclass THEN objid::regclass::text
      WHEN 'pg_constraint'::regclass
        THEN ( SELECT conname FROM pg_constraint WHERE oid = pg_depend.objid )
    END AS objname,
    refclassid::regclass::text AS refclassname,
    refobjid,
    CASE refclassid
      WHEN 'pg_proc'::regclass THEN refobjid::regproc::text
      WHEN 'pg_class'::regclass THEN refobjid::regclass::text
      WHEN 'pg_constraint'::regclass
        THEN ( SELECT conname FROM pg_constraint WHERE oid = pg_depend.refobjid )
    END AS refobjname,
    CASE deptype
      WHEN 'n' THEN 'normal'
      WHEN 'a' THEN 'auto'
      ELSE 'other'
    END AS deptype
  FROM pg_depend
);
CREATE VIEW
```

При обращении к системным каталогам их можно соединять с помощью оператора JOIN, как и обычные таблицы. Однако для упрощения запросов (для уменьшения количества явных соединений) можно использовать различные типы-псевдонимы для типа oid. Все они описаны в разделе документации 8.19 «Идентификаторы объектов». Мы воспользовались типами regclass (ссылается на каталог pg_class) и regproc (ссылается на каталог pg_proc).

Описание зависимого объекта из pg_proc получаем по его OID из каталога pg_proc с помощью операции приведения типа objid::regproc::text. Описание объекта из pg_class получаем по его OID из каталога pg_class с помощью операции приведения типа objid::regclass::text. Для каталога pg_constraint псевдоним типа oid не предусмотрен, поэтому приходится использовать подзапрос.

Для получения сведений о вышестоящем объекте, идентификатор которого берется из столбца refobjid, используются аналогичные приемы.

Более подробно использование системных каталогов для поиска зависимостей между объектами базы данных, а также применение типов-псевдонимов будут рассмотрены в упражнении 24 (с. 414).

5.2.1. Зависимость объектов базы данных от функций

Начнем с рассмотрения зависимости объектов от функции. В этом случае не важно, представлено ли ее тело в виде символьной строки или она написана в стиле стандарта SQL.

Создадим временную таблицу:

```
CREATE TEMP TABLE aircrafts_tmp AS SELECT * FROM aircrafts;
SELECT 9
```

Создадим функцию для проверки допустимости кода модели самолета. Конечно, эта функция используется только для иллюстрации рассматриваемых концепций. В реальной работе можно было бы обойтись проверкой непосредственно в ограничении CHECK.

```
CREATE FUNCTION air_code_correct( a_code char )
RETURNS boolean
RETURN a_code ~ '^[0-9A-Z]{3}$';
CREATE FUNCTION
```

Добавим ограничение, основанное на этой функции:

```
ALTER TABLE aircrafts_tmp ADD CHECK ( air_code_correct( aircraft_code ) );
ALTER TABLE
```

Вот это ограничение:

```
\d aircrafts_tmp
                        Таблица "pg_temp_11.aircrafts_tmp"
    Столбец    |     Тип      | Правило сортировки | Допустимость NULL | По умолчанию
---------------+--------------+--------------------+-------------------+--------------
 aircraft_code | character(3) |                    |                   |
 model         | text         |                    |                   |
 range         | integer      |                    |                   |
Ограничения-проверки:
    "aircrafts_tmp_aircraft_code_check" CHECK (air_code_correct(aircraft_code))
```

Оно работает:

```
INSERT INTO aircrafts_tmp
VALUES ( '96', 'Ильюшин ИЛ96-300', 10000 );
ОШИБКА:  новая строка в отношении "aircrafts_tmp" нарушает ограничение-проверку
"aircrafts_tmp_aircraft_code_check"
ПОДРОБНОСТИ:  Ошибочная строка содержит (96 , Ильюшин ИЛ96-300, 10000).
```

Сначала посмотрим, какие ограничения зависят от таблицы aircrafts_tmp:

```
SELECT objname AS dependent_object, refobjname AS referenced_object, deptype
FROM pg_depend_v
WHERE classname = 'pg_constraint'
  AND refobjname = 'aircrafts_tmp';
         dependent_object          | referenced_object | deptype
-----------------------------------+-------------------+---------
 aircrafts_tmp_aircraft_code_check | aircrafts_tmp     | auto
 aircrafts_tmp_aircraft_code_check | aircrafts_tmp     | normal
(2 строки)
```

Между двумя объектами может существовать и более одной связи, как в нашем случае.

Теперь выясним, какие ограничения зависят от функции air_code_correct:

```
SELECT objname AS dependent_object, refobjname AS referenced_object, deptype
FROM pg_depend_v
WHERE classname = 'pg_constraint'
  AND refobjname = 'air_code_correct';
         dependent_object          | referenced_object | deptype
-----------------------------------+-------------------+---------
 aircrafts_tmp_aircraft_code_check | air_code_correct  | normal
(1 строка)
```

Попробуем удалить функцию:

```
DROP FUNCTION air_code_correct;
ОШИБКА:  удалить объект функция air_code_correct(character) нельзя, так как от него
зависят другие объекты
ПОДРОБНОСТИ:  ограничение aircrafts_tmp_aircraft_code_check в отношении таблица
aircrafts_tmp зависит от объекта функция air_code_correct(character)
ПОДСКАЗКА:  Для удаления зависимых объектов используйте DROP ... CASCADE.
```

Здесь зависимость между функцией и ограничением обычная (normal), поэтому предлагается добавить в команду предложение CASCADE. Так и сделаем:

```
DROP FUNCTION air_code_correct CASCADE;
ЗАМЕЧАНИЕ:  удаление распространяется на объект ограничение
aircrafts_tmp_aircraft_code_check в отношении таблица aircrafts_tmp
DROP FUNCTION
```

Ограничения теперь нет:

```
\d aircrafts_tmp
                        Таблица "pg_temp_11.aircrafts_tmp"
    Столбец    |     Тип      | Правило сортировки | Допустимость NULL | По умолчанию
---------------+--------------+--------------------+-------------------+--------------
 aircraft_code | character(3) |                    |                   |
 model         | text         |                    |                   |
 range         | integer      |                    |                   |
```

Если мы воссоздадим функцию, ограничение восстановится?

```
CREATE OR REPLACE FUNCTION air_code_correct( a_code char )
RETURNS bool
LANGUAGE sql
RETURN a_code ~ '^[0-9A-Z]{3}$';
CREATE FUNCTION
```

Нет, оно не восстановилось:

```
\d aircrafts_tmp
                        Таблица "pg_temp_11.aircrafts_tmp"
    Столбец    |     Тип      | Правило сортировки | Допустимость NULL | По умолчанию
---------------+--------------+--------------------+-------------------+--------------
 aircraft_code | character(3) |                    |                   |
 model         | text         |                    |                   |
 range         | integer      |                    |                   |
```

В представленной команде есть важное отличие от предыдущей команды создания функции, а именно предложение OR REPLACE. Его наличие позволяет заменить тело существующей функции, не прибегая к помощи команды DROP FUNCTION. Важно, что удаление существующей функции с помощью команды DROP FUNCTION и последующее создание функции с той же сигнатурой (то есть с тем же именем и с теми же входными параметрами) с помощью команды

CREATE FUNCTION *не тождественно* использованию только лишь команды CREATE FUNCTION, дополненной предложением OR REPLACE. В первом случае воссозданная функция будет являться уже *другой сущностью*, отличной от первоначальной. Функцию нельзя удалить, не удалив при этом и ссылающиеся на нее другие объекты. Мы убедились в этом на примере ограничения целостности CHECK, в котором используется удаляемая функция.

Однако предложение OR REPLACE не решает всех задач. Таким способом нельзя изменить имя функции или типы ее аргументов (просто будет создана новая функция), также нельзя изменить тип возвращаемого ею значения и, следовательно, типы параметров, имеющих модификатор OUT. Нельзя изменить имена входных параметров, а если выходных параметров больше одного, то их имена также нельзя изменить, поскольку это изменило бы имена полей анонимного составного типа, описывающего возвращаемое значение функции. Обобщая, можно сказать, что при наличии предложения OR REPLACE выполняется поиск существующей функции по сигнатуре. Если он был успешным, тело функции заменяется новым, в противном случае создается новая функция, поскольку изменить сигнатуру нельзя. Эти ограничения накладываются, чтобы обеспечить работоспособность уже существующих вызовов функции после замены ее определения.

Если же предложение OR REPLACE не позволяет решить задачу, придется удалить функцию и создать ее заново. Более подробно эти вопросы объясняются в документации в описании SQL-команды CREATE FUNCTION.

Давайте вернемся к нашим экспериментам и добавим ограничение, основанное на использовании этой функции.

```
ALTER TABLE aircrafts_tmp
  ADD CHECK ( air_code_correct( aircraft_code ) );
ALTER TABLE
\d aircrafts_tmp
                    Таблица "pg_temp_11.aircrafts_tmp"
    Столбец    |     Тип      | Правило сортировки | Допустимость NULL | По умолчанию
---------------+--------------+--------------------+-------------------+--------------
 aircraft_code | character(3) |                    |                   |
 model         | text         |                    |                   |
 range         | integer      |                    |                   |
Ограничения-проверки:
    "aircrafts_tmp_aircraft_code_check" CHECK (air_code_correct(aircraft_code))
```

Повторить запросы к системным каталогам читатель может самостоятельно. Мы же ограничимся тем, что посмотрим, какое значение OID имеет наша функция. Потом мы проверим, изменится ли оно при создании нового определения функции.

```
SELECT oid, proname
FROM pg_proc
WHERE proname = 'air_code_correct';
```

```
  oid  |     proname
-------+-----------------
 16618 | air_code_correct
(1 строка)
```

Модифицируем функцию и пересоздадим ее:

```
CREATE OR REPLACE FUNCTION air_code_correct( a_code char )
RETURNS bool
LANGUAGE sql
RETURN a_code ~ '^[0-9a-z]{3}$'; -- цифры и строчные буквы
```

```
CREATE FUNCTION
```

Идентификатор этой функции не изменился, значит, для системы это тот же самый объект:

```
SELECT oid, proname
FROM pg_proc
WHERE proname = 'air_code_correct';
```

```
  oid  |     proname
-------+-----------------
 16618 | air_code_correct
(1 строка)
```

Сохранилось ли ограничение? Да, оно сохранилось:

```
\d aircrafts_tmp
```

```
                    Таблица "pg_temp_11.aircrafts_tmp"
    Столбец    |     Тип      | Правило сортировки | Допустимость NULL | По умолчанию
---------------+--------------+--------------------+-------------------+--------------
 aircraft_code | character(3) |                    |                   |
 model         | text         |                    |                   |
 range         | integer      |                    |                   |
Ограничения-проверки:
    "aircrafts_tmp_aircraft_code_check" CHECK (air_code_correct(aircraft_code))
```

Таким образом, если от конкретной функции зависят объекты базы данных, то при ее удалении эти объекты также будут удалены (с учетом вида зависимости — deptype). Однако при использовании предложения OR REPLACE функция сохраняется с тем же OID, поэтому зависимые объекты также сохраняются.

5.2.2. Зависимость функций от объектов базы данных

Теперь посмотрим, что происходит с функцией, когда удаляются объекты, которые в ней используются.

Чтобы показать разницу между ситуациями, когда тело функции представлено в виде строки и когда она написана в стиле стандарта SQL, воспользуемся функциями make_or_update_booking и make_or_update_booking_sql, представляющими эти стили. Они рассматривались в подразделе 5.1 «Базовые сведения о функциях» (с. 263). Перед началом экспериментов (которые мы будем проводить в транзакции) обе функции должны присутствовать в системе:

```
\df make_or_update_booking*
```

```
                                Список функций
  Схема   |            Имя              | Тип данных результата |
----------+-----------------------------+-----------------------+-------------------------
 bookings | make_or_update_booking      | numeric               | b_ref character, t_no...
 bookings | make_or_update_booking_sql  | numeric               | b_ref character, t_no...
(2 строки)
```

```
BEGIN;
BEGIN
```

Давайте попробуем удалить таблицу «Перелеты» (ticket_flights), которая используется обеими функциями:

```
DROP TABLE ticket_flights CASCADE;
ЗАМЕЧАНИЕ:  удаление распространяется на ещё 2 объекта
ПОДРОБНОСТИ:  удаление распространяется на объект ограничение
boarding_passes_ticket_no_fkey в отношении таблица boarding_passes
удаление распространяется на объект функция
make_or_update_booking_sql(character,character,character
varying,text,jsonb,integer,character varying,numeric)
DROP TABLE
```

Удаление таблицы проходит успешно. Кроме нее, удаляются внешний ключ таблицы «Посадочные талоны» (boarding_passes) и функция make_or_update_booking_sql. А вот связи функции make_or_update_booking с удаляемой таблицей система не замечает.

Итак, функции, написанной в стиле стандарта SQL, у нас теперь нет (после отмены транзакции она, конечно, восстановится). Но мы можем вызвать оставшуюся функцию make_or_update_booking. Возьмем запрос, который выполняли в подразделе 5.1.4 «Функции, включающие несколько SQL-команд» (с. 280):

```
SELECT make_or_update_booking
( b_ref => 'ABC123',
  t_no => '1234567890123',
  p_id => '0000 123456',
  p_name => 'IVAN PETROV',
  c_data => '{"phone": "+7(123)4567890" }'::jsonb,
  f_id => 20502,
  f_cond => 'Economy',
  amt => 10000::numeric
) AS total_amount;
```

```
ОШИБКА:  отношение "ticket_flights" не существует
СТРОКА 11:   INSERT INTO ticket_flights ( ticket_no, flight_id, fare_co...
 ...
КОНТЕКСТ:  SQL-функция "make_or_update_booking" (при старте)
```

Вот теперь было обнаружено отсутствие таблицы, необходимой для нормальной работы функции make_or_update_booking.

```
ROLLBACK;
```

```
ROLLBACK
```

Обратимся к системному каталогу pg_depend и посмотрим, от каких функций (каталог pg_proc) и таблиц (pg_class) зависят две наши функции:

```
SELECT DISTINCT
  classname,
  objname,
  refclassname,
  refobjname,
  deptype
FROM pg_depend_v
WHERE classname = 'pg_proc'
  AND objname IN ( 'make_or_update_booking', 'make_or_update_booking_sql' )
  AND refclassname IN ( 'pg_proc', 'pg_class' );
```

```
 classname |          objname           | refclassname |   refobjname   | deptype
-----------+----------------------------+--------------+----------------+---------
 pg_proc   | make_or_update_booking_sql | pg_class     | bookings       | normal
 pg_proc   | make_or_update_booking_sql | pg_class     | ticket_flights | normal
 pg_proc   | make_or_update_booking_sql | pg_class     | tickets        | normal
 pg_proc   | make_or_update_booking_sql | pg_proc      | bookings.now   | normal
(4 строки)
```

Таким образом:

- Если тело функции представлено строкой, ее зависимость от удаляемых объектов обнаруживается только во время выполнения запроса с этой функцией.
- Если функция написана в стиле стандарта SQL, то при попытке удаления объекта, который она использует, возникает ошибка. Однако если команда удаления включает предложение CASCADE, то будет удален не только этот объект, но и сама функция.

В общем случае на выполнение операции удаления влияет вид зависимости между объектами, который отражается в столбце depend системного каталога pg_depend.

А что будет, если при создании функции необходимые ей объекты не существуют? Этот вопрос рассмотрен в упражнении 7 (с. 377).

5.3. Функции, возвращающие множества строк

До настоящего момента все функции, которые мы написали в этой главе, возвращали только одну строку. Но, конечно, могут возникать ситуации, когда хотелось бы с помощью функции динамически формировать выборку, то есть получать множество строк.

Предположим, что сотрудникам экономического отдела нашей авиакомпании часто требуется выбирать все маршруты, проложенные из конкретного города в несколько других городов. Давайте сначала напишем функцию, которая будет получать в качестве входных параметров названия города отправления и *одного* города назначения. Очевидно, что результатом такой выборки в общем случае может быть *несколько* строк.

```
CREATE OR REPLACE FUNCTION list_routes
( d_city text DEFAULT 'Москва',
  a_city text DEFAULT 'Санкт-Петербург'
)
RETURNS SETOF routes AS
$$
  SELECT * FROM routes
  WHERE departure_city = d_city
    AND arrival_city = a_city;
$$ LANGUAGE sql;
CREATE FUNCTION
```

Функция определена как возвращающая SETOF routes. Это означает *множество* значений типа routes. Но что это за тип? Как вы знаете, при создании таблицы (а также других отношений, например представлений или материализованных представлений) создается одноименный *составной* тип данных, соответствующий строке. А в нашей базе данных есть представление с именем routes («Маршруты»). Поэтому фактически функция возвращает множество строк представления routes.

Давайте проверим функцию в работе. Поскольку она возвращает значения типа routes, мы можем использовать имена полей этого типа в списке SELECT, выбирая при этом лишь интересующие нас поля:

```
SELECT
  flight_no,
  departure_city AS dep_city,
  departure_airport AS dep_ap,
  arrival_city AS arr_city,
  arrival_airport AS arr_ap,
  days_of_week
FROM list_routes( 'Владивосток', 'Иркутск' )
ORDER BY days_of_week;
 flight_no |  dep_city   | dep_ap | arr_city | arr_ap |  days_of_week
-----------+-------------+--------+----------+--------+-----------------
 PG0660    | Владивосток | VVO    | Иркутск  | IKT    | {1,2,3,4,5,6,7}
(1 строка)
```

Если название города отправления или прибытия в функцию не передано, то будет использоваться значение по умолчанию, заданное с помощью ключевого слова DEFAULT. Давайте выберем все маршруты, ведущие из Москвы в Курган. Для этого достаточно передать функции значение только второго параметра.

Нужно учитывать, что при позиционной передаче аргументов мы не сможем передать значение второго из них, не передав значение первого. Однако это можно сделать с помощью именованного аргумента:

```
SELECT
  flight_no,
  arrival_city AS arr_city,
  arrival_airport AS arr_ap,
  days_of_week
FROM list_routes( a_city => 'Курган' )
ORDER BY days_of_week;
 flight_no | arr_city | arr_ap |  days_of_week
-----------+----------+--------+----------------
 PG0669    | Курган   | KRO    | {1,2,3,4,5,6,7}
 PG0370    | Курган   | KRO    | {2,4,7}
 PG0667    | Курган   | KRO    | {4,7}
(3 строки)
```

Обратите внимание, что массивы сортируются в лексикографическом порядке.

Функции, которые, как и list_routes, возвращают множество строк, называются *табличными функциями*. Строка может состоять лишь из одного скалярного значения, а может представлять собой значение составного типа. Такие функции можно использовать в предложении FROM команды SELECT так же, как и обычную таблицу, представление или подзапрос. Столбцы, возвращенные функцией, могут фигурировать в списке SELECT, в предложении JOIN или в условии WHERE, как будто это столбцы таблицы, представления или результирующие столбцы подзапроса. Подробно этот вопрос рассмотрен в подразделе документации 7.2.1.4 «Табличные функции».

А если вызвать функцию не в предложении FROM, а непосредственно в списке SELECT, то будет явно видно, что она возвращает множество значений составного типа routes:

```
SELECT list_routes( 'Элиста' );
                                  list_routes
-------------------------------------------------------------------------------
 (PG0524,ESL,Элиста,Элиста,LED,Пулково,Санкт-Петербург,CR2,02:30:00,"{4,7}")
(1 строка)
```

Надо заметить, что при вызове табличной функции таким способом возможны неочевидные результаты. Если функция не вернет ни одной строки, то и запрос не вернет ни одной строки, даже если в списке SELECT есть константы:

```
SELECT 'Республика Беларусь' AS country, list_routes( 'Минск' );
 country | list_routes
---------+-------------
(0 строк)
```

При нескольких вызовах табличных функций, возвращающих разное количество строк, число результирующих строк определяется максимальным количеством. Такой случай рассмотрен в упражнении 12 (с. 388).

Наша функция обращается только к представлению «Маршруты» (routes), поэтому если мы захотим для повышения наглядности выводить не код модели самолета, а ее название, нам придется соединить результат с таблицей «Самолеты» (aircrafts). При этом, конечно, мы можем в списке SELECT указать не все столбцы, возвращаемые функцией list_routes. Видно, что функция ведет себя так, как будто она является таблицей:

```
SELECT lr.flight_no, a.model
FROM list_routes( 'Пермь', 'Екатеринбург' ) AS lr
  JOIN aircrafts AS a ON a.aircraft_code = lr.aircraft_code
ORDER BY lr.flight_no;
 flight_no |        model
-----------+---------------------
 PG0589    | Сухой Суперджет-100
 PG0590    | Сухой Суперджет-100
(2 строки)
```

Чтобы облегчить использование функции, можно перенести операцию соединения внутрь нее. Давайте так и сделаем. Новая версия функции отличается от предыдущей типом возвращаемого значения и наличием соединения (JOIN):

```
CREATE OR REPLACE FUNCTION list_routes_2
( d_city text DEFAULT 'Москва',
  a_city text DEFAULT 'Санкт-Петербург'
)
RETURNS SETOF record AS
$$
  SELECT r.flight_no, r.departure_city, r.arrival_city, a.model
  FROM routes AS r
    JOIN aircrafts AS a ON a.aircraft_code = r.aircraft_code
  WHERE r.departure_city = d_city
    AND r.arrival_city = a_city;
$$ LANGUAGE sql;
CREATE FUNCTION
```

Функция формирует несколько столбцов, но их совокупность не соответствует какому-либо составному типу данных, как это было в предыдущей версии функции, которая возвращала множество значений типа routes. Можно было бы создать составной тип с помощью команды CREATE TYPE, как мы показывали ранее в этой же главе, а затем указать его в качестве возвращаемого значения функции. Здесь мы поступили иначе — использовали псевдотип record (запись). Однако его неопределенную структуру надо заранее определить, иначе планировщик не сможет разобрать запрос и построить план выполнения:

```
SELECT flight_no, departure_city, arrival_city, model
FROM list_routes_2( 'Южно-Сахалинск', 'Хабаровск' ) AS lr
ORDER BY flight_no;
```

```
ОШИБКА:  у функций, возвращающих запись, должен быть список определений столбцов
СТРОКА 2: FROM list_routes_2( 'Южно-Сахалинск', 'Хабаровск' ) AS lr
               ^
```

Давайте, как рекомендуется в сообщении, определим структуру значения составного типа, то есть зададим список с определениями столбцов. Определение столбца — это его имя и тип данных, а не просто псевдоним, которого достаточно в других случаях. Конечно, имена столбцов в этом списке могут не совпадать с именами столбцов, которые используются внутри функции.

```
SELECT f_no, dep_city, arr_city, model
FROM list_routes_2( 'Южно-Сахалинск', 'Хабаровск' )
       AS lr ( f_no char( 6 ), dep_city text, arr_city text, model text )
ORDER BY f_no;
```

```
  f_no  |    dep_city    | arr_city  |       model
--------+----------------+-----------+--------------------
 PG0398 | Южно-Сахалинск | Хабаровск | Сессна 208 Караван
 PG0399 | Южно-Сахалинск | Хабаровск | Сессна 208 Караван
(2 строки)
```

Но SQL-функция всегда возвращает значения с одним и тем же набором столбцов. Возникает вопрос: нельзя ли обойтись без задания списка при вызове функции, не создавая нового составного типа? Обойтись можно. Для этого нужно определить поля возвращаемого составного типа в заголовке самой функции с помощью OUT-параметров.

Добавим, что при необходимости можно определить выходной параметр существующего составного типа, например OUT route routes.

Новая, третья, версия функции отличается от предыдущей только списком параметров:

```
CREATE OR REPLACE FUNCTION list_routes_3
( d_city text DEFAULT 'Москва',
  a_city text DEFAULT 'Санкт-Петербург',
  OUT f_no char,
  OUT dep_city text,
  OUT arr_city text,
  OUT model text
)
RETURNS SETOF record AS
$$
  SELECT r.flight_no, r.departure_city, r.arrival_city, a.model
  FROM routes AS r
    JOIN aircrafts AS a ON a.aircraft_code = r.aircraft_code
  WHERE r.departure_city = d_city
    AND r.arrival_city = a_city;
$$ LANGUAGE sql;
CREATE FUNCTION
```

Запрос стал проще. Поскольку теперь мы не определяем столбцы при вызове функции, то в списке SELECT должны использовать имена соответствующих выходных параметров:

```
SELECT dep_city, arr_city, f_no, model
FROM list_routes_3( 'Сочи', 'Ростов-на-Дону' )
ORDER BY f_no;
 dep_city |    arr_city    |  f_no  |        model
----------+----------------+--------+--------------------
 Сочи     | Ростов-на-Дону | PG0486 | Сухой Суперджет-100
 Сочи     | Ростов-на-Дону | PG0487 | Сухой Суперджет-100
(2 строки)
```

Конечно, если все же потребуется изменить заголовки столбцов, можно задать их псевдонимы (не определения!) при вызове функции, а не только в списке SELECT. Заголовки столбцов могут быть и на русском языке, однако использовать такой подход в промышленном коде, конечно, не рекомендуется, ведь это идентификаторы, такие же, как и переменные:

```
SELECT *
FROM list_routes_3( 'Ульяновск', 'Саратов' ) AS f( Рейс, Откуда, Куда, Самолет )
ORDER BY Рейс;
```

```
  Рейс  |  Откуда   |  Куда   |      Самолет
--------+-----------+---------+--------------------
 PG0466 | Ульяновск | Саратов | Сессна 208 Караван
 PG0467 | Ульяновск | Саратов | Сессна 208 Караван
(2 строки)
```

Есть еще один способ объявления возвращаемого значения в виде множества строк: предложение RETURNS TABLE со списком определений столбцов. Напомним, что определение — это не только имя столбца, но и его тип данных. Предложение RETURNS TABLE эквивалентно комбинации предложения RETURNS SETOF record (или SETOF со скалярным типом) и OUT-параметров. В подразделе документации 36.5.10 «Функции SQL, возвращающие таблицы (TABLE)» сказано, что такой способ описан в новых версиях стандарта языка SQL, поэтому он более переносимый, чем RETURNS SETOF. Важно, что при этом не нужно (и нельзя) определять параметры с модификаторами OUT и INOUT.

Поскольку новая, четвертая, версия функции отличается от предыдущей только списком параметров и возвращаемым значением, то приведем лишь ее начало.

```
CREATE OR REPLACE FUNCTION list_routes_4
( d_city text DEFAULT 'Москва',
  a_city text DEFAULT 'Санкт-Петербург'
)
RETURNS TABLE ( f_no char, dep_city text, arr_city text, model text ) AS
...
CREATE FUNCTION
```

Проверим функцию в работе:

```
SELECT f_no, dep_city, arr_city, model
FROM list_routes_4( 'Пермь', 'Архангельск' )
ORDER BY f_no;
```

```
  f_no  | dep_city |  arr_city   |       model
--------+----------+-------------+--------------------
 PG0492 | Пермь    | Архангельск | Сессна 208 Караван
 PG0493 | Пермь    | Архангельск | Сессна 208 Караван
(2 строки)
```

Таким образом, функции, возвращающие множества строк, могут иметь разное оформление списка параметров и возвращаемого значения. Поэтому при вызове функции может потребоваться или не потребоваться задать определения столбцов возвращаемого ею множества строк.

5.4. Функции с переменным числом аргументов

В предыдущем подразделе мы поставили задачу написать функцию, которая будет выбирать все маршруты, проложенные из конкретного города в *несколько* других городов. В качестве первого шага мы разработали функцию, выбирающую маршруты между *двумя* городами.

Настало время вернуться к исходной задаче. Уточним, что число городов прибытия может быть различным при разных вызовах функции. Реализовать это требование можно с помощью параметра с модификатором VARIADIC. Он позволяет при вызове функции передавать ей переменное число аргументов, аналогично функциям printf и scanf в языке C. Хотя параметр с модификатором VARIADIC задается в виде массива, но на самом деле функция ожидает скалярные аргументы, имеющие *тип элементов* этого массива. При вызове функции аргументы, синтаксически оформленные как скалярные, помещаются в массив и передаются функции как единый входной аргумент. Внутри функции эти значения будут доступны ей в виде массива. Неявно предполагается, что параметры с модификатором VARIADIC имеют модификатор IN.

```
CREATE OR REPLACE FUNCTION list_routes_5
( d_city text,
  VARIADIC a_cities text[]
)
RETURNS SETOF routes AS
$$
  SELECT * FROM routes
  WHERE departure_city = d_city
    AND arrival_city = ANY( a_cities );
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

Первым параметром функции является город отправления, вторым — список городов прибытия. Он объявлен как массив a_cities, содержащий элементы типа text. При этом длина массива не указана (поскольку модификаторы типов при создании функций отбрасываются), зато он имеет модификатор VARIADIC.

Поскольку внутри функции список городов назначения будет представлен в виде массива, можно воспользоваться оператором ANY.

Проверим функцию в работе, выбрав маршруты из Москвы в Кемерово, Красноярск и Элисту:

```
SELECT
  flight_no,
  arrival_city AS arr_city,
  arrival_airport AS arr_ap,
  days_of_week
FROM list_routes_5( 'Москва', 'Кемерово', 'Красноярск', 'Элиста' )
ORDER BY arrival_city, days_of_week;
```

Мы передали функции не массив, а скалярные параметры одного и того же типа, причем такого, который указан в определении параметра-массива с модификатором VARIADIC.

```
 flight_no |  arr_city  | arr_ap |  days_of_week
-----------+------------+--------+----------------
 PG0482    | Кемерово   | KEJ    | {2,4,7}
 PG0663    | Кемерово   | KEJ    | {2,5}
 PG0347    | Кемерово   | KEJ    | {4,7}
 PG0547    | Красноярск | KJA    | {1,2,3,4,5,6,7}
 PG0299    | Элиста     | ESL    | {1,2,3,4,5,6,7}
 PG0509    | Элиста     | ESL    | {1,2,3,4,5,6,7}
(6 строк)
```

Параметры с модификатором VARIADIC также могут иметь значения по умолчанию. Представлять их нужно в виде массива, а не в виде списка скалярных значений.

Давайте создадим новую версию функции. Она будет отличаться от предыдущей только наличием списка городов прибытия по умолчанию. Ими станут Москва и Санкт-Петербург:

```
CREATE OR REPLACE FUNCTION list_routes_6
( d_city text,
  VARIADIC a_cities text[] DEFAULT ARRAY[ 'Москва', 'Санкт-Петербург' ]::text[]
)
RETURNS SETOF routes AS
$$
  SELECT * FROM routes
  WHERE departure_city = d_city
    AND arrival_city = ANY( a_cities );
$$ LANGUAGE sql;
CREATE FUNCTION
```

Вызовем функцию, задав только один аргумент:

```
SELECT
  flight_no,
  arrival_city AS arr_city,
  arrival_airport AS arr_ap,
  days_of_week
FROM list_routes_6( 'Элиста' )
ORDER BY arrival_city, days_of_week;
 flight_no |    arr_city     | arr_ap |  days_of_week
-----------+-----------------+--------+----------------
 PG0300    | Москва          | SVO    | {1,2,3,4,5,6,7}
 PG0510    | Москва          | DME    | {1,2,3,4,5,6,7}
 PG0524    | Санкт-Петербург | LED    | {4,7}
(3 строки)
```

Если в какой-то ситуации окажется, что список городов назначения удобно получить с помощью подзапроса, то наша функция подойдет и для такого случая. Массив можно сформировать с помощью конструктора ARRAY (его использование описано в подразделе документации 4.2.12 «Конструкторы массивов»). Передавая функции параметр-массив вместо списка скалярных значений, следует обязательно добавить модификатор VARIADIC.

Вот какие маршруты проложены из Санкт-Петербурга в города часового пояса Asia/Yekaterinburg:

```
SELECT
  flight_no,
  arrival_city AS arr_city,
  arrival_airport AS arr_ap,
  days_of_week
FROM list_routes_6(
    'Санкт-Петербург',
    VARIADIC ARRAY ( SELECT city FROM airports WHERE timezone = 'Asia/Yekaterinburg' )
  )
ORDER BY arrival_city, days_of_week;
 flight_no |   arr_city    | arr_ap |  days_of_week
-----------+---------------+--------+----------------
 PG0156    | Нижневартовск | NJC    | {1,2,3,4,5,6,7}
 PG0114    | Ноябрьск      | NOJ    | {3,6}
 PG0360    | Оренбург      | REN    | {1,2,3,4,5,6,7}
 PG0685    | Советский     | OVS    | {1,2,3,4,5,6,7}
 PG0188    | Тюмень        | TJM    | {1,2,3,4,5,6,7}
(5 строк)
```

Таким образом, можно выбирать способ передачи аргументов в параметр с модификатором VARIADIC: либо список скалярных значений одного и того же типа, либо массив, но во втором случае приходится добавлять это же ключевое слово при вызове функции.

В предыдущем подразделе мы рассматривали ситуацию, когда функция формирует несколько столбцов, совокупность которых не соответствует какому-либо существующему составному типу данных. В таком случае — это была функция list_routes_3 — в предложении RETURNS мы писали SETOF record и добавляли ряд выходных параметров, чтобы при вызове функции не задавать определения результирующих столбцов.

Давайте в этом же ключе создадим новую функцию на основе list_routes_6:

```
CREATE OR REPLACE FUNCTION list_routes_7
( d_city text,
  VARIADIC a_cities text[] DEFAULT ARRAY[ 'Москва', 'Санкт-Петербург' ]::text[],
  OUT f_no char,
  OUT dep_city text,
  OUT arr_city text,
  OUT model text
)
RETURNS SETOF record AS
$$
  SELECT r.flight_no, r.departure_city, r.arrival_city, a.model
  FROM routes AS r
    JOIN aircrafts AS a ON a.aircraft_code = r.aircraft_code
  WHERE r.departure_city = d_city
    AND r.arrival_city = ANY( a_cities );
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

Как видим, наличие параметра с модификатором VARIADIC не препятствует использованию выходных параметров.

Выполняем запрос и получаем ожидаемый результат: задавать определения столбцов при вызове функции не требуется.

```
SELECT arr_city, f_no, model
FROM list_routes_7( 'Сочи', 'Белгород', 'Ростов-на-Дону', 'Новокузнецк' )
ORDER BY arr_city, f_no;
```

```
    arr_city    |  f_no  |        model
----------------+--------+---------------------
 Белгород       | PG0617 | Сессна 208 Караван
 Белгород       | PG0618 | Сессна 208 Караван
 Новокузнецк    | PG0322 | Боинг 737-300
 Ростов-на-Дону | PG0486 | Сухой Суперджет-100
 Ростов-на-Дону | PG0487 | Сухой Суперджет-100
(5 строк)
```

Модификацию функции с использованием предложения RETURNS TABLE предлагаем читателю выполнить самостоятельно.

Посмотреть описания созданных нами функций можно следующей командой:

```
\df list_routes_[5-7]
```

Для полноты картины приведем также команду удаления одной из функций. В этой команде можно не указывать модификатор VARIADIC, однако нужно не забыть о квадратных скобках:

```
DROP FUNCTION IF EXISTS list_routes_6( text, text[] );
DROP FUNCTION
```

Поскольку перегруженной функции с таким же именем нет, то можно было опустить и список типов данных:

```
DROP FUNCTION IF EXISTS list_routes_6;
```

5.5. Конструкция LATERAL и функции

Все табличные функции, разработанные нами до сих пор, получали в качестве аргумента название *только одного* города отправления. А если нам потребуется, скажем, выбрать все маршруты между городами того или иного часового пояса? Давайте возьмем для примера такой часовой пояс, в котором число городов слишком велико для того, чтобы легко решить поставленную задачу, выполняя несколько однотипных запросов. Это будет стимулировать поиск более общего и рационального решения.

```
SELECT timezone, count( * )
FROM airports
GROUP BY timezone ORDER BY count DESC;
```

```
      timezone      | count
--------------------+-------
 Europe/Moscow      |    44
 Asia/Yekaterinburg |    22
 Asia/Krasnoyarsk   |     8
 Europe/Samara      |     5
 ...
 Asia/Chita         |     1
 Asia/Anadyr        |     1
(17 строк)
```

Для экспериментов подходит часовой пояс Asia/Yekaterinburg, в котором 22 аэропорта.

Мы бы решили поставленную задачу, если бы смогли в рамках одного запроса, выбирающего все города в часовом поясе Asia/Yekaterinburg, вызывать функцию, скажем, `list_routes_7` для *каждой* выбираемой строки, так, чтобы из этой строки функция получала значение своего первого параметра. Такая возможность есть! В главе 4 «Конструкция LATERAL команды SELECT» (с. 225) мы рассматривали конструкцию `LATERAL` на примере подзапросов к таблицам, а также встроенным функциям PostgreSQL, возвращающим множества, например `generate_series`. Точно так же этот механизм работает и с пользовательскими функциями. Запрос будет таким:

```
SELECT a.city AS dep_city, lr.arr_city, lr.f_no, lr.model
FROM airports AS a
  LEFT OUTER JOIN LATERAL list_routes_7(
      a.city,
      VARIADIC ARRAY( SELECT city FROM airports WHERE timezone = 'Asia/Yekaterinburg' )
    ) AS lr
    ON true
WHERE a.timezone = 'Asia/Yekaterinburg'
ORDER BY dep_city, lr.arr_city;
```

Запрос выполняется так: сканируется таблица «Аэропорты» (`airports`), и для ее текущей строки вызывается функция `list_routes_7`, которая в качестве первого аргумента получает значение поля `a.city` из этой строки. Строки, возвращенные функцией, соединяются с этой строкой таблицы «Аэропорты» (`airports`) обычным образом, как это делается для строк соединяемых таблиц. Затем

эти же действия повторяются для следующей строки таблицы «Аэропорты» (airports). Так шаг за шагом формируется результирующее множество строк.

Исходя из предположения, что могут найтись города, не связанные ни с одним городом этого же часового пояса, мы используем левое внешнее соединение. Оно гарантирует формирование результирующей строки выборки даже в том случае, если функция list_routes_7 не возвратит ни одной строки для какого-либо города. Конечно, при этом значения полей lr.arr_city, lr.f_no и lr.model в результирующей строке будут пустыми.

Попутно напомним, что в предложении ORDER BY можно использовать не только *имена* столбцов, перечисленных в списке SELECT, но и их *псевдонимы*. Поэтому предложение ORDER BY dep_city, lr.arr_city будет работать корректно.

В результате выполнения запроса оказалось, что существует четыре города, из которых не предусмотрено рейсов ни в один город данного часового пояса:

```
    dep_city    |    arr_city    |  f_no  |        model
----------------+----------------+--------+---------------------
 Белоярский     | Курган         | PG0366 | Сессна 208 Караван
 Белоярский     | Тюмень         | PG0577 | Сессна 208 Караван
 Екатеринбург   | Когалым        | PG0102 | Сессна 208 Караван
 Екатеринбург   | Когалым        | PG0103 | Сессна 208 Караван
 Екатеринбург   | Магнитогорск   | PG0166 | Бомбардье CRJ-200
 Екатеринбург   | Пермь          | PG0588 | Сухой Суперджет-100
 Екатеринбург   | Пермь          | PG0587 | Сухой Суперджет-100
 Екатеринбург   | Сургут         | PG0303 | Сухой Суперджет-100
 Екатеринбург   | Ханты-Мансийск | PG0599 | Сухой Суперджет-100
 ...
 Надым          |                |        |
 Нефтеюганск    |                |        |
 Нижневартовск  | Советский      | PG0641 | Бомбардье CRJ-200
 ...
 Нягань         |                |        |
 Оренбург       | Уфа            | PG0149 | Бомбардье CRJ-200
 ...
 Салехард       |                |        |
 ...
 Челябинск      | Пермь          | PG0675 | Бомбардье CRJ-200
 Челябинск      | Сургут         | PG0191 | Сессна 208 Караван
 Челябинск      | Сургут         | PG0190 | Сессна 208 Караван
(54 строки)
```

Чтобы лучше разобраться в механизме выполнения этого запроса, обратимся к его плану:

```
EXPLAIN ( analyze, costs off, timing off )
...
                              QUERY PLAN
-----------------------------------------------------------------------
 Sort (actual rows=54 loops=1)
   Sort Key: ((ml.city ->> lang())), lr.arr_city
   Sort Method: quicksort  Memory: 29kB
   InitPlan 1
     -> Seq Scan on airports_data ml_1 (actual rows=22 loops=1)
          Filter: (timezone = 'Asia/Yekaterinburg'::text)
          Rows Removed by Filter: 82
   -> Nested Loop Left Join (actual rows=54 loops=1)
        -> Seq Scan on airports_data ml (actual rows=22 loops=1)
             Filter: (timezone = 'Asia/Yekaterinburg'::text)
             Rows Removed by Filter: 82
        -> Function Scan on list_routes_7 lr (actual rows=2 loops=22)
 Planning Time: 0.133 ms
 Execution Time: 783.031 ms
(14 строк)
```

В этом плане фрагмент InitPlan 1 отвечает за подзапрос, формирующий массив параметров VARIADIC для функции list_routes_7. Показатель loops=1 говорит о том, что данный подзапрос выполняется только один раз, поскольку он не коррелированный.

Напомним, что выражение ml.city->>lang() означает получение значения поля (в виде символьной строки) из JSON-объекта, а имя поля определяется функцией lang, которая создана в схеме bookings. Структуру представления airports и таблицы airports_data, в которой столбец city имеет тип jsonb, можно посмотреть с помощью команды \d утилиты psql.

Способ соединения строк — вложенные циклы (Nested Loop Left Join). При этом в качестве внешнего набора строк — в узле Seq Scan — используется таблица airports_data, на основе которой создано представление «Аэропорты» (airports), участвующее в запросе. Во внешнем наборе после применения условия, заданного в предложении WHERE, остается всего 22 строки, о чем говорит значение фактического показателя (actual rows=22).

Внутренний набор — в узле Function Scan — формирует функция list_routes_7. Она получает в качестве первого аргумента значение поля city из *текущей строки* внешнего набора. Если функция возвратила одну или несколько строк, они соединяются с этой строкой по обычным правилам. Фактические значения показателей (actual rows=2 loops=22) говорят о том, что функция вызывается 22 раза, то есть для каждой строки из внешнего набора, при этом среднее количество строк, возвращаемых ею, равно двум (это результат округления до целого значения). Добавим, что если даже функция не возвращает ни одной строки, то итоговая строка выборки все равно формируется, поскольку мы используем левое внешнее соединение, что и отражено в плане как Nested Loop Left Join.

Если бы нас не интересовали города, из которых рейсы не выполняются, то можно было бы обойтись без внешнего соединения:

```
SELECT a.city AS dep_city, lr.arr_city, lr.f_no, lr.model
FROM airports AS a
  CROSS JOIN list_routes_7(
      a.city,
      VARIADIC ARRAY( SELECT city FROM airports WHERE timezone = 'Asia/Yekaterinburg' )
    ) AS lr
WHERE a.timezone = 'Asia/Yekaterinburg'
ORDER BY dep_city, lr.arr_city;
```

```
    dep_city    |    arr_city    |  f_no  |        model
----------------+----------------+--------+---------------------
 Белоярский     | Курган         | PG0366 | Сессна 208 Караван
 Белоярский     | Тюмень         | PG0577 | Сессна 208 Караван
 Екатеринбург   | Когалым        | PG0102 | Сессна 208 Караван
 Екатеринбург   | Когалым        | PG0103 | Сессна 208 Караван
 ...
 Ханты-Мансийск | Орск           | PG0350 | Бомбардье CRJ-200
 Челябинск      | Пермь          | PG0675 | Бомбардье CRJ-200
 Челябинск      | Сургут         | PG0191 | Сессна 208 Караван
 Челябинск      | Сургут         | PG0190 | Сессна 208 Караван
(50 строк)
```

Согласно приведенному в документации описанию команды SELECT, при использовании функций в качестве источника строк в предложении FROM можно опускать ключевое слово LATERAL.

Более масштабный пример использования табличных функций в конструкции LATERAL рассмотрен в упражнении 11 (с. 381).

5.6. Категории изменчивости функций

Каждая функция, как встроенная, так и написанная нами, имеет очень важную характеристику: *категорию изменчивости* (volatility). Это свойство проявляется в двух формах. Во-первых, категория изменчивости, как сказано в разделе документации 36.7 «Категории изменчивости функций», представляет собой обещание некоторого предсказуемого поведения функции. Это позволяет оптимизатору построить более экономный план выполнения запроса. Второй стороной категории изменчивости является видимость собственных изменений: доступны ли функции, вызываемой в SQL-операторе, те изменения базы данных, которые произвел этот оператор?

Сначала рассмотрим теоретические положения, а затем подробно разберем различные ситуации.

Итак, всего существует три категории изменчивости функции: `VOLATILE` — изменчивая, `STABLE` — стабильная, `IMMUTABLE` — постоянная.

Изменчивая функция (`VOLATILE`) может не только читать, но и изменять данные. Важно, что в рамках одного SQL-запроса она может возвращать *различные результаты*, если будет вызвана несколько раз с *одинаковыми аргументами* (или без аргументов, если у функции нет параметров). Конечно, если аргументы будут различаться, то и возвращаемое значение вовсе не обязано быть одним и тем же. Когда в запросе присутствует изменчивая функция, оптимизатор не делает никаких предположений о ее поведении. Поэтому значение функции будет вычисляться заново каждый раз, когда потребуется.

Стабильная функция (`STABLE`) не может модифицировать базу данных. Будучи вызванной в рамках одного SQL-оператора, она, получая *одинаковые аргументы*, должна возвращать *одинаковые результаты*. Это позволяет оптимизатору заменить множество вызовов такой функции одним. Но если аргументы будут различаться, то и значения функции также могут различаться, и в таком случае обойтись одним вызовом функции будет нельзя. В качестве примера стабильной функции можно привести `CURRENT_TIMESTAMP`, результат выполнения которой меняется от транзакции к транзакции.

Оптимизатор может применить индекс для поиска по условию, включающему вызов функции. Но при индексном доступе искомое значение вычисляется

только один раз, а не для каждой строки, поэтому функция должна быть стабильной (STABLE): планировщик не будет использовать индекс, если в условии присутствует функция с характеристикой VOLATILE.

Самой строгой категорией является постоянная (IMMUTABLE). Функция с такой категорией не может модифицировать базу данных. Она *всегда*, в любых SQL-запросах, возвращает *одинаковые результаты* для одинаковых значений аргументов. Если аргументы, переданные такой функции в запросе, являются константами, оптимизатор может вычислить значение функции еще на стадии планирования.

Важно понимать, что первичным является содержание функции, а не категория изменчивости, то есть категория назначается в зависимости от операций, выполняемых функцией. Конкретную категорию для нее выбирает разработчик, а не PostgreSQL. В документации рекомендуется по возможности назначать самую строгую из категорий, подходящих для конкретной функции. Если функция изменяет содержимое базы данных, например содержит команды INSERT, UPDATE или DELETE, то ей можно назначить только категорию VOLATILE. А вот если функция содержит только команды SELECT и при этом возвращаемое функцией значение зависит исключительно от ее аргументов, ей, в принципе, можно назначить любую из категорий. Таковы многие функции, если так можно выразиться, вычислительного характера.

Если ошибочно назначена более строгая категория, чем допустимо для данной функции, PostgreSQL может этого не заметить *при создании* функции. Система позволит назначить категорию STABLE или IMMUTABLE функции, которая содержит команды INSERT, UPDATE, DELETE, но при попытке вызвать такую функцию будет сгенерирована ошибка.

Однако в том случае, если функция содержит только команды SELECT, но для одинаковых значений аргументов возвращает различные результаты, мы, в принципе, можем назначить ей категорию IMMUTABLE, хотя это и будет нарушением требования назначать самую строгую из допустимых категорий. В результате могут быть получены недостоверные результаты запросов к базе данных. Однако если данные, от которых зависит результат функции, изменяются крайне редко и контролируемо, иногда в подобных ситуациях назначают функции категорию IMMUTABLE с целью ускорения запросов.

5.6.1. Влияние изменчивости на выбор оптимального плана

Теперь перейдем к примерам, демонстрирующим рассмотренные концепции. Начнем с категории изменчивости VOLATILE. Эта самая «либеральная» категория. Вообще, все функции, разработанные в этой главе до сих пор, имели категорию изменчивости VOLATILE, потому что она назначается по умолчанию. Но сейчас для демонстрации выберем стандартную функцию random. В системе существуют несколько ее перегруженных версий, описанных в документации в разделе 9.3 «Математические функции и операторы» (см. таблицу 9.6 «Случайные функции»).

```
\df random
                                  Список функций
   Схема    |  Имя   | Тип данных результата |  Типы данных аргументов  |  Тип
------------+--------+-----------------------+--------------------------+-------
 pg_catalog | random | double precision      |                          | функ.
 pg_catalog | random | bigint                | min bigint, max bigint   | функ.
 pg_catalog | random | integer               | min integer, max integer | функ.
 pg_catalog | random | numeric               | min numeric, max numeric | функ.
(4 строки)
```

Давайте воспользуемся функцией, не имеющей параметров:

```
SELECT random(), random()
FROM generate_series( 1, 3 ) AS g;
       random        |       random
---------------------+---------------------
  0.5730866847145439 | 0.03652018300293447
 0.18976291110929533 |  0.8421532545403392
 0.44970059875321544 |  0.6837243829665287
(3 строки)
```

Как и полагается вести себя изменчивой функции, random возвращает различные значения при каждом вызове, в том числе и при неоднократном вызове для одной строки.

Узнать категорию изменчивости функции можно с помощью команды \df+ утилиты psql (см. описание этой утилиты в документации). Обратите внимание, что команда завершается минусом. Этот символ показывает, что нас интересуют только функции, число параметров которых *равно* числу заданных шаблонов типов этих параметров. Таким образом, следующая команда выберет именно функции без параметров, поскольку ни один шаблон типа не задан:

```
\df+ random -
Список функций
-[ RECORD 1 ]----------+-----------------
Схема                  | pg_catalog
Имя                    | random
Тип данных результата  | double precision
Типы данных аргументов |
Тип                    | функ.
Изменчивость           | изменчивая
Параллельность         | ограниченная
Владелец               | postgres
Безопасность           | вызывающего
Права доступа          |
Язык                   | internal
Внутреннее имя         | drandom
Описание               | random value
```

Для выбора перегруженной функции, имеющей параметры, нужно задать шаблоны их типов. При этом можно указать шаблоны не для всех параметров, например только для первого, не ограничивая их общее количество:

```
\df random (numeric|bigint)
                                Список функций
   Схема    |  Имя   | Тип данных результата |  Типы данных аргументов  |  Тип
------------+--------+-----------------------+--------------------------+-------
 pg_catalog | random | bigint                | min bigint, max bigint   | функ.
 pg_catalog | random | numeric               | min numeric, max numeric | функ.
(2 строки)
```

Тот же результат в нашем случае можно получить, задав точное соответствие числа шаблонов числу параметров функций. Минус в конце команды показывает, что нас интересуют лишь функции, имеющие в точности два параметра:

```
\df random (numeric|bigint) (numeric|bigint) -
                                Список функций
   Схема    |  Имя   | Тип данных результата |  Типы данных аргументов  |  Тип
------------+--------+-----------------------+--------------------------+-------
 pg_catalog | random | bigint                | min bigint, max bigint   | функ.
 pg_catalog | random | numeric               | min numeric, max numeric | функ.
(2 строки)
```

А вот с одним параметром функций random нет:

```
\df random (numeric|bigint) -
                          Список функций
 Схема | Имя | Тип данных результата | Типы данных аргументов | Тип
-------+-----+----------------------+------------------------+-----
(0 строк)
```

Второй способ выяснения категории изменчивости функции — запрос к системному каталогу pg_proc:

```
SELECT proname, proargtypes::regtype[], provolatile
FROM pg_proc
WHERE proname = 'random';
 proname |       proargtypes       | provolatile
---------+-------------------------+-------------
 random  | {}                      | v
 random  | [0:1]={integer,integer} | v
 random  | [0:1]={bigint,bigint}   | v
 random  | [0:1]={numeric,numeric} | v
(4 строки)
```

В столбце proargtypes приводятся типы данных, которые имеют параметры функции. При этом показано, что нумерация элементов массива начинается не с единицы (как принято по умолчанию), а с нуля. В столбце provolatile указывается категория изменчивости: v — изменчивая, s — стабильная, i — постоянная.

Еще одним представителем этой категории является функция timeofday. А вот функция CURRENT_TIMESTAMP будет стабильной (STABLE), поскольку ее значение остается постоянным в течение всей транзакции. Для демонстрации работы стабильных функций возьмем функцию date_part. Это перегруженная функция, существующая в нескольких версиях, из которых большинство имеет категорию изменчивости IMMUTABLE, но та версия, которая получает в качестве второго параметра значение типа timestamp with time zone, является стабильной (STABLE). Она не является постоянной (IMMUTABLE), поскольку ее результат зависит от параметров локализации сервера. Эта функция нам и нужна:

```
SELECT proname, proargtypes::regtype[], provolatile
FROM pg_proc
WHERE proname = 'date_part';
```

```
  proname  |                proargtypes                 | provolatile
-----------+--------------------------------------------+-------------
 date_part | [0:1]={text,"timestamp with time zone"}    | s
 date_part | [0:1]={text,interval}                      | i
 date_part | [0:1]={text,"time with time zone"}         | i
 date_part | [0:1]={text,"time without time zone"}      | i
 date_part | [0:1]={text,"timestamp without time zone"} | i
 date_part | [0:1]={text,date}                          | i
(6 строк)
```

Чтобы следующие запросы можно было правильно интерпретировать, получим текущую отметку времени:

```
SELECT CURRENT_TIMESTAMP;
       current_timestamp
-------------------------------
 2025-02-19 19:08:33.337829+07
(1 строка)
```

В следующем запросе на первый взгляд нет ничего особенного. Условие в предложении WHERE будет заведомо истинным:

```
SELECT * FROM generate_series( 1, 3 )
WHERE date_part( 'year', CURRENT_TIMESTAMP ) = '2025';
 generate_series
-----------------
               1
               2
               3
(3 строки)
```

Но теперь в плане выполнения запроса условие проверяется только один раз (One-Time Filter), а не для каждой строки, формируемой в предложении FROM. Поскольку условие не зависит от данных, содержащихся в конкретной строке, то три вызова функций CURRENT_TIMESTAMP и date_part заменяются одним.

```
                                    QUERY PLAN
-----------------------------------------------------------------------------------------------
 Result (actual rows=3 loops=1)
   One-Time Filter: (date_part('year'::text, CURRENT_TIMESTAMP) = '2025'::double prec...
   -> Function Scan on generate_series (actual rows=3 loops=1)
 Planning Time: 0.024 ms
 Execution Time: 0.019 ms
(5 строк)
```

Теперь зададим условие, которое будет заведомо ложным:

```
SELECT * FROM generate_series( 1, 3 )
WHERE date_part( 'year', CURRENT_TIMESTAMP ) = '1799';
 generate_series
-----------------
(0 строк)
```

Из узла Function Scan явствует, что *запланированного* обращения к функции generate_series фактически вообще не происходило: вместо ожидаемого элемента (actual ...) видим (never executed), а в узле Result фактическое значение показателя rows равно 0. Важно, что условие в предложении WHERE проверялось на этапе выполнения запроса. Отметим этот факт, потому что в принципе, как мы увидим в дальнейшем, возможна проверка условий и на этапе планирования.

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Result (actual rows=0 loops=1)
   One-Time Filter: (date_part('year'::text, CURRENT_TIMESTAMP) = '1799'::double prec...
   -> Function Scan on generate_series (never executed)
 Planning Time: 0.024 ms
 Execution Time: 0.014 ms
(5 строк)
```

Теперь обратимся к постоянным (IMMUTABLE) функциям. Опять нам поможет семейство перегруженных функций date_part. Большинство из них имеют категорию IMMUTABLE. Вот «наша» функция:

```
\df+ date_part text "timestamp without time zone"
Список функций
-[ RECORD 1 ]----------+---------------------------------
Схема                  | pg_catalog
Имя                    | date_part
Тип данных результата  | double precision
Типы данных аргументов | text, timestamp without time zone
Тип                    | функ.
Изменчивость           | постоянная
Параллельность         | безопасная
Владелец               | postgres
Безопасность           | вызывающего
Права доступа          |
Язык                   | internal
Внутреннее имя         | timestamp_part
Описание               | extract field from timestamp
```

Обратите внимание, что тип ее второго параметра — timestamp without time zone. Это важно для задания правильного значения аргумента при вызове функции в следующем запросе. Этот запрос отличается от двух предыдущих тем, что в нем не используется функция CURRENT_TIMESTAMP: ведь она является стабильной функцией, а нам сейчас нужны только постоянные. Вместо вызова этой функции мы передадим функции date_part в качестве аргумента константу соответствующего вида и типа. При этом важно, чтобы тип данных этого аргумента был timestamp without time zone. В этом запросе условие дает заведомо истинный результат:

```
SELECT * FROM generate_series( 1, 3 )
WHERE date_part( 'year', '2025-02-19 19:08:33.894271+07'::timestamp ) = '2025';
 generate_series
-----------------
               1
               2
               3
(3 строки)
```

Казалось бы, ничего особенного не произошло. Но если посмотреть план запроса, то мы увидим, что фильтрация строк теперь вообще отсутствует. Проверка условия в предложении WHERE была выполнена еще на этапе планирования.

Функция date_part — постоянная. В нашем запросе она не берет никаких данных из строк таблицы, значения ее аргументов — константы, поэтому ее результат будет *одним и тем же* для всех строк, сформированных в предложении FROM. Следовательно, можно вычислить функцию только один раз и на основе полученного результата принять решение, включать ли в выборку все сформированные строки или не включать ни одной — фильтрация в плане не требуется. В данном случае результат вычисления будет истинным, следовательно, в выборку попадут все строки:

```
                          QUERY PLAN
-------------------------------------------------------------
 Function Scan on generate_series (actual rows=3 loops=1)
 Planning Time: 0.025 ms
 Execution Time: 0.010 ms
(3 строки)
```

Если задать в этом запросе заведомо ложное условие, то получим другую картину, но опять же решение будет принято уже при планировании:

```
SELECT * FROM generate_series( 1, 3 )
WHERE date_part( 'year', '2025-02-19 19:08:33.894271+07'::timestamp ) = '1837';
 generate_series
-----------------
(0 строк)
```

Теперь даже не требуется сканировать строки, поскольку для планировщика очевидно, что все они будут отброшены. Поэтому в плане фигурирует лаконичная запись One-Time Filter: false, а в узле Result фактическое число строк оказалось равным нулю:

```
           QUERY PLAN
-------------------------------
 Result (actual rows=0 loops=1)
   One-Time Filter: false
 Planning Time: 0.034 ms
 Execution Time: 0.007 ms
(4 строки)
```

Прогнозное значение числа строк также будет равно нулю. Предлагаем читателю самостоятельно проверить это, выполнив команду EXPLAIN с параметром COSTS.

Назначая (обоснованно, конечно) функциям категорию IMMUTABLE, не нужно думать, что всегда удастся переложить часть работы по выполнению запроса на стадию планирования. В следующем запросе воспользуемся постоянной функцией string_to_array для преобразования символьной строки в массив символов. Значения параметра функции будут одинаковыми для всех строк выборки, и функция возвратит для всех них один и тот же результат, как и положено постоянной функции. Однако это не позволяет планировщику упростить план, исключив из него фильтрацию строк. Функция получает на вход не константу, а значение из текущей строки выборки, и планировщик заранее не знает, что все эти значения окажутся одинаковыми.

```
WITH databases( dbms ) AS
( VALUES ( 'PostgreSQL' ), ( 'PostgreSQL' ), ( 'PostgreSQL' )
)
SELECT dbms
FROM databases
WHERE string_to_array( dbms, NULL ) @> ARRAY[ 'S', 'Q', 'L' ];
```

```
    dbms
------------
 PostgreSQL
 PostgreSQL
 PostgreSQL
(3 строки)
```

Напомним, что если подзапрос из общего табличного выражения фигурирует в главном запросе только один раз (и не использует изменчивые функции), то по умолчанию такой подзапрос не материализуется, а встраивается в главный запрос:

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Values Scan on "*VALUES*" (actual rows=3 loops=1)
   Filter: (string_to_array(column1, NULL::text) @> '{S,Q,L}'::text[])
 Planning Time: 0.040 ms
 Execution Time: 0.015 ms
(4 строки)
```

От абстрактных примеров перейдем к более реалистичной ситуации. Предположим, что маркетологи нашей авиакомпании планируют ежедневно разыгрывать среди пассажиров поощрительные призы. Победители розыгрыша будут выбираться путем определения «счастливых» мест в салонах самолетов. В качестве пробного шага будет выбираться единый номер кресла для всех моделей самолетов. Конечно, у этого подхода есть недостаток: «счастливые» места небольшим самолетам будут доставаться гораздо реже, чем большим. Например, кресла G и H есть только в двух моделях самолетов, а J и K — лишь в одной. Это же замечание касается и количества рядов, которое варьируется от 6 до 51. Но, с другой стороны, небольшие самолеты летают гораздо чаще, чем большие. Поэтому если «счастливое» место все же окажется в салоне небольшого самолета, то охват пассажиров будет очень широким.

Для генерации случайных чисел мы не будем использовать стандартную функцию `random` (почему — станет понятно ниже), а обратимся к содержимому базы данных. Сформируем номер «счастливого» кресла для конкретной даты на основе количества пассажиров, перевезенных нашей авиакомпанией за данный день, то есть пассажиров рейсов, имеющих статус `Arrived`. Таким образом, этот номер должен формироваться не до интересующей нас даты, а *после* нее, возможно назавтра. При этом должна быть обеспечена повторяемость результата

при многократном выполнении запроса в разных отделениях компании, находящихся в разных часовых поясах.

Алгоритм формирования номера кресла будет таким: отдельно определить номер ряда и буквенный индекс, а затем объединить их. Для этого напишем функцию, которая возвращает целое число в диапазоне от единицы до заданного максимального значения:

```
CREATE OR REPLACE FUNCTION get_rand_num( dt date, max_num integer )
RETURNS integer AS
$$
  SELECT count( * ) % max_num + 1
  FROM boarding_passes AS bp
    JOIN flights AS f ON f.flight_id = bp.flight_id
  WHERE f.scheduled_departure::date = dt
    AND f.status = 'Arrived';
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

Функция `get_rand_num` получает в качестве первого параметра дату, для которой определяется случайное число (строго говоря, его, конечно, нельзя считать случайным). Вторым параметром является максимальное значение порождаемого числа. В запросе подсчитывается количество пассажиров, прибывших в указанный день. Остаток от деления увеличиваем на единицу, поскольку функция будет использоваться для формирования номера ряда кресел в салоне самолета, а эти номера начинаются не с нуля, а с единицы. Индекс в массиве буквенных обозначений мест также начинается с единицы.

Категория изменчивости этой функции будет `VOLATILE`, поскольку мы не указали иное.

Для решения поставленной задачи сначала нужно определить максимальный номер ряда кресел для всех моделей самолетов, а также максимально полный список буквенных индексов мест в ряду.

```
SELECT
  aircraft_code,
  max( left( seat_no, length( seat_no ) - 1 )::integer )
FROM seats
GROUP BY 1
ORDER BY 2 DESC;
```

```
 aircraft_code | max
---------------+-----
 773           |  51
 763           |  39
 321           |  31
 320           |  25
 CR2           |  23
 733           |  23
 319           |  21
 SU9           |  20
 CN1           |   6
(9 строк)
```

Поскольку мы выбираем одно место на весь самолет, примем еще одно упрощение: сформируем списки буквенных индексов без учета классов обслуживания (бизнес, комфорт или экономический). Это не помешает нам показать влияние категории изменчивости функций на получаемые результаты.

```
WITH distinct_letters ( aircraft_code, letter ) AS
( SELECT DISTINCT aircraft_code, right( seat_no, 1 )
  FROM seats
),
arrays_of_letters AS
( SELECT aircraft_code, array_agg( letter ORDER BY letter ) AS letters
  FROM distinct_letters
  GROUP BY aircraft_code
)
SELECT aircraft_code, letters
FROM arrays_of_letters
ORDER BY array_length( letters, 1 ) DESC, aircraft_code;
```

```
 aircraft_code |        letters
---------------+-----------------------
 773           | {A,B,C,D,E,F,G,H,J,K}
 763           | {A,B,C,D,E,F,G,H}
 319           | {A,B,C,D,E,F}
 320           | {A,B,C,D,E,F}
 321           | {A,B,C,D,E,F}
 733           | {A,B,C,D,E,F}
 SU9           | {A,C,D,E,F}
 CR2           | {A,B,C,D}
 CN1           | {A,B}
(9 строк)
```

Можно было, пожалуй, обойтись и более простым запросом, но тогда мы не получили бы столь полную картину распределения мест в разных моделях.

Вот этот упрощенный запрос:

```
SELECT array_agg( letter ) AS array_letters
FROM
  ( SELECT DISTINCT right( seat_no, 1 ) AS letter
    FROM seats
    ORDER BY letter
  ) AS unique_letters;
```

```
     array_letters
----------------------
 {A,B,C,D,E,F,G,H,J,K}
(1 строка)
```

Итак, теперь мы готовы к выполнению пробного запроса, который сформирует номер «счастливого» кресла. При вызовах функции get_rand_num будем использовать аргументы-константы, а не подзапросы, чтобы не усложнять интерпретацию плана запроса.

```
SELECT * FROM seats
WHERE seat_no = get_rand_num( '2017-07-28'::date, 51 )::text ||
    ( ARRAY[ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'K' ] )
      [ get_rand_num (
          '2017-07-28'::date,
          array_length( ARRAY[ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'K' ], 1 )
        )
      ];
```

```
 aircraft_code | seat_no | fare_conditions
---------------+---------+-----------------
 319           | 7F      | Economy
 320           | 7F      | Economy
 321           | 7F      | Business
 733           | 7F      | Economy
 SU9           | 7F      | Economy
(5 строк)
```

Нельзя просто сформировать номер кресла с помощью выражения, использованного в предложении WHERE, потому что такого места может не оказаться в салоне той или иной модели. Ведь в разных моделях разное число рядов кресел, да и ряды могут иметь разное их количество, например в бизнес-классе мест в ряду меньше, чем в экономическом. Приходится проверять наличие сформированного номера места в каждом салоне.

В принципе, возможна ситуация, когда при использовании предложенного способа не будет выбрано ни одного «счастливого» места. Например, может выпасть кресло в первом ряду с местом J, которого не существует ни в одном самолете, а значит, общее число отобранных строк будет равно нулю.

Предлагаем читателю самостоятельно модифицировать способ выбора «счастливого» места таким образом, чтобы подобных ситуаций не возникало. В качестве одной из идей можно предложить такую: изменить способ получения числа в функции get_rand_num так, чтобы при повторных вызовах эта функция возвращала другие значения. Номер кресла можно формировать в рекурсивном общем табличном выражении, а условием завершения процесса считать наличие такого кресла хотя бы в одной модели самолета. В главном запросе нужно будет отобрать из таблицы «Места» (seats) строки с полученным номером кресла.

А если вместо функции get_rand_num воспользоваться функцией random? Она может получать на вход границы допустимого диапазона значений:

```
SELECT * FROM seats
WHERE seat_no = random( 1, 51 )::text ||
    ( ARRAY[ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'K' ] )
      [ random(
          1,
          array_length( ARRAY[ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'K' ], 1 )
        )
      ];
```

Теперь запрос дает принципиально другой результат — выбираются разные номера кресел:

```
 aircraft_code | seat_no | fare_conditions
---------------+---------+-----------------
 319           | 18A     | Economy
 773           | 17C     | Economy
 773           | 17E     | Economy
 773           | 29H     | Economy
 773           | 38K     | Economy
(5 строк)
```

Такая картина объясняется тем, что выражение в условии WHERE, в котором используется функция random, дает для каждой строки разные значения. Однако

такая ситуация нас не устраивает, потому что изначально мы не планировали формировать несколько номеров кресел для одной модели самолета, как это получилось в приведенном примере. К тому же при использовании функции random мы не сможем обеспечить повторяемость результатов, о которой шла речь в постановке задачи, даже если поместим ее вызов в общее табличное выражение.

Вернемся к запросу с функцией get_rand_num и посмотрим его план. Число отобранных строк в плане в общем случае может различаться в зависимости от выбранной даты.

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Seq Scan on seats (actual rows=5 loops=1)
   Filter: ((seat_no)::text = ((get_rand_num('2017-07-28'::date, 51))::text ||
   ('{A,B,C,D,E,F,G,H,J,K}'::text[])[get_rand_num('2017-07-28'::date, 10)]))
   Rows Removed by Filter: 1334
 Planning Time: 0.148 ms
 Execution Time: 23431.043 ms
(6 строк)
```

Используется последовательный просмотр таблицы. Функция array_length (она постоянная, STABLE) вызывается еще на стадии планирования запроса, поэтому в плане во второй вызов функции get_rand_num подставляется длина массива в виде константы — 10. Выражение в условии фильтрации заведомо дает один и тот же результат для каждой строки, но поскольку категория изменчивости функции get_rand_num — VOLATILE, она все равно вычисляется для каждой строки, то есть 1339 раз. Поэтому запрос выполняется довольно долго.

В функции get_rand_num нет операций, модифицирующих базу данных. В ней используется только постоянная функция count, поэтому она вполне соответствует требованиям стабильной категории (STABLE). Давайте изменим это свойство функции:

```
ALTER FUNCTION get_rand_num STABLE;
ALTER FUNCTION
```

Выполним тот же самый запрос и получим его план:

```
                                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Index Scan using seats_pkey on seats (actual rows=5 loops=1)
   Index Cond: ((seat_no)::text = ((get_rand_num('2017-07-28'::date, 51))::text ||
   ('{A,B,C,D,E,F,G,H,J,K}'::text[])[get_rand_num('2017-07-28'::date, 10)]))
 Planning Time: 21.736 ms
 Execution Time: 18.043 ms
(5 строк)
```

Выполнение запроса ускорилось на три порядка. Теперь планировщик решил воспользоваться поиском по индексу вместо последовательного просмотра с фильтром. Почему это произошло? В первом запросе индекс не использовался, значит, дело только в новой категории изменчивости функции `get_rand_num`. В разделе документации 36.7 «Категории изменчивости функций» сказано, что при поиске по индексу выражение, которое участвует в операции сравнения, вычисляется только один раз, а не для каждой строки. Но чтобы однократное вычисление выражения позволяло выполнить корректный поиск, оно должно давать один и тот же результат для всех строк запроса. А именно это требование является необходимым для назначения стабильной (`STABLE`) категории.

При изучении плана запроса следует обратить внимание на то, что используется индекс `seats_pkey`, который создан по столбцам `aircraft_code` и `seat_no`. При этом условие поиска сформировано только по второму столбцу индекса, а по первому столбцу, `aircraft_code`, ограничения нет. Тем не менее, как сказано в разделе документации 11.3 «Составные индексы», PostgreSQL может использовать индекс, построенный на основе B-дерева, в условиях с любым подмножеством столбцов индекса, не обязательно лидирующих. Но, конечно, поиск окажется не столь эффективным, потому что придется просканировать весь индекс. Поэтому в подобных случаях планировщик зачастую выбирает не индексный поиск, а последовательный просмотр.

Таким образом, функция `get_rand_num` была вычислена двукратно (ведь в предложении `WHERE` два ее вызова с разными параметрами), причем на стадии планирования запроса. Этим объясняется довольно большое время планирования. А время выполнения, несмотря на использование индекса, также получилось заметным из-за его полного сканирования.

А что будет, если назначить функции `get_rand_num` категорию изменчивости `IMMUTABLE`?

Давайте так и сделаем, а затем обсудим полученные результаты:

```
ALTER FUNCTION get_rand_num IMMUTABLE;
ALTER FUNCTION
```

Теперь план станет таким:

```
                QUERY PLAN
-----------------------------------------
 Seq Scan on seats (actual rows=5 loops=1)
   Filter: ((seat_no)::text = '7F'::text)
   Rows Removed by Filter: 1334
 Planning Time: 20.404 ms
 Execution Time: 0.106 ms
(5 строк)
```

Теперь планировщик выбрал последовательный просмотр, отказавшись от использования индекса (и это значительно ускорило работу), и опять все выражение в условии WHERE, в котором используется функция get_rand_num, было вычислено на стадии планирования запроса. В условие фильтрации подставляется готовое значение.

Обратите внимание, что если воспользоваться командой EXPLAIN без параметра ANALYZE, то есть фактически запрос не выполнять, все равно в план будет подставлено вычисленное значение функции:

```
                QUERY PLAN
-----------------------------------------
 Seq Scan on seats
   Filter: ((seat_no)::text = '7F'::text)
(2 строки)
```

Запрос выполняется, на первый взгляд, корректно. Однако является ли наша функция get_rand_num в действительности постоянной? Нет, не является. Но система этого не заметила и позволила назначить категорию изменчивости IMMUTABLE, хотя результат функции get_rand_num при одних и тех же значениях аргументов может изменяться в зависимости от содержимого базы данных.

Давайте проверим это.

```
BEGIN;
BEGIN
```

```
SELECT * FROM seats
WHERE seat_no = get_rand_num( '2017-07-28'::date, 51 )::text ||
    ( ARRAY[ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'K' ] )
      [ get_rand_num(
          '2017-07-28'::date,
          array_length( ARRAY[ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'K' ], 1 )
        )
      ];
 aircraft_code | seat_no | fare_conditions
---------------+---------+-----------------
 319           | 7F      | Economy
 320           | 7F      | Economy
 321           | 7F      | Business
 733           | 7F      | Economy
 SU9           | 7F      | Economy
(5 строк)
DELETE FROM boarding_passes WHERE flight_id = 312;
DELETE 151
SELECT * FROM seats
WHERE seat_no = get_rand_num( '2017-07-28'::date, 51 )::text ||
     ( ARRAY[ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'K' ] )
       [ get_rand_num(
           '2017-07-28'::date,
           array_length( ARRAY[ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'K' ], 1 )
         )
       ];
 aircraft_code | seat_no | fare_conditions
---------------+---------+-----------------
 319           | 9E      | Economy
 320           | 9E      | Economy
 321           | 9E      | Economy
 733           | 9E      | Economy
 SU9           | 9E      | Economy
(5 строк)
ROLLBACK;
ROLLBACK
```

Мы сохранили те же значения аргументов, что и в предыдущих экспериментах, но теперь сформирован другой номер «счастливого» кресла. Можно сказать, что результат корректный, но он не соответствует ожиданиям от постоянной функции, которая должна при одних и тех же значениях аргументов возвращать один и тот же результат. Наверное, в реальной работе удаление строк из таблицы «Посадочные талоны» (boarding_passes) или добавление строк в нее, когда

текущая дата уже прошла, было бы маловероятным. Тем не менее при строгом следовании положениям документации назначать функции `get_rand_num` категорию `IMMUTABLE` нельзя. Отступая от правил, нужно учитывать последствия.

В общем случае неправомерное назначение функции категории `IMMUTABLE` может привести к проблеме, которая не проявляется в однократных интерактивных запросах (как мы это видели), но может проявиться при использовании *подготовленных операторов* (prepared statements).

Процесс выполнения SQL-оператора включает ряд стадий, в том числе разбор запроса, планирование и непосредственно исполнение. В PostgreSQL существует возможность подготовить оператор, то есть разобрать его и сохранить на стороне сервера. В языке SQL для этого служит команда `PREPARE`. Будучи один раз подготовленным, такой оператор затем может выполняться многократно с помощью команды `EXECUTE`, которой останется спланировать и исполнить запрос.

Подготовленные операторы позволяют исключить повторный разбор запроса, а в ряде случаев и значительно уменьшить время его планирования. Дело в том, что в команде `PREPARE` можно задать параметры, которые будут передаваться оператору при его исполнении. Если параметры не предусмотрены, PostgreSQL сохраняет и план запроса (такой план называется *общим*). План может быть сохранен и в том случае, когда параметры заданы, но перед принятием такого решения как минимум первые пять раз запрос исполняется со *специализированными* планами, учитывающими значения параметров. Если превышение стоимости общего плана над средней стоимостью специализированных планов не является значительным, в дальнейшем используется сохраненный общий план. В противном случае снова выполняется планирование в надежде, что затраты на него окупятся за счет сокращения времени исполнения.

Таким образом, при использовании сохраненного плана значительно сокращается (практически до нуля) время, затрачиваемое на стадии планирования. Для многократно выполняемых запросов, в которых время планирования велико или сравнимо со временем исполнения, экономия за счет однократного планирования может быть существенной.

Опасность в том, что если в подготовленном операторе окажется функция, имеющая категорию изменчивости `IMMUTABLE`, но на самом деле не соответствующая этой категории, то при повторном выполнении этого оператора планировщик может воспользоваться устаревшим значением функции, которое было

вычислено на стадии планирования и попало в сохраненный план подготовленного оператора.

Покажем это также на примере определения номера «счастливого» кресла. Создадим подготовленный оператор:

```
PREPARE get_happy_seat AS
SELECT * FROM seats
WHERE seat_no = get_rand_num( '2017-07-28'::date, 51 )::text ||
    ( ARRAY[ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'K' ] )
      [ get_rand_num(
          '2017-07-28'::date,
          array_length( ARRAY[ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'K' ], 1 )
        )
      ];
PREPARE
```

Сразу после создания подготовленного оператора выполним его и получим план:

```
EXPLAIN ( analyze, costs off, timing off )
EXECUTE get_happy_seat;
               QUERY PLAN
-----------------------------------------
 Seq Scan on seats (actual rows=5 loops=1)
   Filter: ((seat_no)::text = '7F'::text)
   Rows Removed by Filter: 1334
 Planning Time: 21.326 ms
 Execution Time: 0.121 ms
(5 строк)
```

Проделаем то же самое еще раз:

```
EXPLAIN ( analyze, costs off, timing off )
EXECUTE get_happy_seat;
               QUERY PLAN
-----------------------------------------
 Seq Scan on seats (actual rows=5 loops=1)
   Filter: ((seat_no)::text = '7F'::text)
   Rows Removed by Filter: 1334
 Planning Time: 0.005 ms
 Execution Time: 0.126 ms
(5 строк)
```

Самое интересное здесь — время планирования, которое во втором запросе сократилось практически до нуля. Значение критерия отбора строк осталось неизменным, как и ожидалось. Давайте выполним этот же запрос без команды EXPLAIN. Как видим, номер кресла останется таким же, как и был:

```
EXECUTE get_happy_seat;
 aircraft_code | seat_no | fare_conditions
---------------+---------+-----------------
 319           | 7F      | Economy
 320           | 7F      | Economy
 321           | 7F      | Business
 733           | 7F      | Economy
 SU9           | 7F      | Economy
(5 строк)
```

Увидеть подготовленные в текущем сеансе операторы можно в системном каталоге pg_prepared_statements (см. раздел 52.15 «pg_prepared_statements» документации).

```
SELECT * FROM pg_prepared_statements \gx
-[ RECORD 1 ]---+--------------------------------------------------------------------------
name            | get_happy_seat
statement       | PREPARE get_happy_seat AS                                               +
                | SELECT * FROM seats                                                     +
                | WHERE seat_no = get_rand_num( '2017-07-28'::date, 51 )::text ||        +
                |     ( ARRAY[ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'K' ] )      +
                |       [ get_rand_num(                                                   +
                |           '2017-07-28'::date,                                          +
                |            array_length( ARRAY[ 'A', 'B', 'C', 'D', 'E', 'F', 'G', '...+
                |          )                                                              +
                |       ];
prepare_time    | 2025-02-19 19:08:35.207134+07
parameter_types | {}
result_types    | {character,"character varying","character varying"}
from_sql        | t
generic_plans   | 3
custom_plans    | 0
```

Обратите внимание на значение поля generic_plans, равное трем. Это число исполнений запроса с сохраненным общим планом.

А теперь покажем, что вычисленное значение функции, сохраненное в плане, может оказаться устаревшим.

```
BEGIN;
BEGIN
DELETE FROM boarding_passes
WHERE flight_id = 6477;
DELETE 46
EXECUTE get_happy_seat;
```

Выборка не изменилась, поскольку используется сохраненное в плане значение функции:

```
 aircraft_code | seat_no | fare_conditions
---------------+---------+-----------------
 319           | 7F      | Economy
 320           | 7F      | Economy
 321           | 7F      | Business
 733           | 7F      | Economy
 SU9           | 7F      | Economy
(5 строк)
ROLLBACK;
ROLLBACK
```

Подготовленный оператор доступен в текущем сеансе работы с сервером и сохраняется до его завершения, но можно удалить оператор и раньше. Давайте так и сделаем, чтобы провести еще один эксперимент:

```
DEALLOCATE get_happy_seat;
DEALLOCATE
```

Выше уже было сказано, что подготовленный оператор может иметь параметры. Для их задания достаточно указать тип данных. В тексте запроса для ссылки на параметры служат идентификаторы вида $1, $2 и т. д.

```
PREPARE get_happy_seat ( date, integer ) AS
SELECT * FROM seats
WHERE seat_no = get_rand_num( $1, $2 )::text ||
    ( ARRAY[ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'K' ] )
      [ get_rand_num(
          $1,
          array_length( ARRAY[ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J', 'K' ], 1 )
        )
      ];
PREPARE
```

Выполним оператор с параметром:

```
EXPLAIN ( analyze, costs off, timing off )
EXECUTE get_happy_seat ( '2017-07-28'::date, 51 );
                QUERY PLAN
-------------------------------------------
 Seq Scan on seats (actual rows=5 loops=1)
   Filter: ((seat_no)::text = '7F'::text)
   Rows Removed by Filter: 1334
 Planning Time: 21.003 ms
 Execution Time: 0.104 ms
(5 строк)
```

Проделаем то же самое еще раз, оставив значение параметра неизменным:

```
EXPLAIN ( analyze, costs off, timing off )
EXECUTE get_happy_seat ( '2017-07-28'::date, 51 );
                QUERY PLAN
-------------------------------------------
 Seq Scan on seats (actual rows=5 loops=1)
   Filter: ((seat_no)::text = '7F'::text)
   Rows Removed by Filter: 1334
 Planning Time: 20.447 ms
 Execution Time: 0.122 ms
(5 строк)
```

Теперь, как и прежде, функция вычисляется на стадии планирования запроса, но наличие параметра приводит к тому, что это делается при *каждом* выполнении подготовленного оператора. Выигрыша во времени планирования уже нет, поскольку план (пока) не сохраняется.

Будет ли работа запроса корректной, если модифицировать базу данных, сохранив неизменными значения аргументов?

```
BEGIN;
BEGIN
DELETE FROM boarding_passes
WHERE flight_id = 6477;
DELETE 46
EXECUTE get_happy_seat ( '2017-07-28'::date, 51 );
```

Да, теперь выбирается другой номер кресла:

```
 aircraft_code | seat_no | fare_conditions
---------------+---------+-----------------
 773           | 12K     | Comfort
(1 строка)

ROLLBACK;
ROLLBACK
```

В нашем примере при использовании постоянной функции в подготовленном операторе с параметрами планирование запроса производилось при каждом выполнении оператора. Поэтому получались корректные результаты, хотя фактически функция не является постоянной. Однако всегда рассчитывать на благоприятный исход нельзя. На принятие решения планировщиком о сохранении плана или повторном планировании влияет ряд факторов и параметров сервера, значения которых можно переопределить.

Предлагаем читателю выполнить подготовленный оператор несколько раз, изменяя параметр-дату, проверить, как при этом изменятся значения столбцов `generic_plans` и `custom_plans` в системном представлении `pg_prepared_statements`, и попытаться объяснить увиденное.

Подведем итог проведенных экспериментов.

Когда функция `get_rand_num` имела категорию изменчивости `VOLATILE`, запрос выполнялся корректно, но нерационально: оптимизатор не мог заменить многократные вызовы функции, заведомо дающие один и тот же результат, одним вызовом.

Когда функция `get_rand_num` получила категорию изменчивости `STABLE`, которой она в точности соответствует, оптимизатор смог заменить многократные вызовы функции одним вызовом (точнее двумя, поскольку в предложении `WHERE` их два) и перенести вычисление функции с этапа исполнения запроса на этап планирования. В результате выполнение запроса значительно ускорилось.

После назначения функции `get_rand_num` категории изменчивости `IMMUTABLE` общее время выполнения запроса еще уменьшилось. Хотя категория изменчивости теперь более строгая, чем та, которой функция фактически соответствует, результат получался корректный в силу особенностей конкретной ситуации.

При использовании подготовленных операторов, когда в запросе есть постоянная функция, не соответствующая этой категории изменчивости, важен факт наличия параметров у оператора. При их отсутствии план выполнения запроса при повторном выполнении не изменится. Поэтому значение функции, вычисленное при его первом выполнении, будет использоваться и при повторном. Это может привести к тому, что запрос станет выдавать некорректные результаты. Если же подготовленный оператор имеет параметры, то результат в общем случае может оказаться как корректным, так и некорректным.

Провести эксперимент с использованием стабильной функции в подготовленном операторе читатель может самостоятельно.

Подробно о подготовленных операторах можно прочитать в описаниях команд `PREPARE`, `EXECUTE` и `DEALLOCATE`, приведенных в документации.

5.6.2. Видимость изменений

Мы рассмотрели влияние категории изменчивости функций на выбор планировщиком плана выполнения запросов. Но возникает еще один вопрос: видит ли функция изменения, сделанные запросом, из которого она вызывается? Ответ на него также зависит от категории изменчивости функции, а вот от уровня изоляции транзакции, в которой выполняется запрос, не зависит.

В разделе документации 36.7 «Категории изменчивости функций» сказано следующее: «У функций, написанных на SQL или на любом другом стандартном процедурном языке, есть еще одно важное свойство, определяемое характеристикой изменчивости, а именно видимость изменений, произведенных 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 (см. главу 13 «Управление конкурентным доступом» документации): `STABLE` и `IMMUTABLE` используют снимок, полученный в начале вызывающего запроса, тогда как функции `VOLATILE` получают свежий снимок в начале каждого запроса, который они выполняют».

В качестве иллюстрации расширим демобазу учетом числа мест багажа и его общего веса при регистрации каждого пассажира. К завершению регистрации будут получены итоговые значения, необходимые службе, доставляющей багаж на борт самолета, и пилотам для расчета взлетных характеристик.

Для реализации требований создадим новую таблицу «Багаж» (luggage). В ней для каждого пассажира будет столько строк, сколько мест багажа он провозит.

```
CREATE TABLE luggage
( flight_id integer,
  boarding_no integer,
  piece_no smallint NOT NULL CHECK ( piece_no > 0 ),
  weight numeric( 3, 1 ) CHECK ( weight > 0.0 ),
  PRIMARY KEY ( flight_id, boarding_no, piece_no ),
  FOREIGN KEY ( flight_id, boarding_no )
    REFERENCES boarding_passes ( flight_id, boarding_no ) ON DELETE CASCADE
);
```

```
CREATE TABLE
```

Поскольку багаж привязан к посадочному талону, внешний ключ таблицы «Багаж» (luggage) будет ссылаться на уникальный ключ таблицы «Посадочные талоны» (boarding_passes). А из-за того что пассажир может провозить несколько мест багажа, первичный ключ таблицы «Багаж» (luggage) будет включать и столбец piece_no — порядковый номер места багажа, уникальный для конкретного пассажира. Столбец weight содержит вес данного места багажа.

Нам потребуются две функции: первая будет вводить в таблицы сведения об очередном пассажире, включая его багаж, а вторая — подводить текущие итоги. Начнем с первой:

```
CREATE OR REPLACE FUNCTION boarding
( flight_id integer,
  boarding_no integer,
  VARIADIC weights numeric[]
)
RETURNS void AS
$$
  WITH luggage_pieces AS
  ( SELECT boarding.flight_id, boarding.boarding_no, num, weight
    FROM unnest( weights ) WITH ORDINALITY AS lw( weight, num )
  )
  INSERT INTO luggage ( flight_id, boarding_no, piece_no, weight )
  SELECT boarding.flight_id, boarding.boarding_no, num, weight
  FROM luggage_pieces;
$$
LANGUAGE sql VOLATILE;
```

```
CREATE FUNCTION
```

Функция принимает идентификатор рейса, номер посадочного талона и веса всех мест багажа данного пассажира. Число мест багажа в этой версии функции может быть произвольным (задачу ограничения этого числа читатель может решить в качестве упражнения). Запрос с конструкцией WITH преобразует массив весов во временную таблицу, которая используется в качестве источника данных командой INSERT. Имена части параметров совпадают с именами столбцов; таким параметрам в тексте функции предшествует ее имя.

Поскольку функция должна вносить изменения в базу данных, ей назначена категория изменчивости VOLATILE.

Вторая функция — boarding_info — будет собирать сведения о количестве зарегистрированных пассажиров, общем числе мест багажа и его общем весе:

```
CREATE OR REPLACE FUNCTION boarding_info
( INOUT flight_id integer,
  OUT total_passengers bigint,
  OUT total_luggage_pieces bigint,
  OUT total_luggage_weight numeric
)
RETURNS record AS
$$
  WITH boarding_pass_info AS
  ( SELECT count( * ) AS total_passengers
    FROM boarding_passes
    WHERE flight_id = boarding_info.flight_id
  ),
  luggage_info AS
  ( SELECT count( * ) AS total_luggage_pieces, sum( weight ) AS total_luggage_weight
    FROM luggage
    WHERE flight_id = boarding_info.flight_id
  )
  SELECT
    flight_id,
    bpi.total_passengers,
    li.total_luggage_pieces,
    li.total_luggage_weight
  FROM
    boarding_pass_info AS bpi,
    luggage_info AS li;
$$
LANGUAGE sql STABLE;
CREATE FUNCTION
```

Нам пригодился параметр flight_id с модификатором INOUT, позволяющим не только задать идентификатор конкретного рейса для сбора сведений, но и вывести этот же идентификатор для повышения наглядности результата работы функции. Поскольку функция формирует всего одну строку, типом результата будет record, а не SETOF record. По той же причине предложение RETURNS record можно было не указывать. Тем не менее при просмотре описания функции в поле «Тип данных результата» мы все равно увидим record.

В разделе документации 36.7 «Категории изменчивости функций» сказано, что функцию, содержащую только команды SELECT, можно безопасно пометить как STABLE, даже если она выбирает данные из таблиц, которые могут быть изменены параллельными запросами. Так мы и сделали.

У этой функции совпадают имена выходных параметров и имена соответствующих им столбцов в запросе, который формирует выходные значения. Напомним, что такое совпадение не является обязательным.

Итак, весь необходимый инструментарий готов — можно приступать к экспериментам.

Давайте выберем какой-нибудь рейс, на который уже открыта регистрация билетов (статус рейса — On Time), но еще не зарегистрирован ни один пассажир. Пусть это будет ближайший рейс из Красноярска в Москву:

```
SELECT
  f.flight_id,
  f.flight_no,
  f.scheduled_departure,
  f.scheduled_arrival,
  f.departure_airport, a1.airport_name, a1.city,
  f.arrival_airport, a2.airport_name, a2.city,
  f.status,
  f.aircraft_code,
  ac.model
FROM flights AS f
  JOIN airports AS a1 ON f.departure_airport = a1.airport_code
  JOIN airports AS a2 ON f.arrival_airport = a2.airport_code
  JOIN aircrafts AS ac ON f.aircraft_code = ac.aircraft_code
WHERE a1.city = 'Красноярск'
  AND a2.city = 'Москва'
  AND f.status = 'On Time'
ORDER BY f.scheduled_departure LIMIT 1 \gx
```

```
-[ RECORD 1 ]-------+----------------------
flight_id           | 13841
flight_no           | PG0548
scheduled_departure | 2017-08-16 12:40:00+07
scheduled_arrival   | 2017-08-16 17:05:00+07
departure_airport   | KJA
airport_name        | Емельяново
city                | Красноярск
arrival_airport     | SVO
airport_name        | Шереметьево
city                | Москва
status              | On Time
aircraft_code       | 319
model               | Аэробус A319-100
```

На настоящий момент число оформленных посадочных талонов равно нулю:

```
SELECT count( * )
FROM boarding_passes
WHERE flight_id = 13841;
 count
-------
     0
(1 строка)
```

Для регистрации нам потребуются номера билетов, купленных на этот рейс:

```
SELECT ticket_no, fare_conditions
FROM ticket_flights
WHERE flight_id = 13841
ORDER BY ticket_no;
   ticket_no   | fare_conditions
---------------+-----------------
 0005432003745 | Economy
 0005432003746 | Economy
 0005432003747 | Economy
 0005433815389 | Economy
 0005433815414 | Economy
 0005433815415 | Economy
 0005433846718 | Economy
 0005433846800 | Business
 ...
 0005435282865 | Business
 0005435282866 | Economy
(91 строка)
```

Необходимо также знать компоновку салона самолета «Аэробус A-319» для корректного размещения регистрируемых пассажиров в соответствии с классом обслуживания, указанным в билете:

```
SELECT seat_no, fare_conditions
FROM seats
WHERE aircraft_code = '319'
ORDER BY left( seat_no, length( seat_no ) - 1 )::integer, right( seat_no, 1 );
 seat_no | fare_conditions
---------+----------------
 1A      | Business
 1C      | Business
 1D      | Business
 1F      | Business
 ...
 21F     | Economy
(116 строк)
```

Теперь у нас есть вся необходимая информация о выбранном рейсе — можно приступать к регистрации пассажиров. Цель проведения экспериментов — изучить влияние сочетания категории изменчивости функции и уровня изоляции транзакции на возможность функции boarding_info видеть изменения, выполненные запросом, из которого она вызвана. Функция boarding носит вспомогательный характер.

Для проведения первого эксперимента выберем уровень изоляции транзакций Read Committed, тот, который используется по умолчанию. Функция boarding_info в этом эксперименте имеет категорию изменчивости STABLE.

Начинаем работу.

```
BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
```

В главном запросе функция boarding вставляет строки в таблицу «Багаж» (luggage). Значения внешних ключей в этой таблице должны быть согласованы с таблицей «Посадочные талоны» (boarding_passes). Это достигается за счет того, что в качестве параметров функций используются значения полей, возвращенные предложением RETURNING общего табличного выражения.

```
WITH make_boarding AS
( INSERT INTO boarding_passes ( ticket_no, flight_id, boarding_no, seat_no )
  VALUES ( '0005433846800', 13841, 1, '1A' )
  RETURNING *
)
SELECT
  bi.flight_id,
  bi.total_passengers,
  bi.total_luggage_pieces,
  bi.total_luggage_weight
FROM
  make_boarding AS mb,
  boarding( mb.flight_id, mb.boarding_no, 15.0, 12.5 ) AS b,
  boarding_info( mb.flight_id ) AS bi \gx
-[ RECORD 1 ]--------+------
flight_id            | 13841
total_passengers     | 0
total_luggage_pieces | 0
total_luggage_weight |
```

Видим, что значения всех итоговых показателей, которые вычисляет функция boarding_info, равны нулю или NULL, то есть текущая операция регистрации пассажира на рейс (а она является первой) не учтена функцией.

Да, действительно, стабильная функция *не видит* изменений, которые произвел еще не завершившийся запрос, в котором она вызвана. Как выше уже было сказано, это объясняется тем, что такая функция использует тот же снимок базы данных, что и сам запрос (и также все подзапросы в конструкции WITH), поэтому она видит состояние базы данных на момент начала выполнения запроса. В разделе документации 7.8.4 «Изменение данных в WITH» сказано, что основной запрос и подзапросы в конструкции WITH выполняются одновременно, взаимный порядок их выполнения не определен. Определенным является использование ими одного снимка данных.

Вот что получилось в базе данных:

```
SELECT * FROM boarding_passes
WHERE flight_id = 13841;
   ticket_no   | flight_id | boarding_no | seat_no
---------------+-----------+-------------+---------
 0005433846800 |     13841 |           1 | 1A
(1 строка)
```

```
SELECT * FROM luggage;
 flight_id | boarding_no | piece_no | weight
-----------+-------------+----------+--------
     13841 |           1 |        1 |   15.0
     13841 |           1 |        2 |   12.5
(2 строки)
```

Заметим попутно, что функция boarding смогла успешно добавить в таблицу «Багаж» (luggage) строки, связанные по внешнему ключу с таблицей «Посадочные талоны» (boarding_passes), хотя взаимный порядок выполнения подзапроса в конструкции WITH и главного запроса не определен. Успех объясняется тем, что проверка ограничения внешнего ключа по умолчанию выполняется сразу после завершения запроса. Это сказано в секции «DEFERRABLE / NOT DEFERRABLE» описания команды CREATE TABLE, приведенного в документации. Аналогичная ситуация рассматривалась в разделе 2.4 «Модификация данных в общем табличном выражении» (с. 69).

А что будет, если вызвать функцию boarding_info в отдельном запросе в этой же транзакции?

```
SELECT
  bi.flight_id,
  bi.total_passengers,
  bi.total_luggage_pieces,
  bi.total_luggage_weight
FROM boarding_info( 13841 ) AS bi \gx
```

Изменения, произведенные завершившимся запросом текущей транзакции, стабильная (и любая другая) функция, конечно же, видит:

```
-[ RECORD 1 ]--------+------
flight_id            | 13841
total_passengers     | 1
total_luggage_pieces | 2
total_luggage_weight | 27.5
```

Если бы мы в текущей транзакции вместо этого проверочного запроса вызвали бы функцию boarding_info, она уже увидела бы изменения базы данных, произведенные предыдущим — завершившимся — запросом этой же транзакции.

```
END;
COMMIT
```

Для получения полной картины нужно провести этот же эксперимент с использованием уровней изоляции транзакций Repeatable Read и Serializable. Читатель может проделать это самостоятельно, чтобы убедиться, что будут получены совершенно такие же результаты, как и на уровне изоляции Read Committed.

Перед повторением эксперимента нужно привести базу данных в исходное состояние, чтобы не допустить дублирования данных: удалить строки из таблицы «Посадочные талоны» (boarding_passes). А каскадное удаление строк из таблицы «Багаж» (luggage) выполнит PostgreSQL.

```
DELETE FROM boarding_passes
WHERE flight_id = 13841;
DELETE 1
```

Функция boarding_info в проведенных экспериментах имела категорию изменчивости STABLE. Давайте назначим ей категорию VOLATILE и опять проведем эксперименты при разных уровнях изоляции транзакций. Начнем с уровня изоляции Read Committed:

```
ALTER FUNCTION boarding_info VOLATILE;
ALTER FUNCTION
BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
WITH make_boarding AS
( INSERT INTO boarding_passes ( ticket_no, flight_id,
                                boarding_no, seat_no )
  VALUES ( '0005433846800', 13841, 1, '1A' )
  RETURNING *
)
SELECT bi.flight_id, bi.total_passengers,
       bi.total_luggage_pieces, bi.total_luggage_weight
FROM make_boarding AS mb,
     boarding( mb.flight_id, mb.boarding_no, 15.0, 12.5 ) AS b,
     boarding_info( mb.flight_id ) AS bi \gx
-[ RECORD 1 ]--------+------
flight_id           | 13841
total_passengers    | 1
total_luggage_pieces | 2
total_luggage_weight | 27.5
END;
COMMIT
```

Главный результат этого эксперимента заключается в том, что теперь функции boarding_info стали видны изменения, произведенные в базе данных тем запросом, из которого она была вызвана.

Предлагаем читателю самостоятельно убедиться, что на уровнях изоляций транзакций Repeatable Read и Serializable будут получены точно такие же результаты, как и на уровне изоляции Read Committed.

Таким образом, можно сделать вывод, что на всех уровнях изоляции транзакций изменчивая (VOLATILE) функция видит изменения, произведенные в базе данных запросом, из которого она вызвана, а стабильная (STABLE) — нет.

Мы не проводили эксперименты с постоянной (IMMUTABLE) функцией. Предлагаем читателю выполнить их самостоятельно, чтобы убедиться, что постоянная функция ведет себя так же, как стабильная.

Вопрос о том, видит ли изменчивая функция изменения, произведенные конкурентной транзакцией, рассмотрен в упражнении 18 (с. 403).

5.7. Дополнительные сведения о функциях

5.7.1. Подстановка кода функций в запрос

Давайте рассмотрим простую функцию, которая подсчитывает число пассажиров, перевезенных по каждому маршруту за весь период времени, представленный в базе данных «Авиаперевозки»:

```
CREATE OR REPLACE FUNCTION count_passengers( OUT f_no char, OUT pass_num bigint )
RETURNS SETOF record AS
$$
  SELECT flight_no, count( * )
  FROM flights AS f
    JOIN boarding_passes AS bp ON bp.flight_id = f.flight_id
  WHERE status IN ( 'Departed', 'Arrived' )
  GROUP BY flight_no;
$$
LANGUAGE sql VOLATILE;
CREATE FUNCTION
```

Проведем ряд экспериментов, включив секундомер. Начнем с запроса, выводящего сведения по всем маршрутам:

```
\timing on
Секундомер включён.
SELECT f_no, pass_num
FROM count_passengers()
ORDER BY pass_num DESC;
  f_no  | pass_num
--------+----------
 PG0222 |     9550
 PG0224 |     9362
 PG0225 |     9294
 ...
 PG0098 |        8
 PG0590 |        4
(470 строк)
Время: 205,114 мс
```

Получить один маршрут, скажем Оренбург — Уфа, можно так:

```
SELECT f_no, pass_num
FROM count_passengers()
WHERE f_no = 'PG0149';
  f_no  | pass_num
--------+----------
 PG0149 |     1333
(1 строка)
Время: 201,982 мс
```

Оказывается, что выборка лишь одного маршрута занимает примерно такое же время, как и выборка всех маршрутов. План запроса показывает причину: сначала выполняется функция, а потом из всего результата ее выполнения выбирается одна строка.

```
EXPLAIN ( costs off ) SELECT f_no, pass_num
FROM count_passengers()
WHERE f_no = 'PG0149';
             QUERY PLAN
------------------------------------
 Function Scan on count_passengers
   Filter: (f_no = 'PG0149'::bpchar)
(2 строки)
```

Можно ли каким-то образом ускорить выполнение запроса? Давайте назначим этой функции категорию изменчивости STABLE. Согласно разделу документации 36.7 «Категории изменчивости функций» это правомерное решение, поскольку функция не изменяет базу данных.

```
ALTER FUNCTION count_passengers STABLE;
ALTER FUNCTION
```

Каким теперь будет план запроса?

```
EXPLAIN ( costs off ) SELECT f_no, pass_num
FROM count_passengers()
WHERE f_no = 'PG0149';
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Subquery Scan on "*SELECT*"
   -> HashAggregate
       Group Key: f.flight_no
       -> Nested Loop
           -> Bitmap Heap Scan on flights f
               Recheck Cond: ((flight_no)::bpchar = 'PG0149'::bpchar)
               Filter: ((status)::text = ANY ('{Departed,Arrived}'::text[]))
               -> Bitmap Index Scan on flights_flight_no_scheduled_departure_key
                   Index Cond: ((flight_no)::bpchar = 'PG0149'::bpchar)
           -> Index Only Scan using boarding_passes_flight_id_seat_no_key on boarding...
               Index Cond: (flight_id = f.flight_id)
(11 строк)
```

Код функции оказался встроенным непосредственно в запрос. При этом условие, заданное в запросе в предложении WHERE, перешло в запрос, находящийся в теле функции. Этому не помешало то, что в запросе столбец имеет имя f_no, а в коде функции — flight_no. Время выполнения сократилось на два порядка, ведь группировка теперь выполняется только для одного маршрута:

```
SELECT f_no, pass_num
FROM count_passengers()
WHERE f_no = 'PG0149';
  f_no  | pass_num
--------+----------
 PG0149 |     1333
(1 строка)
Время: 0,917 мс
```

Похожий эффект можно было бы получить, добавив функции count_passengers параметр для номера маршрута. Но мы достигли даже большей гибкости. Можно передать в запрос несколько маршрутов, скажем Оренбург — Уфа и Уфа — Оренбург:

```
SELECT f_no, pass_num
FROM count_passengers()
WHERE f_no IN ( 'PG0149', 'PG0148' );
  f_no  | pass_num
--------+----------
 PG0148 |     1017
 PG0149 |     1333
(2 строки)
Время: 2,030 мс
```

С тем же успехом мы можем задавать любые другие условия (что рассмотрено в упражнении 25 на с. 417), ведь теперь наша функция стала прозрачной для планировщика: запрос из ее тела оптимизируется вместе с основным запросом. С похожей ситуацией мы уже сталкивались в главе 2 «Общие табличные выражения». Можно сказать, что запрос в невстроенной функции ведет себя подобно материализованному общему табличному выражению, а во встроенной — подобно нематериализованному.

Мы продемонстрировали встраивание в запрос кода *табличной* функции. Чтобы такое встраивание было возможным, необходимо выполнение целого ряда условий, описанных в документе «Inlining of SQL functions» (wiki.postgresql.org/wiki/Inlining_of_SQL_functions). В частности, функция должна быть написана на языке SQL и иметь категорию изменчивости STABLE или IMMUTABLE, а ее тело должно состоять из единственной команды SELECT.

В ходе экспериментов мы видели, что код функции, удовлетворяющей всем условиям, кроме одного — категории изменчивости, — в запрос не встраивается. Влияние последнего из перечисленных условий читатель может проверить самостоятельно, добавив в начало тела функции любой запрос (например, SELECT 1) и повторив эксперименты.

Встраиваться в запрос может код не только табличных, но и *скалярных* функций, то есть возвращающих скалярное значение, которое можно использовать, например, в списке SELECT или в предложении WHERE. Эта возможность рассмотрена в упражнении 22 (с. 408).

5.7.2. Функции и параллельный режим выполнения запросов

В PostgreSQL при выполнении ряда условий запрос может быть выполнен в параллельном режиме. В этом случае в помощь основному (ведущему) процессу создаются один или несколько фоновых рабочих процессов, которые берут часть работы на себя. Этот вопрос подробно освещается в документации в главе 15 «Параллельный запрос».

Выполнение запросов в параллельном режиме мы уже рассматривали в подразделе 3.1.2 «Агрегирование в параллельном режиме» (с. 114), но тогда мы не касались одной из характеристик функций — `PARALLEL`. Она может влиять на принятие планировщиком решения о выборе режима выполнения. Эта характеристика принимает одно из следующих значений:

- `SAFE` — функция может без ограничений выполняться в параллельном режиме;
- `RESTRICTED` — функция может работать только в ведущем процессе, но не препятствует распараллеливанию запроса;
- `UNSAFE` — функция запрещает распараллеливание запроса (это значение выбирается по умолчанию при создании функции).

В подразделе документации 15.4.1 «Пометки параллельности для функций и агрегатов» определены условия, при которых функциям нужно назначать ту или иную характеристику параллельности. Например, функции должны помечаться как небезопасные для параллельного выполнения (`UNSAFE`), если они изменяют состояние базы данных, обращаются к последовательностям или меняют значения конфигурационных параметров сервера, а пометку ограниченной параллельности (`RESTRICTED`) надо ставить, если функция обращается к временным таблицам.

Позволив выполняться в параллельном режиме функции, которая для этого не пригодна, можно получить неверный результат или ошибку, а более строгая, чем необходимо, пометка может привести к неэффективному выполнению запроса.

Предположим, что для анализа финансовых результатов работы нашей авиакомпании нужно рассчитать два показателя:

- количество операций бронирования, полная стоимость которых попадает в различные диапазоны. Ширина таких диапазонов может задаваться по желанию пользователя, например 100 тысяч рублей;
- количество операций бронирования, проведенных за каждую неделю отчетного периода.

Давайте напишем для этого две небольшие функции.

Первая функция вычисляет границы диапазона стоимостей, к которому относится переданное в первом параметре значение. Второй параметр определяет ширину диапазона. Для получения результата воспользуемся функцией div, возвращающей целочисленный результат деления ее первого аргумента на второй (см. раздел документации 9.3 «Математические функции и операторы»).

```
CREATE OR REPLACE FUNCTION get_amount_range(
  amount numeric,
  range_width integer,
  OUT min_amount integer,
  OUT max_amount integer
)
LANGUAGE sql IMMUTABLE PARALLEL SAFE
BEGIN ATOMIC
  SELECT
    range * range_width,
    ( range + 1 ) * range_width
  FROM
  ( SELECT div( amount, range_width ) AS range
  );
END;
```

```
CREATE FUNCTION
```

Вторая функция похожа на первую, но предназначена для дат и работает с фиксированной шириной диапазона, равной семи дням. Она определяет понедельник и воскресенье той недели, на которую выпадает указанная дата.

Обе функции написаны в стиле стандарта SQL. Каждая содержит только один оператор, однако нам пришлось использовать синтаксис `BEGIN ATOMIC ... END` из-за двух возвращаемых значений.

```
CREATE OR REPLACE FUNCTION get_date_range(
  dt date,
  OUT monday date,
  OUT sunday date
)
LANGUAGE sql IMMUTABLE PARALLEL SAFE
BEGIN ATOMIC
  SELECT
    prev_sunday + 1,
    prev_sunday + 7
  FROM
  ( SELECT dt - extract( isodow FROM dt )::integer AS prev_sunday
  );
END;
CREATE FUNCTION
```

Обе функции являются безопасными для выполнения в параллельном режиме, поэтому мы назначили им характеристику PARALLEL SAFE. Проверим это:

```
\df+ get_*_range
Список функций
-[ RECORD 1 ]----------+-----------------
Схема                  | bookings
Имя                    | get_amount_range
 ...
Параллельность         | безопасная
 ...
-[ RECORD 2 ]----------+-----------------
Схема                  | bookings
Имя                    | get_date_range
 ...
Параллельность         | безопасная
 ...
```

За этими сведениями можно обратиться и к системному каталогу pg_proc:

```
SELECT proname, proparallel
FROM pg_proc
WHERE proname LIKE 'get_%_range';
     proname      | proparallel
------------------+-------------
 get_amount_range | s
 get_date_range   | s
(2 строки)
```

Вычислим требуемые показатели в одном запросе. Конечно, более рациональным решением было бы использование конструкции GROUPING SETS, но для иллюстрации параллельного режима работы прибегнем к оператору UNION ALL:

```
SELECT
  amount_range.min_amount / 1000 AS min_amount,
  amount_range.max_amount / 1000 AS max_amount,
  NULL::date AS monday,
  NULL::date AS sunday,
  count( * ) AS book_num
FROM bookings, get_amount_range( total_amount, 100000 ) AS amount_range
GROUP BY 1, 2
UNION ALL
SELECT
  NULL::integer AS min_amount,
  NULL::integer AS max_amount,
  week.monday,
  week.sunday,
  count( * ) AS book_num
FROM bookings, get_date_range( book_date::date ) AS week
GROUP BY 3, 4
ORDER BY 1, 2, 3, 4;
```

```
 min_amount | max_amount |   monday   |   sunday   | book_num
------------+------------+------------+------------+----------
          0 |        100 |            |            |   198314
        100 |        200 |            |            |    46943
        200 |        300 |            |            |    11916
        300 |        400 |            |            |     3260
        400 |        500 |            |            |     1357
        500 |        600 |            |            |      681
        600 |        700 |            |            |      222
        700 |        800 |            |            |       55
        800 |        900 |            |            |       24
        900 |       1000 |            |            |       11
       1000 |       1100 |            |            |        4
       1200 |       1300 |            |            |        1
            |            | 2017-06-19 | 2017-06-25 |      262
            |            | 2017-06-26 | 2017-07-02 |    14074
            |            | 2017-07-03 | 2017-07-09 |    37223
            |            | 2017-07-10 | 2017-07-16 |    39160
            |            | 2017-07-17 | 2017-07-23 |    38523
            |            | 2017-07-24 | 2017-07-30 |    38641
            |            | 2017-07-31 | 2017-08-06 |    39282
            |            | 2017-08-07 | 2017-08-13 |    42397
            |            | 2017-08-14 | 2017-08-20 |    13226
(21 строка)
```

Оказывается, в диапазоне от 1100 тысяч до 1200 тысяч рублей не было ни одной операции бронирования.

А каким будет план запроса?

```
EXPLAIN ( costs off )
...
                                    QUERY PLAN
-----------------------------------------------------------------------------------------
 Sort
   Sort Key: ((amount_range.min_amount / 1000)), ((amount_range.max_amount / 1000)), ...
   -> Gather
       Workers Planned: 2
       -> Parallel Append
           -> HashAggregate
               Group Key: (amount_range.min_amount / 1000), (amount_range.max_amount ...
               -> Nested Loop
                   -> Seq Scan on bookings
                   -> Function Scan on get_amount_range amount_range
           -> HashAggregate
               Group Key: week.monday, week.sunday
               -> Nested Loop
                   -> Seq Scan on bookings bookings_1
                   -> Function Scan on get_date_range week
(15 строк)
```

План показывает, что запрос будет выполняться в параллельном режиме. Это объяснимо, ведь мы для обеих функций задали характеристику PARALLEL SAFE.

Обратите внимание на узел Parallel Append, дочерние узлы которого выполняются параллельно. Этот узел описан в подразделе документации 15.3.4 «Параллельное присоединение».

Предлагаем читателю самостоятельно выполнить запрос с параметром ANALYZE команды EXPLAIN, чтобы увидеть сведения о фактически запущенных рабочих процессах, а также значения показателей rows и loops в распараллеленных узлах планов.

Давайте теперь назначим функции get_amount_range характеристику PARALLEL RESTRICTED и проверим план повторно:

```
ALTER FUNCTION get_amount_range PARALLEL RESTRICTED;
ALTER FUNCTION
```

```
EXPLAIN ( costs off )
...
                                    QUERY PLAN
-----------------------------------------------------------------------------------------
 Sort
   Sort Key: ((amount_range.min_amount / 1000)), ((amount_range.max_amount / 1000)), ...
   -> Append
       -> HashAggregate
           Group Key: (amount_range.min_amount / 1000), (amount_range.max_amount / 1000)
           -> Nested Loop
               -> Seq Scan on bookings
               -> Function Scan on get_amount_range amount_range
       -> Finalize GroupAggregate
           Group Key: week.monday, week.sunday
           -> Gather Merge
               Workers Planned: 1
               -> Sort
                   Sort Key: week.monday, week.sunday
                   -> Partial HashAggregate
                       Group Key: week.monday, week.sunday
                       -> Nested Loop
                           -> Parallel Seq Scan on bookings bookings_1
                           -> Function Scan on get_date_range week
(19 строк)
```

Распараллеливается только поддерево плана, в котором выполняется функция `get_date_range`, сохранившая пометку `PARALLEL SAFE`. Дополнительный рабочий процесс создается в рамках этого поддерева.

Теперь назначим функции `get_amount_range` характеристику параллельности `PARALLEL UNSAFE`:

```
ALTER FUNCTION get_amount_range PARALLEL UNSAFE;
ALTER FUNCTION
```

Поскольку одна из функций теперь считается небезопасной для параллельного выполнения запроса, не распараллеливается выполнение и второй функции. Таким образом, подтверждается положение документации, говорящее, что наличие в запросе даже одной небезопасной функции препятствует распараллеливанию всего запроса.

```
EXPLAIN ( costs off )
...
                                    QUERY PLAN
-----------------------------------------------------------------------------------------
 Sort
   Sort Key: ((amount_range.min_amount / 1000)), ((amount_range.max_amount / 1000)), ...
   -> Append
       -> HashAggregate
           Group Key: (amount_range.min_amount / 1000), (amount_range.max_amount / 1000)
           -> Nested Loop
               -> Seq Scan on bookings
               -> Function Scan on get_amount_range amount_range
       -> HashAggregate
           Group Key: week.monday, week.sunday
           -> Nested Loop
               -> Seq Scan on bookings bookings_1
               -> Function Scan on get_date_range week
(13 строк)
```

Мы провели три эксперимента. Поскольку в нашем распоряжении две функции и три значения характеристики параллельности, можно поставить еще целый ряд экспериментов, например назначив обеим функциям характеристику RESTRICTED.

Предлагаем читателю проделать эти упражнения самостоятельно.

5.8. Элементы теории принятия решений

Аналитические возможности PostgreSQL, представленные в главе 3 «Аналитические возможности PostgreSQL» (с. 109), можно расширить с помощью пользовательских функций на языке SQL. В качестве примера мы рассмотрим реализацию одного из методов теории принятия решений.

Наверное, никто не станет спорить с тем, что для принятия обоснованных решений нужна полная и достоверная информация. Однако не менее важны и адекватные методы. В нашей авиакомпании для серьезных задач применяется теория принятия решений, одним из методов которой является формирование *множества Парето*. Метод носит имя итальянского экономиста и социолога Вильфредо Парето (1848–1923). Критерий Парето таков: «Следует считать,

что любое изменение, которое никому не причиняет убытков и которое приносит некоторым людям пользу, является улучшением». Говоря более строгим языком, система находится в Парето-оптимальном состоянии, когда ни один ее показатель не может быть улучшен без ухудшения какого-либо другого показателя.

Итак, предположим, что нашей авиакомпании потребовался поставщик качественных обедов для пассажиров. В теории принятия решений варианты выбора называют *альтернативами*. Принятие решения в нашем случае заключается в выборе лучшей альтернативы, то есть наиболее подходящего варианта обеда.

Показатели для оценки альтернатив выберем следующие: калорийность, разнообразие (определяемое количеством различных блюд и продуктов, составляющих обед), а также цена:

```
CREATE TABLE meal
( meal_code text PRIMARY KEY,      -- код обеда
  price numeric( 6, 2 ) NOT NULL,  -- цена
  calories smallint NOT NULL,      -- калорийность
  variety smallint NOT NULL        -- число блюд
);
CREATE TABLE
```

В литературе по теории принятия решений показатели зачастую называют критериями, хотя это, строго говоря, не одно и то же. *Критерий* — это правило интерпретации значений показателя, например отнесения конкретного значения к группе допустимых или недопустимых. Критерии разделяются на позитивные и негативные. Для позитивного критерия желательным является увеличение его значения, а для негативного — уменьшение.

Если альтернативы оценивают по нескольким критериям, как в нашем примере, то такая задача принятия решений называется многокритериальной. В многокритериальных задачах проявляется эффект несравнимости альтернатив. Что это такое?

В рассматриваемом методе — формировании множества Парето — важнейшим понятием является доминирование альтернатив. Альтернатива *A* называется *доминирующей* по отношению к альтернативе *B*, если оценки альтернативы *A* по всем критериям не хуже, чем оценки альтернативы *B*, а хотя бы по одному критерию — строго лучше. При этом альтернатива *B* называется *доминируемой*.

Если же альтернатива *A* превосходит альтернативу *B* по одним критериям, но уступает ей по другим, тогда эти альтернативы *несравнимы*. Точнее говоря, их можно сравнить, но для этого нужно привлечь дополнительные критерии.

Если из исходного множества альтернатив отобрать недоминируемые, то это подмножество и будет множеством Парето для исходного множества. Любая альтернатива, входящая в множество Парето, будет предпочтительнее любой из оставшихся альтернатив исходного множества. Однако между собой альтернативы из множества Парето несравнимы.

Таким образом, альтернатива называется Парето-оптимальной, если для нее не существует доминирующей альтернативы. Для Парето-оптимальной альтернативы нельзя найти другую альтернативу, которая превосходила бы ее хотя бы по одному критерию и при этом ни по одному не уступала бы.

Во многих методах принятия решений формирование множества Парето выполняется в качестве первого этапа. Если требуется выбрать одну лучшую (по многим критериям) альтернативу, то она обязательно принадлежит этому множеству. Цель такого этапа — сузить исходное множество альтернатив, с тем чтобы потом с помощью привлечения дополнительной информации выбрать из него одну — лучшую — альтернативу.

Один из способов формирования множества Парето — попарное сравнение альтернатив и *исключение* доминируемых. Именно его мы и покажем.

Если из двух сравниваемых альтернатив *A* и *B* альтернатива *A* оказалась лучше, чем *B*, то альтернатива *B* однозначно не должна быть включена в множество Парето. Однако насчет альтернативы *A*, победившей в этом сравнении, пока еще ничего сказать нельзя: ведь при ее сравнении с другими альтернативами может найтись такая, которая окажется лучше *A* (тогда *A* не войдет в множество Парето), но может и не найтись (тогда *A* войдет в множество Парето).

Когда множество Парето определено, для отыскания единственной оптимальной альтернативы среди его элементов можно пойти одним из двух путей:

- предоставить возможность выбора лицу, принимающему решения, на основе его неформализованных предпочтений;
- уменьшить множество Парето (в идеале — до одного элемента) с помощью некоторых формализованных процедур, но для этого требуется дополнительная информация о критериях или о свойствах оптимального решения.

Теперь вернемся к нашему примеру. Поместим исходные данные в таблицу «Обеды» (meal):

```
INSERT INTO meal ( meal_code, price, calories, variety )
VALUES
  ( 'A', 550.00, 1500, 3 ),
  ( 'B', 490.00, 1300, 4 ),
  ( 'C', 600.00, 1400, 4 ),
  ( 'D', 580.00, 1600, 5 ),
  ( 'E', 570.00, 1380, 5 ),
  ( 'F', 520.00, 1450, 3 ),
  ( 'G', 580.00, 1580, 4 ),
  ( 'H', 570.00, 1380, 4 ),
  ( 'I', 510.00, 1450, 3 ),
  ( 'J', 530.00, 1450, 6 );
INSERT 0 10
```

Сравним альтернативы *E* и *H*. В нашем примере цена обеда является, конечно, негативным критерием, а калорийность и разнообразие обеда — критерии позитивные. Значения первых двух показателей совпадают, а по третьему альтернатива *E* превосходит альтернативу *H*. Значит, альтернатива *H* не войдет в множество Парето, а насчет судьбы альтернативы *E* на основании одной победы над альтернативой *H* ничего определенного сказать нельзя.

Сравнив альтернативу *E* с альтернативой *J*, видим, что первая уступает второй по всем критериям. Таким образом, альтернатива *E* не войдет в множество Парето, хотя она и превзошла альтернативу *H*.

А вот альтернативы *J* и *A* оказываются несравнимыми по Парето, потому что первая превосходит вторую по первому и третьему критериям, но уступает ей по второму критерию. Выражаясь спортивным языком, счет в состязании этих двух альтернатив не «сухой» — 2:1. А для признания одной из двух альтернатив победительницей счет должен быть непременно «сухим», пусть даже и минимальным — 1:0. Совпадающие оценки, то есть «ничьи», не учитываются.

Теперь, когда механизм сопоставления альтернатив понятен, можно представить код функции, сравнивающей две альтернативы. Будем подсчитывать, сколько раз показатели первой из них оказались лучше, чем у второй, а сколько раз — хуже. Число совпадающих значений показателей в расчет не принимается. Подсчитанное количество побед и поражений первой альтернативы по

отдельным критериям запишем в массив, состоящий из двух элементов. Поскольку важен факт наличия или отсутствия таких побед и поражений, приходится использовать именно массив, а не скалярную величину, представляющую собой разницу в счете.

```
CREATE OR REPLACE FUNCTION compare_pairwise( a1 meal, a2 meal )
RETURNS smallint[] AS
$$
  SELECT ARRAY[
      -- подсчет числа побед первой альтернативы
      ( CASE WHEN a1.price < a2.price THEN 1 ELSE 0 END ) +
      ( CASE WHEN a1.calories > a2.calories THEN 1 ELSE 0 END ) +
      ( CASE WHEN a1.variety > a2.variety THEN 1 ELSE 0 END ),
      -- подсчет числа поражений первой альтернативы
      ( CASE WHEN a1.price > a2.price THEN 1 ELSE 0 END ) +
      ( CASE WHEN a1.calories < a2.calories THEN 1 ELSE 0 END ) +
      ( CASE WHEN a1.variety < a2.variety THEN 1 ELSE 0 END )
    ]::smallint[] AS score;
$$ LANGUAGE sql;
CREATE FUNCTION
```

Параметрами функции будут значения *составного* типа meal. До сих пор в разрабатываемых функциях мы не использовали такие параметры, настало время восполнить этот пробел. Напомним, что при создании таблицы создается составной тип, имя которого совпадает с ее именем. Внутри функции к элементам такого значения можно обращаться так же, как к столбцам таблицы (например, a1.calories).

При сравнении пары альтернатив не важно, *сколько раз* каждая из них превзошла другую по отдельным критериям. Важно лишь зафиксировать, превзошла ли первая альтернатива вторую хотя бы по одному критерию, а вторая — первую также хотя бы по одному критерию. Поэтому в принципе вместо массива, состоящего из двух элементов, результат сравнения альтернатив можно представить целым числом с помощью битовых строк (см. раздел документации 8.10 «Битовые строки») и побитового «или».

При такой реализации значение 0 будет означать, что альтернативы совпадают, 1 — что лучше первая альтернатива, 2 — что лучше вторая и 3 — что альтернативы несравнимы.

```
CREATE OR REPLACE FUNCTION compare_pairwise_2( a1 meal, a2 meal )
RETURNS smallint AS
$$
  SELECT (
    CASE WHEN a1.price < a2.price THEN b'01' -- первая альтернатива лучше
         WHEN a2.price < a1.price THEN b'10' -- вторая альтернатива лучше
         ELSE b'00'                          -- значения одинаковые
    END |
    CASE WHEN a1.calories > a2.calories THEN b'01'
         WHEN a2.calories > a1.calories THEN b'10'
         ELSE b'00'
    END |
    CASE WHEN a1.variety > a2.variety THEN b'01'
         WHEN a2.variety > a1.variety THEN b'10'
         ELSE b'00'
    END )::integer AS score;
$$ LANGUAGE sql;
CREATE FUNCTION
```

Прежде чем перейти к разработке функции формирования множества Парето на основе результатов попарных сравнений альтернатив, проверим работу функции compare_pairwise. В качестве примера используем альтернативу *D*:

```
SELECT
  m1.meal_code AS alt1, m2.meal_code AS alt2,
  m1.price AS price1, m2.price AS price2,
  m1.calories AS calories1, m2.calories AS calories2,
  m1.variety AS variety1, m2.variety AS variety2,
  compare_pairwise( m1, m2 ) AS score
FROM meal AS m1, meal AS m2
WHERE m1.meal_code = 'D'
ORDER BY m2.meal_code;
 alt1 | alt2 | price1 | price2 | calories1 | calories2 | variety1 | variety2 | score
------+------+--------+--------+-----------+-----------+----------+----------+-------
 D    | A    | 580.00 | 550.00 |      1600 |      1500 |        5 |        3 | {2,1}
 D    | B    | 580.00 | 490.00 |      1600 |      1300 |        5 |        4 | {2,1}
 D    | C    | 580.00 | 600.00 |      1600 |      1400 |        5 |        4 | {3,0}
 D    | D    | 580.00 | 580.00 |      1600 |      1600 |        5 |        5 | {0,0}
 D    | E    | 580.00 | 570.00 |      1600 |      1380 |        5 |        5 | {1,1}
 D    | F    | 580.00 | 520.00 |      1600 |      1450 |        5 |        3 | {2,1}
 D    | G    | 580.00 | 580.00 |      1600 |      1580 |        5 |        4 | {2,0}
 D    | H    | 580.00 | 570.00 |      1600 |      1380 |        5 |        4 | {2,1}
 D    | I    | 580.00 | 510.00 |      1600 |      1450 |        5 |        3 | {2,1}
 D    | J    | 580.00 | 530.00 |      1600 |      1450 |        5 |        6 | {1,2}
(10 строк)
```

На основании результатов выполненного запроса можно сказать, что альтернативы *C* и *G* не войдут в множество Парето.

Переходя к функции формирования множества Парето, примем такой алгоритм ее работы: сначала на основе результатов попарных сравнений сформируем множество доминируемых альтернатив, а затем отберем из исходного множества только те альтернативы, которых нет в списке доминируемых. Эти альтернативы могут быть и не сравнимы по данным критериям. Возможно также совпадение значений их показателей. В двух последних случаях конструкция CASE возвратит NULL.

```
CREATE OR REPLACE FUNCTION pareto()
RETURNS SETOF meal AS
$$
WITH dominated_alternatives AS
( SELECT DISTINCT
    CASE
      -- первая альтернатива побеждала, а вторая нет
      WHEN score[ 1 ] > 0 AND score[ 2 ] = 0 THEN m2
      -- вторая альтернатива побеждала, а первая нет
      WHEN score[ 1 ] = 0 AND score[ 2 ] > 0 THEN m1
      -- Альтернативы несравнимы или равны
      ELSE NULL
    END AS dominated
  FROM meal AS m1
    JOIN meal AS m2 ON m1.meal_code < m2.meal_code
    CROSS JOIN compare_pairwise( m1.*, m2.* ) AS cp( score )
)
SELECT meal FROM meal
EXCEPT
SELECT dominated FROM dominated_alternatives;
$$ LANGUAGE sql;
CREATE FUNCTION
```

Обратите внимание на условие соединения строк в предложении ON. Здесь используется операция «меньше», поскольку порядок сравниваемых альтернатив в паре не имеет значения. Сравнение альтернативы *A* с альтернативой *B* даст ту же информацию, что и сравнение *B* с *A*, только лишь в массиве, содержащем числовые результаты сравнения, поменяются местами значения первого и второго элементов. Таким образом, использование декартова произведения или условия «не равно» привело бы к дублированию работы.

Функции compare_pairwise передаются значения m1.* и m2.*, а не m1 и m2, как в предыдущем запросе. Различия между двумя вариантами будут рассмотрены в упражнении 28 (с. 419).

Для наглядности покажем результат попарного сравнения всех альтернатив. Воспользуемся запросом из функции, немного модифицировав его. Поскольку функция compare_pairwise возвращает всего один столбец, можно написать AS score вместо, например, AS cp(score):

```
SELECT
  m1.meal_code AS alt1,
  m2.meal_code AS alt2,
  score
FROM meal AS m1
  JOIN meal AS m2 ON m1.meal_code < m2.meal_code
  CROSS JOIN compare_pairwise( m1.*, m2.* ) AS score
ORDER BY alt1, alt2;
```

```
 alt1 | alt2 | score
------+------+-------
 A    | B    | {1,2}
 A    | C    | {2,1}
 A    | D    | {1,2}
 A    | E    | {2,1}
 ...
 C    | D    | {0,3}
 C    | E    | {1,2}
 C    | F    | {1,2}
 C    | G    | {0,2}
 ...
 G    | H    | {1,1}
 G    | I    | {2,1}
 G    | J    | {1,2}
 H    | I    | {1,2}
 H    | J    | {0,3}
 I    | J    | {1,1}
(45 строк)
```

Поскольку мы проводим оценку с позиции альтернативы, указанной в столбце alt1, то у доминируемых альтернатив первый элемент массива, содержащего результаты сравнения, будет равен нулю, а второй — больше нуля. Конечно, мы могли бы провести оценку и с позиции альтернативы, указанной в столбце alt2. Для этого потребовалось бы только поменять местами аргументы в вызове функции compare_pairwise.

Теперь мы можем провести первый этап конкурса обедов для пассажиров:

```
SELECT *
FROM pareto()
ORDER BY meal_code;
```

```
 meal_code | price  | calories | variety
-----------+--------+----------+---------
 A         | 550.00 |     1500 |       3
 B         | 490.00 |     1300 |       4
 D         | 580.00 |     1600 |       5
 I         | 510.00 |     1450 |       3
 J         | 530.00 |     1450 |       6
(5 строк)
```

Наша первоначальная группа претендентов значительно сократилась, однако единственный победитель выявлен не был. Это типичный исход при использовании отбора альтернатив на основе принципа доминирования по Парето. Для выбора единственной альтернативы необходима дополнительная информация. Этот выбор можно сделать на основе различных методов, которые рассмотрены в упражнении 26 (с. 418).

Предположим, что уже после проведения первоначального отбора обедов для наших пассажиров к нам обратился еще один потенциальный поставщик (обозначим буквой *K* предлагаемый им обед). Конечно, отбор кандидатов на конкурс уже завершен, но для того чтобы понять, стоит ли нам сожалеть об этом, мы можем сравнить обед *K* с исходной группой обедов с помощью функции compare_pairwise. При этом значения показателей обеда *K* нужно передать в функцию в качестве параметра составного типа.

Решить эту задачу можно с помощью конструкции ROW(), позволяющей сформировать значение такого типа из элементарных значений. Обратите внимание на операцию приведения типа.

```
SELECT
  'K' AS alt1,
  meal.meal_code AS alt2,
  compare_pairwise(
    ROW( 'K', 540.00, 1580, 4 )::meal,
    meal.*
  ) AS score
FROM meal
ORDER BY alt2;
```

```
 alt1 | alt2 | score
------+------+-------
 K    | A    | {3,0}
 K    | B    | {1,1}
 K    | C    | {2,0}
 K    | D    | {1,2}
 K    | E    | {2,1}
 K    | F    | {2,1}
 K    | G    | {1,0}
 K    | H    | {2,0}
 K    | I    | {2,1}
 K    | J    | {1,2}
(10 строк)
```

Полученный результат говорит о том, что компания, опоздавшая к началу конкурса, в принципе могла бы рассчитывать на получение контракта, поскольку для предлагаемого ею обеда не нашлось ни одной доминирующей альтернативы. При этом обед *K* превосходит одного из претендентов, успешно преодолевших первый этап отбора, а именно обед *A*. Поскольку в нашем примере число альтернатив невелико, то и без выполнения дополнительного запроса видно, что обед *A* не вошел бы в число победителей первого этапа, если бы обед *K* участвовал в конкурсе. В случае большого числа участников мы могли бы воспользоваться таким запросом:

```
SELECT
  'K' AS alt1,
  p.meal_code AS alt2,
  p.price AS price2,
  p.calories AS calories2,
  p.variety AS variety2,
  score
FROM
  ( SELECT * FROM pareto()
  ) AS p
  CROSS JOIN compare_pairwise( ROW( 'K', 540.00, 1580, 4 )::meal, p.* ) AS score
WHERE score[ 1 ] > 0
  AND score[ 2 ] = 0
ORDER BY p.meal_code;
```

Условия в предложении WHERE написаны исходя из того, что первой в паре сравниваемых альтернатив идет альтернатива *K*. Полученный результат показывает, что альтернатива *A* не вошла бы в множество Парето:

```
 alt1 | alt2 | price2 | calories2 | variety2 | score
------+------+--------+-----------+----------+-------
 K    | A    | 550.00 |      1500 |        3 | {3,0}
(1 строка)
```

Заметим, что представленные здесь функции compare_pairwise и pareto не являются универсальными: имена таблиц и столбцов, участвующих в выработке решения, заданы в их исходном коде. Поэтому в случае решения другой аналогичной задачи код придется корректировать. PostgreSQL позволяет написать эти функции таким образом, чтобы им можно было передавать имена таблиц и столбцов в качестве аргументов. Однако рассмотрение этого вопроса выходит за рамки настоящего учебника.

5.9. Процедуры

В PostgreSQL реализованы процедуры, представленные в разделе документации 36.4 «Пользовательские процедуры». Это объекты, подобные функциям, поэтому значительная часть этой главы, посвященная функциям, остается актуальной и для процедур. Однако между ними есть целый ряд различий.

1. В процедурах можно использовать команды управления транзакциями, а внутри функций это невозможно. Правда, в процедурах, написанных на языке SQL, это тоже невозможно (годится, например, PL/pgSQL), поэтому рассмотрение данного вопроса остается за рамками этого учебника.

2. Функция вызывается как часть команды манипулирования данными (например, SELECT или UPDATE), а для вызова процедуры служит отдельная команда CALL, описание которой приведено в документации.

3. Поскольку процедура выполняется отдельной командой, возврат значения из нее с помощью оператора RETURN не предусмотрен: в отличие от функции, это значение было бы невозможно использовать. Поэтому в определении процедуры нет и не может быть предложения RETURNS. Процедура, как и функция, может возвращать (или «выдавать») значения с помощью параметров, имеющих модификаторы OUT или INOUT. Однако таким способом может быть возвращена только одна строка, поскольку предложения RETURNS SETOF и RETURNS TABLE в процедурах не используются.

4. Процедуры, опять же вследствие особенностей их вызова, не имеют целого ряда характеристик, присущих функциям: категории изменчивости, способа реагирования на аргументы со значением NULL, безопасности выполнения в параллельных запросах. Полное представление о различиях можно получить из описаний команд CREATE FUNCTION и CREATE PROCEDURE, приведенных в документации.

Процедуры и функции используют общее пространство имен, поэтому из двух перегруженных объектов один может быть функцией, а другой — процедурой.

В простых случаях функцию, возвращающую void, можно заменить процедурой. К тому же процедуры являются частью стандарта языка SQL, а возврат void из функции — это расширение PostgreSQL.

От теоретического вступления перейдем к практическим задачам. В разделе 5.8 «Элементы теории принятия решений» (с. 357) была рассмотрена функция pareto, формирующая множество Парето. Давайте создадим процедуру, аналогичную этой функции, но предусмотрим в ней сохранение результата в таблицу базы данных.

Нам потребуется добавить в таблицу «Обеды» (meal) столбец для записи признака принадлежности обеда множеству Парето:

```
ALTER TABLE meal
ADD COLUMN pareto_optimal bool;
ALTER TABLE
```

Функция compare_pairwise будет корректно работать с модифицированным составным типом данных meal.

В процедуре первая команда UPDATE записывает значение t в поле pareto_optimal всех строк. Это необходимо, поскольку формирование множества Парето происходит путем исключения альтернатив из исходного множества, а изначально все альтернативы предполагаются недоминируемыми.

Вторая команда UPDATE записывает значение f в поле pareto_optimal только тех строк, которые соответствуют доминируемым альтернативам. Обновляемые строки отбираются здесь в предложении FROM команды UPDATE. Можно было бы использовать и предикат EXISTS в предложении WHERE.

```
CREATE OR REPLACE PROCEDURE pareto_proc() AS
$$
  UPDATE meal SET pareto_optimal = 't';
  WITH dominated_alternatives AS
  ( SELECT DISTINCT
      CASE
        -- первая альтернатива побеждала, а вторая нет
        WHEN score[ 1 ] > 0 AND score[ 2 ] = 0 THEN m2
        -- вторая альтернатива побеждала, а первая нет
        WHEN score[ 1 ] = 0 AND score[ 2 ] > 0 THEN m1
        -- альтернативы несравнимы или равны
        ELSE NULL
      END AS dominated
    FROM meal AS m1
      JOIN meal AS m2 ON m1.meal_code < m2.meal_code
      CROSS JOIN compare_pairwise( m1.*, m2.* ) AS cp( score )
  )
  UPDATE meal SET pareto_optimal = 'f'
  FROM dominated_alternatives AS da
  WHERE meal.meal_code = (da.dominated).meal_code;
$$
LANGUAGE sql;
CREATE PROCEDURE
```

Обратите внимание на запись (da.dominated).meal_code: элемент dominated представляет значение составного типа meal.

Посмотреть описание процедуры, как и описание функции, можно с помощью команды \df утилиты psql:

```
\df pareto_proc
Список функций
-[ RECORD 1 ]----------+------------
Схема                  | bookings
Имя                    | pareto_proc
Тип данных результата  |
Типы данных аргументов |
Тип                    | проц.
```

Выполним процедуру:

```
CALL pareto_proc();
CALL
```

Поскольку процедура не возвращает никакого значения и никакие сообщения на консоль не выводятся, для проверки результата ее выполнения придется выполнить отдельный запрос:

```
SELECT meal_code, price, calories, variety, pareto_optimal
FROM meal
WHERE pareto_optimal
ORDER BY meal_code;
```

Результат получился тот же, что и при использовании функции pareto:

```
 meal_code | price  | calories | variety | pareto_optimal
-----------+--------+----------+---------+----------------
 A         | 550.00 |     1500 |       3 | t
 B         | 490.00 |     1300 |       4 | t
 D         | 580.00 |     1600 |       5 | t
 I         | 510.00 |     1450 |       3 | t
 J         | 530.00 |     1450 |       6 | t
(5 строк)
```

Так как процедуры могут возвращать значения при помощи выходных параметров, давайте попробуем сразу показать полученный результат. Конечно, придется использовать агрегатную функцию, поскольку, как было сказано выше, вывести несколько строк невозможно. Всю работу по формированию множества Парето выполнит процедура pareto_proc, а в процедуру pareto_proc_2 добавим только выборку из таблицы «Обеды» (meal) с группировкой:

```
CREATE OR REPLACE PROCEDURE pareto_proc_2( OUT pareto_optimals text ) AS
$$
  CALL pareto_proc();
  SELECT string_agg( meal_code, ', ' ORDER BY meal_code )
  FROM meal
  WHERE pareto_optimal;
$$ LANGUAGE sql;
CREATE PROCEDURE
```

Теперь мы можем получить список кодов тех обедов, которые вошли в множество Парето. Обратите внимание, что, в отличие от вызовов функций, для выходного параметра процедуры передается значение. В данном случае мы использовали NULL, но это может быть любое значение типа text (или других строковых типов).

```
CALL pareto_proc_2( NULL );
 pareto_optimals
-----------------
 A, B, D, I, J
(1 строка)
```

Как и в случае с функциями, имена столбцов полученной таблицы определяются именами выходных параметров, а не псевдонимами столбцов выборки. Поэтому они не обязаны совпадать.

Отметим, что в настоящее время (версия 17 PostgreSQL) в случае формирования тела процедуры в стиле стандарта языка SQL вызов другой процедуры не поддерживается:

```
CREATE OR REPLACE PROCEDURE pareto_proc_2( OUT pareto_optimals text )
LANGUAGE sql
BEGIN ATOMIC
  CALL pareto_proc();
  SELECT string_agg( meal_code, ', ' ORDER BY meal_code )
  FROM meal
  WHERE pareto_optimal;
END;
ОШИБКА:  CALL на данный момент не поддерживается в теле SQL-функции, задаваемом не в
кавычках
```

Для удаления процедуры служит команда `DROP PROCEDURE`. Если имя удаляемой процедуры уникально, то ее параметры можно не указывать. Но если это перегруженная процедура (функция), тогда PostgreSQL потребует указать и параметры, имеющие модификаторы `IN` или `INOUT`.

```
DROP PROCEDURE pareto_proc;
DROP PROCEDURE
```

Удаление прошло успешно, хотя данная процедура используется в процедуре `pareto_proc_2`. Теперь при попытке вызова процедуры `pareto_proc_2` будет сгенерирована ошибка:

```
CALL pareto_proc_2( NULL );
ОШИБКА:  процедура pareto_proc() не существует
СТРОКА 2:   CALL pareto_proc();
 ...
```

5.10. Контрольные вопросы и задания

1 Совместное использование параметров с модификатором OUT и предложения RETURNS

Параметры с модификатором OUT позволяют функции возвратить более одного значения. При использовании таких параметров можно в ряде случаев обойтись без предложения RETURNS в определении функции.

Вопрос. Когда бывает целесообразно использовать параметры с модификатором OUT вместе с предложением RETURNS? Что это дает?

Указание. Полезные сведения можно найти в подразделе документации 36.5.9 «Функции SQL, возвращающие множества» и в тексте главы.

2 Значение параметра функции в качестве идентификатора в ее коде — это возможно?

Как вы думаете, можно ли значение параметра, переданного в SQL-функцию, использовать в качестве идентификатора в коде этой функции, например в качестве имени столбца? Почему?

Задание. Проведите эксперименты для проверки вашей гипотезы, взяв в качестве тестового примера одну из функций, приведенных в тексте этой главы, например count_seats.

Указание. За справкой можно обратиться к подразделу документации 36.5.1 «Аргументы SQL-функций».

3 Если имена параметров функции совпадают с именами столбцов таблицы

Как вы думаете, что произойдет, если имена параметров функции совпадут с именами столбцов таблицы, используемой в коде этой функции? Будет ли такая функция успешно создана или команда CREATE FUNCTION выдаст ошибку? Будет ли созданная функция работать корректно? Если нет, то что для этого нужно сделать?

Задание. Проведите эксперименты для проверки вашей гипотезы, взяв в качестве тестового примера одну из функций, приведенных в тексте этой главы, например `count_seats`.

Указание. За справкой можно обратиться к подразделу документации 36.5.1 «Аргументы SQL-функций».

4 Символ одинарной кавычки в теле функции

При создании функций можно использовать символ одинарной кавычки для ограничения тела функции, которое представляет собой символьную строку. Но при этом возможно несколько вариантов обработки одинарных кавычек внутри тела функции. Предположим, что функция, назовем ее `test`, должна выводить строку в таком виде:

```
   test
-----------
 PGConf'25
(1 строка)
```

Давайте рассмотрим первый вариант решения задачи:

```
CREATE OR REPLACE FUNCTION test()
RETURNS text AS
'SELECT ''PGConf''''25'';'
LANGUAGE sql;
CREATE FUNCTION
```

Как вы думаете, почему в этом случае требуется четыре кавычки?

Вот еще один вариант, в котором используется так называемая спецпоследовательность (escape-string). Эта возможность PostgreSQL является расширением стандарта SQL:

```
CREATE OR REPLACE FUNCTION test()
RETURNS text AS
'SELECT E''PGConf\''25'';'
LANGUAGE sql;
CREATE FUNCTION
```

А вот еще более экзотический вариант решения задачи:

```
CREATE OR REPLACE FUNCTION test()
RETURNS text AS
E'SELECT \'PGConf\'\'25\';'
LANGUAGE sql;
```

```
CREATE FUNCTION
```

Задание. Объясните каждый вариант решения задачи.

Указание. Обратитесь к подразделу документации 4.1.2 «Константы». Посмотрите, в каком виде код функции сохраняется в базе данных, с помощью команды `\sf` утилиты psql.

5 Перегруженные функции и значения параметров по умолчанию

Функции в СУБД PostgreSQL можно *перегружать*, то есть создавать одноименные функции с различающимися типами входных параметров (имеющих модификаторы `IN` или `INOUT` либо объявленных без модификатора). В таком случае сигнатуры функций будут различными. При вызове PostgreSQL определяет конкретную перегруженную функцию на основании переданных ей аргументов. Однако с этим могут возникать сложности, если параметры имеют значения по умолчанию.

```
CREATE OR REPLACE FUNCTION test_default(
  par1 int,
  par2 int DEFAULT 2
) RETURNS text AS
$$
  SELECT format( 'test_default1; par1 = %s, par2 = %s', par1, par2);
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

```
CREATE OR REPLACE FUNCTION test_default(
  par1 int,
  par2 int DEFAULT 2,
  par3 int DEFAULT 3
) RETURNS text AS
$$
  SELECT format( 'test_default2; par1 = %s, par2 = %s, par3 = %s', par1, par2, par3);
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

Мы создали две перегруженные функции в текущей схеме. Попробуем вызвать первую из них, задавая сначала один, а затем два аргумента:

```
SELECT test_default( 10 );
ОШИБКА:  функция test_default(integer) не уникальна
СТРОКА 1: SELECT test_default( 10 );
                  ^
ПОДСКАЗКА:  Не удалось выбрать лучшую кандидатуру функции. Возможно, вам следует
добавить явные приведения типов.

SELECT test_default( 10, 20 );
ОШИБКА:  функция test_default(integer, integer) не уникальна
СТРОКА 1: SELECT test_default( 10, 20 );
                  ^
ПОДСКАЗКА:  Не удалось выбрать лучшую кандидатуру функции. Возможно, вам следует
добавить явные приведения типов.
```

Лишь вызов функции с указанием трех аргументов будет успешным:

```
SELECT test_default( 10, 20, 30 );
                   test_default
-----------------------------------------------
 test_default2; par1 = 10, par2 = 20, par3 = 30
(1 строка)
```

Можно вызвать вторую версию функции и с применением именованных аргументов. Причем их можно переставить местами и опустить параметр par2, поскольку он имеет значение по умолчанию:

```
SELECT test_default( par3 => 30, par1 => 10 );
                  test_default
-----------------------------------------------
 test_default2; par1 = 10, par2 = 2, par3 = 30
(1 строка)
```

Таким образом, вызвать первую версию функции, которая имеет два параметра, невозможно. Это объясняется тем, что при задании одного или двух аргументов под этот вызов подходят обе наши функции, поэтому PostgreSQL не может выбрать конкретную функцию.

Задание. Удалите обе функции, созданные в текущей схеме:

```
DROP FUNCTION test_default( int, int );
DROP FUNCTION
DROP FUNCTION test_default;
DROP FUNCTION
```

Затем создайте их в разных схемах базы данных. Выяснить, какие схемы есть в вашей базе данных, можно с помощью команды `\dn`.

Например, одну из функций создайте в схеме `bookings`, а другую — в схеме `public`. Для этого укажите имя схемы в команде создания функции, например:

```
CREATE OR REPLACE FUNCTION public.test_default(
  par1 int,
  par2 int DEFAULT 2
)
...
CREATE OR REPLACE FUNCTION bookings.test_default(
  par1 int,
  par2 int DEFAULT 2,
  par3 int DEFAULT 3
)
...
```

Проверить, в каких схемах оказались созданы ваши функции, можно командой `\df test_default`.

Выполните вызов функции с одним, двумя и тремя аргументами. При вызове функции с одним или двумя аргументами будет вызвана та из них, которая будет найдена *первой* согласно пути поиска, заданного параметром *search_path*:

```
SHOW search_path;
   search_path
-----------------
 bookings, public
(1 строка)
```

Измените путь поиска:

```
SET search_path = public, bookings;
SET
```

Снова повторите запросы с вызовами функции с одним, двумя и тремя аргументами. Что получается теперь?

Конечно, если перегруженные функции находятся в разных схемах, вы всегда можете указать имя схемы при вызове функции, например:

```
SELECT bookings.test_default( 10 );
                test_default
--------------------------------------------
 test_default2; par1 = 10, par2 = 2, par3 = 3
(1 строка)
```

После завершения работы над упражнением не забудьте вернуть значение параметра *search_path* в исходное состояние:

```
RESET search_path;
RESET
```

6 Аргументом функции может быть и значение NULL

При выполнении запросов функция может получить значение NULL в качестве аргумента. Такие ситуации должны корректно обрабатываться и не приводить к сбоям. Данный вопрос рассмотрен в подразделе 5.1.6 «Значения NULL в качестве аргументов функции» (с. 285).

Задание. Посмотрите функции, разработанные в этой главе, и оцените необходимость использования предложения RETURNS NULL ON NULL INPUT (или равнозначного предложения STRICT). Проведите необходимые эксперименты.

7 Взаимосвязи объектов в базе данных

Как вы думаете, можно ли удалить таблицу, которая используется в функции на языке SQL? А столбец такой таблицы? Можно ли создать функцию, в которой используется еще не созданная таблица или столбец, которого нет в существующей таблице? Как повлияет на результаты эксперимента способ оформления тела функции (в виде символьной строки или в стиле стандарта SQL)?

Сначала выскажите обоснованные гипотезы, а потом проверьте на практике.

Указание. Можно воспользоваться примерно такими средствами (порядок выполнения команд может быть другим):

```
CREATE TABLE test_depend_t ( a int, b int );
INSERT INTO test_depend_t VALUES ( 1, 2 );
CREATE OR REPLACE FUNCTION test_depend()
RETURNS int AS
$$
  SELECT count( * ) FROM test_depend_t;
$$ LANGUAGE sql;
ALTER TABLE test_depend_t DROP COLUMN b;
ALTER TABLE test_depend_t ADD COLUMN c int;
CREATE OR REPLACE FUNCTION test_depend_2()
RETURNS int AS
$$
  SELECT b FROM test_depend_t LIMIT 1;
$$ LANGUAGE sql;
```

Для проведения экспериментов самостоятельно перепишите код функций в стиле стандарта SQL.

8 Могут ли параметры с модификаторами INOUT и OUT идти вперемежку?

В разделе 5.3 «Функции, возвращающие множества строк» (с. 298) была представлена функция `list_routes_3`:

```
CREATE OR REPLACE FUNCTION list_routes_3(
  d_city text DEFAULT 'Москва',
  a_city text DEFAULT 'Санкт-Петербург',
  OUT f_no char,
  OUT dep_city text,
  OUT arr_city text,
  OUT model text
)
RETURNS SETOF record AS
$$
  SELECT r.flight_no, r.departure_city, r.arrival_city, a.model
  FROM routes AS r
    JOIN aircrafts AS a ON a.aircraft_code = r.aircraft_code
  WHERE r.departure_city = d_city
    AND r.arrival_city = a_city;
$$ LANGUAGE sql;
```

Эта функция выбирает все маршруты, проложенные из одного города в другой.

Что, если отказаться от двух входных параметров d_city и a_city, заменив модификатор OUT на INOUT у параметров dep_city и arr_city? Так мы сократим число параметров функции.

Предварительно удалите старую версию функции, если она присутствует в базе данных. В противном случае создать ее новую версию не получится, поскольку изменять имена входных параметров нельзя.

```
DROP FUNCTION IF EXISTS list_routes_3;
```
```
DROP FUNCTION
```
```
CREATE OR REPLACE FUNCTION list_routes_3(
  OUT f_no char,
  INOUT dep_city text DEFAULT 'Москва',
  INOUT arr_city text DEFAULT 'Санкт-Петербург',
  OUT model text
)
RETURNS SETOF record AS
$$
  SELECT r.flight_no, r.departure_city, r.arrival_city, a.model
  FROM routes AS r
    JOIN aircrafts AS a ON a.aircraft_code = r.aircraft_code
  WHERE r.departure_city = dep_city
    AND r.arrival_city = arr_city;
$$ LANGUAGE sql;
```
```
CREATE FUNCTION
```

Функция успешно создана.

Задание. Выполните ряд запросов с новой версией функции. Попробуйте использовать как позиционные, так и именованные аргументы (задавая, например, только название города прибытия). Отличается ли порядок вызова этой версии функции от исходной?

9 Параметр VARIADIC идет последним. Почему?

В разделе документации 36.5.6 «Функции SQL с переменным числом аргументов» сказано, что в объявлении функции параметр, помеченный как VARIADIC, должен быть последним. Как вы думаете, почему?

10 Псевдонимы таблиц и столбцов: есть некоторая свобода

Функциям в предложении FROM могут назначаться псевдонимы. В таких случаях, как правило, задают псевдонимы и столбцам. В качестве иллюстрации воспользуемся запросом для формирования номеров кресел в салонах самолетов.

В первом варианте используется и псевдоним таблицы, и псевдоним столбца:

```
SELECT
  row,
  letter,
  row || letter AS seat_no
FROM
  generate_series( 1, 3 ) AS rows( row ),
  unnest( ARRAY[ 'A', 'C', 'D', 'F' ] ) AS letters( letter )
ORDER BY row, letter;
```

Во втором варианте также используется и псевдоним таблицы, и псевдоним столбца, но в списке SELECT вместо псевдонимов столбцов мы видим псевдонимы таблиц:

```
SELECT
  rows,
  letters,
  rows || letters AS seat_no
FROM
  generate_series( 1, 3 ) AS rows( row ),
  unnest( ARRAY[ 'A', 'C', 'D', 'F' ] ) AS letters( letter )
ORDER BY row, letter;
```

И третий вариант. В нем используются только псевдонимы таблиц, они же фигурируют и в списке SELECT:

```
SELECT
  row,
  letter,
  row || letter AS seat_no
FROM
  generate_series( 1, 3 ) AS row,
  unnest( ARRAY[ 'A', 'C', 'D', 'F' ] ) AS letter
ORDER BY row, letter;
```

Результаты получим одинаковые (хотя имена столбцов могут различаться):

```
 row | letter | seat_no
-----+--------+---------
   1 | A      | 1A
   1 | C      | 1C
   1 | D      | 1D
   1 | F      | 1F
   2 | A      | 2A
   2 | C      | 2C
   2 | D      | 2D
   2 | F      | 2F
   3 | A      | 3A
   3 | C      | 3C
   3 | D      | 3D
   3 | F      | 3F
(12 строк)
```

Задание. Попытайтесь объяснить, почему все три варианта работают одинаково, причем правильно. Найдите обоснование в документации. Начать можно с описания команды SELECT, приведенного в документации.

11 Когда в предложении FROM несколько табличных функций

В разделе 5.5 «Конструкция LATERAL и функции» (с. 309) мы рассматривали запрос, в предложении FROM которого была всего одна такая конструкция. Давайте рассмотрим ситуацию, в которой этих конструкций будет две.

На принятие решения руководством авиакомпании о сохранении тех или иных маршрутов влияет в том числе их востребованность у пассажиров. Она зависит от разных факторов: от исторически сложившихся связей между городами, от стоимости авиабилетов, от наличия других транспортных возможностей, от численности населения городов и т. д. Интегральным показателем востребованности будем считать степень заполнения самолетов, выполняющих рейсы по конкретным маршрутам. Мы будем принимать во внимание лишь рейсы, имеющие статус Departed или Arrived.

Начнем разработку с функции вычисления интересующего нас показателя для конкретной пары городов за указанный период, причем для рейсов как «туда», так и «обратно».

```
CREATE OR REPLACE FUNCTION get_routes_occupation(
  dep_city text,
  arr_city text,
  from_date date,
  till_date date
) RETURNS TABLE (
  dep_city text,
  arr_city text,
  total_passengers numeric, -- число перевезенных пассажиров
  total_seats numeric,      -- общее число мест в самолетах
  occupancy_rate numeric    -- доля занятых мест
) AS
$$
  WITH seats_counts AS
  ( SELECT aircraft_code, count( * ) AS seats_cnt
    FROM seats
    GROUP BY aircraft_code
  ),
  per_flight_results AS
  ( SELECT
      r.departure_city,
      r.arrival_city,
      f.flight_id,
      f.aircraft_code,
      count( * ) AS passengers_cnt
  FROM routes AS r
     JOIN flights AS f ON f.flight_no = r.flight_no
     JOIN ticket_flights AS tf ON tf.flight_id = f.flight_id
  WHERE ( ( r.departure_city = dep_city AND r.arrival_city = arr_city ) -- туда
       OR ( r.departure_city = arr_city AND r.arrival_city = dep_city ) -- обратно
        )
    AND f.scheduled_departure BETWEEN from_date AND till_date
    AND f.status IN ( 'Departed', 'Arrived' )
  GROUP BY r.departure_city, r.arrival_city, f.flight_id, f.aircraft_code
)
SELECT
  pfr.departure_city,
  pfr.arrival_city,
  sum( pfr.passengers_cnt ) AS total_passengers,
  sum( sc.seats_cnt ) AS total_seats,
  round( sum( pfr.passengers_cnt ) / sum( sc.seats_cnt ), 2 ) AS occupancy_rate
FROM per_flight_results AS pfr
  JOIN seats_counts AS sc ON sc.aircraft_code = pfr.aircraft_code
GROUP BY pfr.departure_city, pfr.arrival_city;
$$ LANGUAGE sql;
CREATE FUNCTION
```

В первом подзапросе в конструкции WITH вычисляется количество мест в каждой модели самолета.

Во втором подзапросе вычисляется количество пассажиров *на каждом рейсе*, выполненном по одному из двух заданных направлений («туда» и «обратно»). Хотя целью этого подзапроса является вычисление количества пассажиров на каждом рейсе, а не на каждом направлении, тем не менее в группировке участвуют также столбцы «Город отправления», «Город прибытия» и «Код модели самолета», поскольку они будут нужны на заключительном этапе, в главном запросе. Конечно, можно было бы упростить подзапрос, убрав эти столбцы из его списка SELECT и предложения GROUP BY, но тогда пришлось бы включить обращения к таблицам «Маршруты» (routes) и «Рейсы» (flights) и в предложение FROM главного запроса. В результате запрос в целом не стал бы проще.

Тип numeric для столбцов total_passengers и total_seats в таблице, которую формирует функция, выбран потому, что функция count возвращает тип bigint, а функция sum дает для аргумента типа bigint результат типа numeric (см. раздел документации 9.21 «Агрегатные функции»). Отметим также, что при вычислении доли занятых мест операция деления не будет целочисленной и округление будет корректным.

Перейдем к проверке функции get_routes_occupation в работе. Начнем с простейшего случая — зададим ее аргументы в запросе явным образом:

```
SELECT
  dep_city,
  arr_city,
  total_passengers AS pass,
  total_seats AS seats,
  occupancy_rate AS rate
FROM get_routes_occupation( 'Владивосток', 'Москва', '2017-08-01', '2017-08-15' );
  dep_city   |  arr_city   | pass | seats | rate
-------------+-------------+------+-------+------
 Владивосток | Москва      |  461 |  3108 | 0.15
 Москва      | Владивосток |  567 |  3108 | 0.18
(2 строки)
```

Самолеты летают полупустые, возможно, из-за очень высоких цен на билеты.

Теперь посмотрим, как часто летают в Москву жители самых восточных регионов России, то есть тех, в которых аэропорты расположены восточнее долготы 150 градусов:

```
SELECT
  gro.dep_city,
  gro.arr_city,
  gro.total_passengers AS pass,
  gro.total_seats AS seats,
  gro.occupancy_rate AS rate
FROM airports
  CROSS JOIN LATERAL
    get_routes_occupation( city, 'Москва', '2017-08-01', '2017-08-15' ) AS gro
WHERE coordinates[ 0 ] > 150;
```

Напомним, что столбец coordinates имеет тип данных point (точка). Для обращения к отдельным координатам точки используется та же нотация, что и для обращения к массивам. В элементе с индексом 0 записана географическая долгота, а в элементе с индексом 1 — географическая широта.

```
        dep_city         |         arr_city         | pass | seats | rate
-------------------------+--------------------------+------+-------+------
 Москва                  | Петропавловск-Камчатский |  449 |  1332 | 0.34
 Петропавловск-Камчатский | Москва                   |  415 |  1332 | 0.31
 Анадырь                 | Москва                   |   64 |   232 | 0.28
 Москва                  | Анадырь                  |   92 |   232 | 0.40
(4 строки)
```

В выполненном запросе первый аргумент функции брался из текущей строки таблицы «Аэропорты» (airports), а второй оставался неизменным.

Давайте усложним задачу: нужно определить степень заполнения самолетов на всех направлениях, проложенных из каждого города, находящегося, например, в часовом поясе Asia/Vladivostok. Очевидно, придется каким-то образом определять список всех городов, с которыми имеет авиасообщение конкретный город, а затем подставлять полученные названия городов поочередно в качестве второго аргумента функции get_routes_occupation, которая будет вызываться для каждого города из выбранного часового пояса.

Функция, формирующая список городов, в которые можно улететь из указанного города, будет несложной.

Она возвращает множество строк, состоящих из одного поля, поэтому в предложении RETURNS SETOF лучше написать не record, а имя конкретного скалярного типа — у нас это тип text.

```
CREATE OR REPLACE FUNCTION list_connected_cities(
  city text,
  OUT connected_city text
)
RETURNS SETOF text AS
$$
  SELECT DISTINCT arrival_city
  FROM routes
  WHERE departure_city = city;
$$ LANGUAGE sql;
```

```
CREATE FUNCTION
```

Поскольку для каждого маршрута существует обратный маршрут, то города, в которые можно *улететь* из данного города, совпадают с городами, из которых можно *прилететь* в данный город. Следовательно, если в запросе поменять местами имена столбцов arrival_city и departure_city, получим тот же самый список городов. Поэтому можно выбрать любой из двух вариантов запроса.

Проверим эту функцию в работе:

```
SELECT city, connected_city
FROM airports AS a
  CROSS JOIN list_connected_cities( city ) AS connected_city
WHERE timezone = 'Asia/Vladivostok'
ORDER BY city, connected_city;
```

```
         city         | connected_city
----------------------+----------------
 Владивосток          | Иркутск
 Владивосток          | Москва
 Владивосток          | Хабаровск
 Комсомольск-на-Амуре | Екатеринбург
 Хабаровск            | Анадырь
 Хабаровск            | Благовещенск
 Хабаровск            | Владивосток
 Хабаровск            | Москва
 Хабаровск            | Санкт-Петербург
 Хабаровск            | Усть-Илимск
 Хабаровск            | Южно-Сахалинск
(11 строк)
```

Теперь, имея функцию list_connected_cities, можно решить поставленную выше задачу. В предложении FROM поставим вызов функции list_connected_cities *левее* вызова функции get_routes_occupation, чтобы вторая функция могла ссылаться на результаты работы первой.

Напомним: поскольку в предложении FROM используются функции, ключевое слово LATERAL является необязательным.

Запрос выполняется относительно долго:

```
SELECT
  gro.dep_city,
  gro.arr_city,
  gro.total_passengers AS pass,
  gro.total_seats AS seats,
  gro.occupancy_rate AS rate
FROM airports AS a
  CROSS JOIN LATERAL list_connected_cities( a.city ) AS lcc
  CROSS JOIN LATERAL
    get_routes_occupation( a.city, lcc.connected_city, '2017-08-01', '2017-08-15' )
    AS gro
WHERE timezone = 'Asia/Vladivostok'
ORDER BY gro.dep_city, gro.arr_city;
```

```
       dep_city       |       arr_city       | pass | seats | rate
----------------------+----------------------+------+-------+------
 Анадырь              | Хабаровск            |   90 |   232 | 0.39
 Благовещенск         | Хабаровск            |  639 |  1358 | 0.47
 Владивосток          | Иркутск              |  122 |   700 | 0.17
 Владивосток          | Москва               |  461 |  3108 | 0.15
 Владивосток          | Хабаровск            | 1201 |  1358 | 0.88
 Владивосток          | Хабаровск            | 1201 |  1358 | 0.88
 Екатеринбург         | Комсомольск-на-Амуре |  127 |   888 | 0.14
 Иркутск              | Владивосток          |  113 |   700 | 0.16
 Комсомольск-на-Амуре | Екатеринбург         |  117 |   888 | 0.13
 ...
 Усть-Илимск          | Хабаровск            |  122 |   200 | 0.61
 Хабаровск            | Анадырь              |   85 |   232 | 0.37
 Хабаровск            | Благовещенск         |  622 |  1358 | 0.46
 Хабаровск            | Владивосток          | 1171 |  1358 | 0.86
 Хабаровск            | Владивосток          | 1171 |  1358 | 0.86
 Хабаровск            | Москва               | 2835 |  3108 | 0.91
 Хабаровск            | Санкт-Петербург      | 1797 |  3108 | 0.58
 Хабаровск            | Усть-Илимск          |  119 |   300 | 0.40
 Хабаровск            | Южно-Сахалинск       |   85 |   168 | 0.51
 Южно-Сахалинск       | Хабаровск            |   89 |   168 | 0.53
(22 строки)
```

В выборке повторяются строки «Владивосток — Хабаровск» и «Хабаровск — Владивосток», поскольку оба этих города находятся в одном часовом поясе Asia/Vladivostok.

Посмотрим план запроса:

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Sort (actual rows=22 loops=1)
   Sort Key: gro.dep_city, gro.arr_city
   Sort Method: quicksort  Memory: 26kB
   -> Nested Loop (actual rows=22 loops=1)
       -> Nested Loop (actual rows=11 loops=1)
           -> Seq Scan on airports_data ml (actual rows=3 loops=1)
               Filter: (timezone = 'Asia/Vladivostok'::text)
               Rows Removed by Filter: 101
           -> Function Scan on list_connected_cities lcc (actual rows=4 loops=3)
       -> Function Scan on get_routes_occupation gro (actual rows=2 loops=11)
 Planning Time: 0.110 ms
 Execution Time: 11354.892 ms
(12 строк)
```

Прежде чем перейти к обсуждению плана, напомним, что объект «Аэропорты» (airports) на самом деле является представлением, за которым скрывается таблица airports_data.

В этом плане мы видим двойной вложенный цикл. Работа начинается с отбора трех строк из таблицы airports_data, для каждой из которых вызывается функция list_connected_cities. Она порождает *в среднем* по четыре строки при каждом вызове (показатели actual rows=4 loops=3), а общее число порожденных строк равно 11, как свидетельствует показатель actual rows=11 во внутреннем узле Nested Loop.

Рассуждая аналогично, можно заключить, что во внешнем вложенном цикле для каждой из одиннадцати строк, порожденных во внутреннем цикле, вызывается функция get_routes_occupation, выдающая по две строки за один вызов (actual rows=2 loops=11). В результате число сформированных строк становится равным 22.

Задание. В последнем запросе, приведенном в тексте упражнения, отчетный период задавался двумя параметрами функции get_routes_occupation. Модифицируйте запрос таким образом, чтобы можно было задать несколько отчетных периодов. Если в конкретном периоде не было ни одного рейса, итоговая строка все равно должна быть сформирована. Добавьте два столбца в список SELECT — начало и конец отчетного периода.

Указание. Можно воспользоваться такой конструкцией (временны́е периоды могут быть другими):

```
unnest(
  ARRAY[ '2017-07-16', '2017-08-01', '2017-09-01' ]::date[],
  ARRAY[ '2017-07-31', '2017-08-31', '2017-09-15' ]::date[]
) AS periods( from_date, till_date )
```

12 Табличные функции в списке SELECT? Иногда можно, но все же лучше в предложении FROM

Табличные функции можно вызывать в списке SELECT. Иногда это упрощает запрос, однако зачастую вызывает затруднения, которых не бывает при использовании функций в предложении FROM.

Давайте решим такую задачу: требуется сформировать списки городов, в которые есть рейсы из Томска, Красноярска, Кемерова и Новосибирска. Причем эти списки нужно поместить в отдельные колонки, чтобы было удобно сопоставлять полученные результаты. Воспользуемся функцией list_connected_cities, рассмотренной в предыдущем упражнении. Запрос будет таким:

```
SELECT
  list_connected_cities( 'Томск' ) AS "Томск",
  list_connected_cities( 'Красноярск' ) AS "Красноярск",
  list_connected_cities( 'Кемерово' ) AS "Кемерово",
  list_connected_cities( 'Новосибирск' ) AS "Новосибирск";
```

```
     Томск      |  Красноярск   | Кемерово |  Новосибирск
----------------+---------------+----------+----------------
 Абакан         | Барнаул       | Кызыл    | Абакан
 Архангельск    | Курган        | Москва   | Горно-Алтайск
 Волгоград      | Москва        | Удачный  | Калуга
 Москва         | Новокузнецк   |          | Киров
 Новокузнецк    | Новосибирск   |          | Красноярск
 Ростов-на-Дону | Новый Уренгой |          | Магнитогорск
 Ярославль      | Советский     |          | Мирный
                | Сочи          |          | Москва
                | Усть-Илимск   |          | Новокузнецк
                | Усть-Кут      |          | Пермь
                |               |          | Салехард
                |               |          | Сургут
                |               |          | Удачный
                |               |          | Ханты-Мансийск
(14 строк)
```

В разделе документации 36.5.9 «Функции SQL, возвращающие множества» сказано, что при вызове табличных функций в списке SELECT первая результирующая строка формируется из первых строк, возвращаемых функциями, вторая — из вторых строк и так далее. Общее число строк определяется функцией, возвратившей больше всего строк, а результат остальных функций дополняется при необходимости значениями NULL, как в нашем примере.

Можно получить такой же результат, вызывая функции не в списке SELECT, а в предложении FROM с помощью конструкции ROWS FROM (...), которую PostgreSQL предлагает как расширение стандарта SQL. Эта конструкция представлена в документации в описании команды SELECT и в подразделе 7.2.1.4 «Табличные функции». Вот каким станет запрос:

```
SELECT *
FROM ROWS FROM
  ( list_connected_cities( 'Томск' ),
    list_connected_cities( 'Красноярск' ),
    list_connected_cities( 'Кемерово' ),
    list_connected_cities( 'Новосибирск' )
  ) AS lcc( "Томск", "Красноярск", "Кемерово", "Новосибирск" );
```

```
    Томск       |  Красноярск   | Кемерово |  Новосибирск
----------------+---------------+----------+----------------
 Абакан         | Барнаул       | Кызыл    | Абакан
 Архангельск    | Курган        | Москва   | Горно-Алтайск
 Волгоград      | Москва        | Удачный  | Калуга
 Москва         | Новокузнецк   |          | Киров
 Новокузнецк    | Новосибирск   |          | Красноярск
 Ростов-на-Дону | Новый Уренгой |          | Магнитогорск
 Ярославль      | Советский     |          | Мирный
                | Сочи          |          | Москва
                | Усть-Илимск   |          | Новокузнецк
                | Усть-Кут      |          | Пермь
                |               |          | Салехард
                |               |          | Сургут
                |               |          | Удачный
                |               |          | Ханты-Мансийск
(14 строк)
```

Обратите внимание на использование одинарных и двойных кавычек. Псевдонимы результирующих столбцов выборки можно было задать вообще без кавычек (хотя называть столбцы по-русски в принципе не рекомендуется):

```
...
  ) AS lcc( Томск, Красноярск, Кемерово, Новосибирск );
```

В обоих запросах не было предложения ORDER BY. При этом названия городов в каждой колонке оказались отсортированными, хотя и в коде функции list_connected_cities также нет предложения ORDER BY. Это случайный результат. Но если бы в коде функции и была предусмотрена сортировка, полагаться на то, что ее результат сохранится и на уровне всего запроса, нельзя. В описании команды SELECT, приведенном в документации, сказано следующее: «Если присутствует предложение ORDER BY, возвращаемые строки сортируются в указанном порядке. В отсутствие ORDER BY строки возвращаются в том порядке, в каком системе будет проще их выдать». Напомним, что при использовании оконных функций также рекомендуется задавать предложение ORDER BY на верхнем уровне запроса, а не полагаться на сортировку, выполненную при формировании окна. Об этом говорится в подразделе документации 7.2.5 «Обработка оконных функций».

Но даже предложение ORDER BY в приведенных запросах не может гарантировать получение правильного результата, поскольку в нашем примере значения в каждой колонке должны сортироваться *независимо* от других колонок, а предложение ORDER BY не может этого обеспечить.

Таким образом, рассмотренные решения поставленной задачи нельзя считать корректными. Они были бы приемлемыми, если бы порядок сортировки не был критичным.

Можно предложить решение с вызовом функций в предложении FROM:

```
WITH cities(col, city, num) AS
( SELECT 1, *
  FROM list_connected_cities( 'Томск' ) WITH ORDINALITY
  UNION ALL
  SELECT 2, *
  FROM list_connected_cities( 'Красноярск' ) WITH ORDINALITY
  UNION ALL
  SELECT 3, *
  FROM list_connected_cities( 'Кемерово' ) WITH ORDINALITY
  UNION ALL
  SELECT 4, *
  FROM list_connected_cities( 'Новосибирск' ) WITH ORDINALITY
)
```

```
SELECT num,
  any_value( city ) FILTER ( WHERE col = 1 ) AS "Томск",
  any_value( city ) FILTER ( WHERE col = 2 ) AS "Красноярск",
  any_value( city ) FILTER ( WHERE col = 3 ) AS "Кемерово",
  any_value( city ) FILTER ( WHERE col = 4 ) AS "Новосибирск"
FROM cities
GROUP BY num
ORDER BY num;
```

```
 num |     Томск      |  Красноярск   | Кемерово |  Новосибирск
-----+----------------+---------------+----------+----------------
   1 | Абакан         | Барнаул       | Кызыл    | Абакан
   2 | Архангельск    | Курган        | Москва   | Горно-Алтайск
   3 | Волгоград      | Москва        | Удачный  | Калуга
   4 | Москва         | Новокузнецк   |          | Киров
   5 | Новокузнецк    | Новосибирск   |          | Красноярск
   6 | Ростов-на-Дону | Новый Уренгой |          | Магнитогорск
   7 | Ярославль      | Советский     |          | Мирный
   8 |                | Сочи          |          | Москва
   9 |                | Усть-Илимск   |          | Новокузнецк
  10 |                | Усть-Кут      |          | Пермь
  11 |                |               |          | Салехард
  12 |                |               |          | Сургут
  13 |                |               |          | Удачный
  14 |                |               |          | Ханты-Мансийск
(14 строк)
```

В списке SELECT, в отличие от предложения FROM, табличные функции могут быть вложенными. В подразделе документации 36.5.9 «Функции SQL, возвращающие множества» приведен абстрактный пример использования таких конструкций (srf — set returning function):

```
SELECT
  srf1( srf2( x ), srf3( y ) ),
  srf4( srf5( z ) )
FROM tab;
```

Там сказано, что каждый уровень вложенности обрабатывается отдельно, как будто он формируется с помощью отдельной конструкции LATERAL ROWS FROM(...). В приведенном примере для каждой строки из таблицы tab сначала будут вызваны функции srf2, srf3 и srf5, а затем для каждой комбинированной строки, сформированной ими, будут вызваны функции srf1 и srf4. Конечно же, все эти функции могут возвращать разное число строк; значения NULL в результирующих строках могут приводить к неожиданным результатам.

Давайте снова обратимся к примеру, рассмотренному в предыдущем упражнении, — определению степени заполнения самолетов на всех направлениях, проложенных из городов часового пояса Asia/Vladivostok. Сначала покажем, куда можно улететь из этих городов:

```
SELECT city, list_connected_cities( city ) AS connected_city
FROM airports
WHERE timezone = 'Asia/Vladivostok'
ORDER BY city, connected_city;
```

```
         city          | connected_city
-----------------------+----------------
 Владивосток           | Иркутск
 Владивосток           | Москва
 Владивосток           | Хабаровск
 Комсомольск-на-Амуре  | Екатеринбург
 Хабаровск             | Анадырь
 Хабаровск             | Благовещенск
 Хабаровск             | Владивосток
 Хабаровск             | Москва
 Хабаровск             | Санкт-Петербург
 Хабаровск             | Усть-Илимск
 Хабаровск             | Южно-Сахалинск
(11 строк)
```

Функция get_routes_occupation имеет четыре параметра: названия городов отправления и прибытия, а также даты начала и конца отчетного периода. Значения всех аргументов, кроме первого, сформируем с помощью функций list_connected_cities и unnest. Нам пришлось сделать два вызова функции unnest, потому что получать несколько аргументов-массивов она может только при использовании в предложении FROM.

```
SELECT get_routes_occupation(
    city,
    list_connected_cities( city ),
    unnest( ARRAY[ '2017-07-16', '2017-08-01' ]::date[] ),
    unnest( ARRAY[ '2017-07-31', '2017-08-15' ]::date[] )
  )
FROM airports
WHERE timezone = 'Asia/Vladivostok'
ORDER BY city;
```

Выведем результат в виде составных значений. Обратите внимание на порядок вывода строк: полеты «туда» и «обратно» идут парами.

```
                get_routes_occupation
-------------------------------------------------
 (Владивосток,Иркутск,79,700,0.11)
 (Иркутск,Владивосток,82,750,0.11)
 (Владивосток,Москва,461,3108,0.15)
 (Москва,Владивосток,567,3108,0.18)
 (Екатеринбург,Комсомольск-на-Амуре,31,444,0.07)
 (Комсомольск-на-Амуре,Екатеринбург,120,1110,0.11)
 (Анадырь,Хабаровск,45,116,0.39)
 (Хабаровск,Анадырь,17,116,0.15)
 (Благовещенск,Хабаровск,639,1358,0.47)
 (Хабаровск,Благовещенск,622,1358,0.46)
(10 строк)
```

Всего получено 10 строк, хотя предполагалось 44: полеты по одиннадцати направлениям совершаются «туда» и «обратно», при этом отчет формируется за два периода. Прояснить причину столь неожиданного результата нам поможет следующий запрос, который показывает, какие аргументы получает функция get_routes_occupation:

```
SELECT
  city,
  list_connected_cities( city ),
  unnest( ARRAY[ '2017-07-16', '2017-08-01' ]::date[] ),
  unnest( ARRAY[ '2017-07-31', '2017-08-15' ]::date[] )
FROM airports
WHERE timezone = 'Asia/Vladivostok'
ORDER BY 1, 2, 3, 4;
```

При вызове табличной функции непосредственно в списке SELECT число строк, попадающих в выборку, зависит от числа строк, возвращаемых функцией для каждой исходной строки, отобранной в предложении FROM с учетом условия WHERE. Если функция возвращает *N* строк, то в выборку идет *N* строк, при этом значения столбцов, представленных в списке SELECT, будут в этих строках дублироваться. В нашем примере дублируются значения столбца city. Если же функция не возвращает ни одной строки, то и в выборку не попадает ни одной строки — исходная строка «исчезает», и результат может оказаться неполным.

В подразделе документации 36.5.9 «Функции SQL, возвращающие множества» сказано, что функции, вызываемые в списке SELECT, всегда вычисляются таким образом, как будто они размещены во внутреннем цикле конструкции Nested

Loop, во внешнем цикле которой идет перебор строк, порождаемых в предложении FROM. Поэтому функции вычисляются до *полного* завершения, прежде чем будет взята следующая строка из внешнего набора строк.

В выборке есть значения NULL. Причина их появления уже обсуждалась выше. Только в пяти строках эти значения отсутствуют. Для каждой из них функция get_routes_occupation возвращает по две строки, что в результате и дает всего десять строк.

```
         city          | list_connected_cities |   unnest   |   unnest
-----------------------+-----------------------+------------+------------
 Владивосток           | Иркутск               | 2017-07-16 | 2017-07-31
 Владивосток           | Москва                | 2017-08-01 | 2017-08-15
 Владивосток           | Хабаровск             |            |
 Комсомольск-на-Амуре  | Екатеринбург          | 2017-07-16 | 2017-07-31
 Комсомольск-на-Амуре  |                       | 2017-08-01 | 2017-08-15
 Хабаровск             | Анадырь               | 2017-07-16 | 2017-07-31
 Хабаровск             | Благовещенск          | 2017-08-01 | 2017-08-15
 Хабаровск             | Владивосток           |            |
 Хабаровск             | Москва                |            |
 Хабаровск             | Санкт-Петербург       |            |
 Хабаровск             | Усть-Илимск           |            |
 Хабаровск             | Южно-Сахалинск        |            |
(12 строк)
```

Из условия следует, что для получения правильного результата нам нужно было каждый маршрут, то есть пару городов, скомбинировать со всеми временны́ми интервалами. Задачу легко выполнить, вызывая функции в предложении FROM, но сделать это в списке SELECT — нетривиальная задача. В результате в предыдущем запросе функция get_routes_occupation вызывается не со всеми требуемыми комбинациями аргументов, что и приводит к некорректному результату.

Задание 1. Выше был рассмотрен следующий запрос:

```
SELECT get_routes_occupation(
    city,
    list_connected_cities( city ),
    unnest( ARRAY[ '2017-07-16', '2017-08-01' ]::date[] ),
    unnest( ARRAY[ '2017-07-31', '2017-08-15' ]::date[] )
  )
FROM airports
WHERE timezone = 'Asia/Vladivostok'
ORDER BY city;
```

Попробуйте объяснить порядок вывода строк, полученный в результате его выполнения. Обратитесь к плану запроса, найдите в нем узел Sort.

Указание. За справкой можно обратиться к подразделу «Список SELECT» описания команды SELECT, приведенного в документации. Там сказано, когда должны вычисляться выражения в выходном списке SELECT при наличии в запросе предложений DISTINCT, ORDER BY или LIMIT.

Задание 2. Выполните модифицированный запрос, в котором формируются отдельные столбцы, а не составные значения:

```
SELECT (
    get_routes_occupation(
      city,
      list_connected_cities( city ),
      unnest( ARRAY[ '2017-07-16', '2017-08-01' ]::date[] ),
      unnest( ARRAY[ '2017-07-31', '2017-08-15' ]::date[] )
    )
  ).*
FROM airports
WHERE timezone = 'Asia/Vladivostok'
ORDER BY 1, 2;
```

В предложении ORDER BY заданы номера столбцов, чтобы не повторять здесь вызовы функции с указанием их имен таким образом:

```
...
ORDER BY
  ( get_routes_occupation( city, ... ) ).dep_city,
  ( get_routes_occupation( city, ... ) ).arr_city;
```

В каком порядке выводятся строки теперь? Почему? В узле Sort плана запроса обратите внимание на выражения, соответствующие столбцам 1 и 2 из предложения ORDER BY.

Задание 3. В подразделе «Список SELECT» описания команды SELECT, приведенного в документации, сказано, что выходные выражения, содержащие функции, возвращающие множества, фактически вычисляются до ограничения количества строк, так что LIMIT будет отбрасывать уже не исходные строки, а строки, выдаваемые функцией, возвращающей множество. Проиллюстрируйте это положение документации на примере запросов, рассмотренных в упражнении.

Задание 4. В тексте упражнения был приведен вариант запроса, выводящего списки городов в четыре колонки с вызовом функций в предложении FROM. Попробуйте детально разобраться в алгоритме его выполнения.

Указание. Примите к сведению следующее:

- предложение WITH ORDINALITY нумерует строки, возвращаемые функцией list_connected_cities;
- агрегатная функция any_value возвращает одно из значений, входящих в группу;
- предложение FILTER отбирает из группы ровно одну строку (точнее говоря, не более одной, ведь число городов в группах не одинаковое).

Вопрос. Можно ли было вместо any_value воспользоваться какой-нибудь другой агрегатной функцией, например min или avg?

Попутно заметим, что данная функция не может заменить функцию random, когда требуется выбрать случайную строку. В качестве примера выполните следующие запросы несколько раз:

```
SELECT any_value( model )
FROM aircrafts;
```

```
SELECT model
FROM aircrafts
ORDER BY random()
LIMIT 1;
```

13 Стабильные функции могут зависеть от настроек сервера

Целый ряд стабильных функций зависит от настроек сервера. Например, функция CURRENT_TIMESTAMP зависит от настройки *timezone*.

Давайте покажем это.

```
BEGIN;
BEGIN
SET timezone = 'Asia/Vladivostok';
SET
```

```
SELECT CURRENT_TIMESTAMP;
       current_timestamp
-------------------------------
 2025-02-19 22:10:05.363264+10
(1 строка)
SET timezone = 'Europe/Volgograd';
SET
SELECT CURRENT_TIMESTAMP;
       current_timestamp
-------------------------------
 2025-02-19 15:10:05.363264+03
(1 строка)
ROLLBACK;
ROLLBACK
```

Задание 1. Покажите, что стабильная функция age (см. раздел документации 9.9 «Операторы и функции даты/времени») также зависит от настройки сервера *timezone*.

Задание 2. Покажите, что стабильная функция CURRENT_TIMESTAMP зависит от настройки сервера *datestyle*.

14 Функция random. Как получить одни и те же значения при многократном вызове?

Функция random является изменчивой (VOLATILE) функцией. В тексте главы был показан результат ее работы при многократном вызове. Но ведь возможны ситуации, когда для каждой строки, формируемой в запросе, требуется случайное значение — но одно и то же.

Каким образом можно получить примерно такой результат?

```
        rand1         |        rand2
----------------------+---------------------
 0.48377601191203623 | 0.48377601191203623
 0.48377601191203623 | 0.48377601191203623
 0.48377601191203623 | 0.48377601191203623
 0.48377601191203623 | 0.48377601191203623
 0.48377601191203623 | 0.48377601191203623
(5 строк)
```

15 Функция random. Неожиданные результаты

При неосторожном обращении с функцией random можно получить неожиданные результаты. Давайте проведем несколько экспериментов.

Это исходный вариант запроса. Все полученные значения функции различны:

```
SELECT g, random(), random()
FROM generate_series( 1, 3 ) AS g;
 g |       random        |       random
---+---------------------+---------------------
 1 | 0.23064923406295246 |  0.5225944595794347
 2 |  0.7226202894085108 |  0.7062034393666694
 3 |  0.7973246708283166 | 0.18759034972053756
(3 строки)
```

Добавим сортировку по второму столбцу. Получаем два одинаковых значения в каждой строке:

```
SELECT g, random(), random()
FROM generate_series( 1, 3 ) AS g
ORDER BY 2;
 g |       random        |       random
---+---------------------+---------------------
 3 |  0.3346565036686884 |  0.3346565036686884
 2 | 0.49660504014015094 | 0.49660504014015094
 1 |  0.9872848197321895 |  0.9872848197321895
(3 строки)
```

Если вместо ORDER BY 2 написать ORDER BY random(), картина будет аналогичная:

```
SELECT g, random(), random()
FROM generate_series( 1, 3 ) AS g
ORDER BY random();
 g |       random        |       random
---+---------------------+---------------------
 2 | 0.20802852099478542 | 0.20802852099478542
 1 | 0.23397408247599127 | 0.23397408247599127
 3 |  0.8469532119175736 |  0.8469532119175736
(3 строки)
```

Но если сортировать по двум столбцам, то при ORDER BY 2, 3 все значения будут разные, а ORDER BY random(), random() даст по два одинаковых значения в каждой строке:

```
SELECT g, random(), random()
FROM generate_series( 1, 3 ) AS g
ORDER BY 2, 3;
 g |       random       |       random
---+--------------------+--------------------
 1 | 0.3978504956287461 |   0.471958614877404
 2 | 0.6591215954377037 | 0.10242505792578593
 3 | 0.9557601179231214 | 0.40101325415299427
(3 строки)
```

```
SELECT g, random(), random()
FROM generate_series( 1, 3 ) AS g
ORDER BY random(), random();
 g |       random        |       random
---+---------------------+---------------------
 3 | 0.19391311224220686 | 0.19391311224220686
 1 |  0.4914591144401763 |  0.4914591144401763
 2 |  0.5732131236638698 |  0.5732131236638698
(3 строки)
```

Задание. Попробуйте объяснить полученные результаты. Кроме команды EXPLAIN с параметрами ANALYZE и VERBOSE, в этом вам может помочь описание команды SELECT (подраздел «Предложение ORDER BY»), приведенное в документации.

16 Поиск корней квадратного уравнения

На языке SQL можно написать функцию, вычисляющую, например, действительные корни квадратного уравнения. В конструкции WITH сначала вычисляется дискриминант, а затем — корни уравнения. В главном запросе значения корней округляются с требуемой точностью. Обратите внимание, что подзапрос в вызове функции sqrt заключается в скобки.

```
CREATE OR REPLACE FUNCTION square_equation(
  a double precision,
  b double precision,
  c double precision,
  accuracy integer DEFAULT 2,
  OUT x1 numeric,
  OUT x2 numeric
) AS
$$
  WITH discriminant AS
  ( SELECT b * b - 4 * a * c AS d
  ),
  roots AS
  ( SELECT
      ( -b + sqrt( ( SELECT d FROM discriminant ) ) ) / ( 2 * a ) AS x_one,
      ( -b - sqrt( ( SELECT d FROM discriminant ) ) ) / ( 2 * a ) AS x_two
  )
  SELECT
    round( x_one::numeric, accuracy ) AS x_one,
    round( x_two::numeric, accuracy ) AS x_two
FROM roots;
$$ LANGUAGE sql;
CREATE FUNCTION
```

Обратите внимание, что в главном запросе псевдонимы для имен корней уравнения x_one и x_two (после слова AS) совпадают с именами столбцов x_one и x_two из подзапроса roots (аргументы функции round). Конечно, делать так совсем не обязательно, но это тоже работает.

Проверим работу функции:

```
SELECT square_equation( 3, -6, 2 );
 square_equation
-----------------
 (1.58,0.42)
(1 строка)
```

Зададим точность округления результатов до четырех цифр:

```
SELECT square_equation( 3, -6, 2, 4 );
 square_equation
-----------------
 (1.5774,0.4226)
(1 строка)
```

В предыдущих запросах, когда функция вызывалась непосредственно в предложении SELECT, результатом было значение составного типа. Если же вызвать функцию в предложении FROM, два ее результирующих значения будут выведены как отдельные столбцы:

```
SELECT * FROM square_equation( 3, -6, 2 );
  x1  |  x2
------+------
 1.58 | 0.42
(1 строка)
```

В случае ошибки работа функции прекращается:

```
SELECT square_equation( 3, -6, 4 );
ОШИБКА:  извлечь квадратный корень отрицательного числа нельзя
КОНТЕКСТ:  SQL-функция "square_equation", оператор 1
SELECT * FROM square_equation( 0, -6, 2 );
ОШИБКА:  деление на ноль
КОНТЕКСТ:  SQL-функция "square_equation", оператор 1
```

Вопрос. При создании этой функции мы не указали категорию изменчивости, поэтому по умолчанию принимается VOLATILE. А можно ли назначить категорию изменчивости STABLE или IMMUTABLE? Почему?

Задание 1. Создайте таблицу coeffs, содержащую значения коэффициентов уравнений (столбцы a, b и c). Введите в нее несколько строк. Напишите запрос, в котором функция решает все уравнения, определяемые коэффициентами из каждой строки таблицы. Представьте результаты не только в виде составных значений, но также и в виде отдельных столбцов. Например:

```
SELECT x1( square_equation( a, b, c ) ), x2( square_equation( a, b, c ) )
FROM coeffs;

SELECT ( square_equation( a, b, c ) ).*
FROM coeffs;

SELECT x1, x2
FROM coeffs, square_equation( a, b, c );
```

Как вы думаете, есть ли принципиальное различие между первым и вторым вариантами? Проверьте ваши предположения с помощью команды EXPLAIN с параметрами ANALYZE и VERBOSE.

Задание 2. Предложите вашу функцию, решающую уравнения другого вида или выполняющую какие-то вычисления на основе базы данных «Авиаперевозки». В качестве примера можно рассмотреть расчет заработной платы пилотов в зависимости от оклада, районного коэффициента и различных персональных надбавок, а также с учетом налогов. Вызовите вашу функцию в запросе, возвращающем более одной строки, чтобы функция проводила вычисления на основе различных исходных значений, получаемых из базы данных. При необходимости создайте дополнительные таблицы.

17 Параметр конфигурации сервера можно изменить на время выполнения функции

В подразделе 5.6.1 «Влияние изменчивости на выбор оптимального плана» (с. 316) мы рассматривали функцию `get_rand_num` для формирования номера «счастливого» кресла. Одним из пожеланий к процедуре определения этого номера была воспроизводимость результата во всех отделениях компании, находящихся в разных часовых поясах.

Номер «счастливого» кресла для конкретной даты формируется на основе количества пассажиров, перевезенных нашей авиакомпанией за этот день. Однако в разных часовых поясах конкретная дата начинается в разное астрономическое время и заканчивается также в разное астрономическое время. Поэтому множества рейсов, попадающих в конкретную дату, могут различаться от одного часового пояса к другому. Следовательно, число перевезенных пассажиров также будет различаться. В результате в разных часовых поясах будут формироваться разные номера «счастливых» кресел.

Результаты были бы корректными, если бы мы могли при выполнении запроса с функцией `get_rand_num` в любом часовом поясе приводить отсчет времени к одному и тому же часовому поясу, например Europe/Moscow.

Это можно выполнить, даже не модифицируя код функции, а устанавливая на время ее выполнения значение параметра сервера *timezone*. В команде `CREATE FUNCTION` для этого есть предложение `SET`. Предусмотрено оно и в команде `ALTER FUNCTION`.

```
ALTER FUNCTION get_rand_num SET timezone = 'Europe/Moscow';
ALTER FUNCTION
```

Задание. Выполните запрос для определения номера «счастливого» кресла. Затем установите в текущем сеансе работы значение часового пояса, отличающееся от вашего, скажем, Asia/Vladivostok.

```
SET timezone = 'Asia/Vladivostok';
SET
```

Повторите запрос. Совпадают ли результаты?

18 Видит ли изменчивая функция изменения, произведенные конкурентной транзакцией?

В разделе 5.6.2 «Видимость изменений» (с. 338) мы показали, что изменчивая функция видит изменения в базе данных, произведенные запросом, из которого она вызвана. Те эксперименты проводились в одной транзакции. Давайте сейчас проведем аналогичный эксперимент уже с двумя конкурентными транзакциями. Для упрощения эксперимента будем задавать порядковые номера посадочных талонов константами 1 и 2.

Функция `boarding_info` должна иметь категорию изменчивости `VOLATILE`:

```
ALTER FUNCTION boarding_info VOLATILE;
ALTER FUNCTION
```

Нужно привести базу данных в исходное состояние, как мы делали это ранее, чтобы избежать дублирования данных:

```
DELETE FROM boarding_passes
WHERE flight_id = 13841;
DELETE 1
```

На первом терминале в качестве уровня изоляции транзакций выберем Read Committed:

```
BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
```

Начинаем выполнение транзакции и, используя функцию `pg_sleep`, задержим выполнение запроса, чтобы было достаточно времени для выполнения транзакции на втором терминале.

```
WITH make_boarding AS
( INSERT INTO boarding_passes ( ticket_no, flight_id, boarding_no, seat_no )
  VALUES ( '0005433846800', 13841, 1, '1A' )
  RETURNING *
)
SELECT bi.*
FROM
  pg_sleep( 40 ),
  make_boarding AS mb,
  boarding( mb.flight_id, mb.boarding_no, 15.0, 12.5 ) AS b,
  boarding_info( mb.flight_id ) AS bi \gx
```

На втором терминале выполняем вторую транзакцию и фиксируем ее.

```
BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
WITH make_boarding AS
( INSERT INTO boarding_passes ( ticket_no, flight_id, boarding_no, seat_no )
  VALUES ( '0005432003745', 13841, 2, '6A' )
  RETURNING *
)
SELECT bi.*
FROM
  make_boarding AS mb,
  boarding( mb.flight_id,
  mb.boarding_no, 18.2, 10.3, 12.8 ) AS b,
  boarding_info( mb.flight_id ) AS bi \gx
```

Как мы уже установили ранее, функция boarding_info, имея категорию изменчивости VOLATILE, увидит изменения базы данных, выполненные запросом, в котором она вызвана. Поэтому запрос выдаст такой результат:

```
-[ RECORD 1 ]--------+------
flight_id            | 13841
total_passengers     | 1
total_luggage_pieces | 3
total_luggage_weight | 41.3
END;
COMMIT
```

Возвращаемся на первый терминал. Выведенный после паузы результат показывает: изменчивая функция видит не только изменения, выполненные запросом, который ее вызвал, но и изменения, выполненные и зафиксированные второй транзакцией, хотя она началась после начала выполнения этого запроса:

```
-[ RECORD 1 ]--------+------
flight_id            | 13841
total_passengers     | 2
total_luggage_pieces | 5
total_luggage_weight | 68.8
END;
COMMIT
```

Вопрос. Как вы думаете, чем объясняется полученный результат с позиции снимка базы данных и правил видимости изменений?

Задание 1. Проведите этот же эксперимент с уровнями изоляции транзакций Repeatable Read и Serializable. Попробуйте объяснить полученные результаты. Позволяет ли изменчивая функция обойти ограничения, накладываемые на взаимодействие транзакций на уровне изоляции Serializable? Вспомните, что означает концепция сериализации транзакций.

Перед повторением экспериментов не забудьте привести базу данных в исходное состояние, как мы делали это ранее:

```
DELETE FROM boarding_passes
WHERE flight_id = 13841;
DELETE 2
```

19 А если вызвать изменчивую функцию из стабильной или постоянной?

В разделе документации 36.7 «Категории изменчивости функций» (см. последнее примечание) сказано, что для предотвращения модификации данных PostgreSQL требует, чтобы стабильные и постоянные функции не содержали иных SQL-команд, кроме SELECT. Однако это ограничение не является «непробиваемым», как сказано в документации, поскольку из таких функций все же могут быть вызваны изменчивые функции, способные модифицировать базу данных. В документации далее говорится, что если реализовать такую схему, то можно увидеть, что стабильные и постоянные функции не замечают изменений в базе данных, произведенных вызванной изменчивой функцией, поскольку такие изменения не проявляются в их снимке данных.

Задание. Проверьте описанный эффект практически, подобрав сначала абстрактный пример, а затем пример из предметной области авиаперевозок.

20 Не только багаж, но и питание

В подразделе 5.6.2 «Видимость изменений» (с. 338) мы рассматривали ситуацию, в которой в процессе регистрации билетов на рейс подсчитывались багажные места и их общий вес. Предположим, что с целью дальнейшего повышения качества обслуживания пассажиров наша авиакомпания решила опрашивать их перед полетом насчет предпочитаемого питания.

Задание 1. Для реализации поставленной задачи создайте дополнительную таблицу «Питание» (`flight_meals`) и соответствующим образом модифицируйте функции, выполняющие регистрацию билета.

Таблица может быть, например, такой:

```
CREATE TABLE flight_meals
( flight_id integer,
  boarding_no integer,
  main_course text NOT NULL
    CHECK ( main_course IN ( 'мясо', 'рыба', 'курица' ) ),
  PRIMARY KEY ( flight_id, boarding_no ),
  FOREIGN KEY ( flight_id, boarding_no )
    REFERENCES boarding_passes ( flight_id, boarding_no )
    ON DELETE CASCADE
);
CREATE TABLE
```

Теперь функция `boarding_info` должна выводить не только сведения о багаже, но также и общее число блюд каждого вида, выбранных пассажирами данного рейса.

Задание 2. Повторите ранее проведенные эксперименты, модифицировав запросы соответствующим образом.

21 Подведение итогов по операции бронирования: подстановка кода функции в запрос

В одной операции бронирования может быть оформлено несколько билетов, причем на разных пассажиров, а в каждом билете может присутствовать несколько перелетов. Было бы удобно иметь функцию, которая собирает всю информацию об операции бронирования примерно таким образом:

```
SELECT
  ticket_no,
  passenger_name,
  flight_no AS flight,
  departure_airport AS da,
  arrival_airport AS aa,
  scheduled_departure,
  amount
FROM get_booking_info( '000181' )
ORDER BY ticket_no, passenger_name, scheduled_departure;
```

```
   ticket_no   |  passenger_name  | flight | da  | aa  | scheduled_departure |  amount
---------------+------------------+--------+-----+-----+---------------------+----------
 0005435545944 | ALEKSANDR ZHUKOV | PG0517 | DME | SCW | 2017-08-28 10:30:00 | 10200.00
 0005435545944 | ALEKSANDR ZHUKOV | PG0570 | SCW | SVX | 2017-08-28 14:05:00 |  7900.00
 0005435545944 | ALEKSANDR ZHUKOV | PG0378 | SVX | NSK | 2017-09-07 13:40:00 | 19100.00
 0005435545944 | ALEKSANDR ZHUKOV | PG0125 | NSK | DME | 2017-09-08 18:45:00 | 28700.00
 0005435545945 | EVGENIYA KARPOVA | PG0517 | DME | SCW | 2017-08-28 10:30:00 | 10200.00
 0005435545945 | EVGENIYA KARPOVA | PG0570 | SCW | SVX | 2017-08-28 14:05:00 |  7900.00
 0005435545945 | EVGENIYA KARPOVA | PG0378 | SVX | NSK | 2017-09-07 13:40:00 | 19100.00
 0005435545945 | EVGENIYA KARPOVA | PG0125 | NSK | DME | 2017-09-08 18:45:00 | 28700.00
(8 строк)
```

Задание 1. Напишите предлагаемую функцию.

Указание. В коде функции, вероятно, будет использоваться представление «Рейсы» (`flights_v`). В силу этого планы запросов могут стать громоздкими. Для их упрощения можно создать таблицу на основе данного представления:

```
CREATE TABLE flights_vt AS
  SELECT * FROM flights_v;
```

Задание 2. Код такой функции в принципе может встраиваться в запрос, что в ряде случаев позволит ускорить его выполнение. Необходимые условия были рассмотрены в подразделе 5.7.1 «Подстановка кода функций в запрос» (с. 347). Выполните несколько запросов, использующих эту функцию, например:

```
SELECT gbi.*
FROM
  bookings AS b,
  get_booking_info( b.book_ref ) AS gbi
WHERE gbi.passenger_name = 'IVAN IVANOV'
ORDER BY gbi.ticket_no, gbi.passenger_name, gbi.scheduled_departure;
```

Обратите внимание, что в условии предложения WHERE фигурирует столбец, возвращаемый функцией. В ходе экспериментов организуйте выполнение запроса как с подстановкой кода функции в запрос, так и без нее. Сравните планы запросов и время их выполнения.

22 Подстановка в запрос кода скалярной функции

В подразделе 5.7.1 «Подстановка кода функций в запрос» (с. 347) была рассмотрена подстановка в запрос кода табличных функций, однако она возможна и для скалярных. Это такие функции, которые можно использовать в выражениях или предикатах, то есть там, где требуется обычное значение или условие.

Давайте возьмем в качестве примера функцию, выбирающую фамилию пассажира из его полного имени:

```
CREATE OR REPLACE FUNCTION get_lastname( fullname text )
RETURNS text AS
$$
  SELECT substr( fullname, strpos( fullname, ' ' ) + 1 );
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
```

Подсчитаем количество пассажиров с заданной фамилией:

```
EXPLAIN ( analyze, costs off )
SELECT count( * )
FROM tickets
WHERE get_lastname( passenger_name ) = 'NOVIKOV';
                                    QUERY PLAN
-----------------------------------------------------------------------------------------
 Aggregate (actual time=120.725..120.726 rows=1 loops=1)
   -> Seq Scan on tickets (actual time=0.085..120.570 rows=2278 loops=1)
        Filter: (substr(passenger_name, (strpos(passenger_name, ' '::text) + 1)) = 'NO...
        Rows Removed by Filter: 364455
 Planning Time: 0.090 ms
 Execution Time: 120.747 ms
(6 строк)
```

Обратите внимание: код скалярной функции оказался подставленным в текст запроса. Чтобы такая подстановка стала возможной, требуется выполнить ряд условий, описанных в документе «Inlining of SQL functions» (wiki.postgresql.org/

wiki/Inlining_of_SQL_functions). В частности, функция должна быть написана на языке SQL, она должна возвращать один столбец, тело функции должно состоять из единственной команды SELECT *выражение*. В этой команде не допускаются другие предложения, такие как FROM, WHERE, GROUP BY и т. д. Возвращаемое значение функции не должно быть определено как RETURNS RECORD, RETURNS SETOF или RETURNS TABLE.

А какой выигрыш в скорости выполнения запроса мы получили при подстановке в него кода функции? Для ответа на вопрос нужно каким-то образом запретить подстановку кода.

У функций есть еще одна характеристика, которую мы не рассматривали, поскольку она выходит за рамки книги. Эта характеристика связана с привилегиями доступа к объектам базы данных (см. описание команды CREATE FUNCTION в документации) и по умолчанию имеет значение SECURITY INVOKER: функция будет исполняться с привилегиями пользователя, который ее вызвал. Такое значение не препятствует встраиванию кода функции в запрос. Однако если изменить его на SECURITY DEFINER (исполнение функции с привилегиями создавшего ее пользователя), встраивание станет невозможным. Воспользуемся этим и назначим функции get_lastname характеристику SECURITY DEFINER, чтобы запретить планировщику встраивать ее код в запрос:

```
ALTER FUNCTION get_lastname SECURITY DEFINER;
ALTER FUNCTION
```

Повторим запрос:

```
EXPLAIN ( analyze, costs off )
SELECT count( * )
FROM tickets
WHERE get_lastname( passenger_name ) = 'NOVIKOV';
                              QUERY PLAN
-----------------------------------------------------------------------
 Aggregate (actual time=551.215..551.216 rows=1 loops=1)
   -> Seq Scan on tickets (actual time=0.601..551.014 rows=2278 loops=1)
        Filter: (get_lastname(passenger_name) = 'NOVIKOV'::text)
        Rows Removed by Filter: 364455
 Planning Time: 0.049 ms
 Execution Time: 551.235 ms
(6 строк)
```

Теперь подстановка кода не выполнялась, и времени для выполнения запроса потребовалось значительно больше.

Перед выполнением заданий верните исходное значение этой характеристики:

```
ALTER FUNCTION get_lastname SECURITY INVOKER;
ALTER FUNCTION
```

Задание 1. Для отбора строк использовался последовательный просмотр. А если мы создадим индекс по столбцу passenger_name, будет ли он использоваться при выполнении запроса? Ведь в коде функции этот столбец присутствует только в качестве параметра функций substr и strpos.

```
CREATE INDEX ON tickets ( passenger_name );
CREATE INDEX
```

Сначала сделайте обоснованное предположение, а потом проведите эксперимент с обеими характеристиками SECURITY DEFINER и SECURITY INVOKER функции get_lastname. Сравните скорости выполнения запроса в этих двух случаях.

Указание. Имейте в виду, что индекс рассматривается планировщиком, только если в условии выражение по одну сторону оператора в точности соответствует выражению, по которому построен индекс. Однако индексный доступ может иногда применяться, даже если в запросе нет подходящих для него условий. В этом случае индекс сканируется полностью, а все условия из предложения WHERE (если они указаны) перепроверяются для каждой строки выборки. Это может оказаться эффективнее полного просмотра таблицы, особенно если индекс имеет небольшие размеры и содержит все необходимые запросу значения. Полезную информацию можно найти в разделе документации 11.9 «Сканирование только индекса и покрывающие индексы».

Задание 2. Создайте перегруженную функцию, имеющую два параметра и другое возвращаемое значение:

```
CREATE OR REPLACE FUNCTION get_lastname( fullname text, lastname text )
RETURNS bool AS
$$
  SELECT substr( fullname, strpos( fullname, ' ' ) + 1 ) = lastname;
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
```

Соответствующим образом измените условие в предложении WHERE:

```
...
WHERE get_lastname( passenger_name, 'NOVIKOV' );
```

Повторите все эксперименты. Будет ли сейчас подставляться код функции в запрос и использоваться индекс по столбцу passenger_name?

23 Пользовательские функции в индексных выражениях

В предыдущем упражнении мы создавали индекс по столбцу passenger_name. В разделе документации 11.7 «Индексы по выражениям» сказано, что в индексных выражениях могут присутствовать и функции. Важно, что они должны иметь категорию изменчивости IMMUTABLE, как говорится в описании команды CREATE INDEX, приведенном в документации.

Создадим индекс по первой из перегруженных функций get_lastname, предложенных в предыдущем упражнении. Если индекс по столбцу passenger_name не сохранился, создадим и его.

```
CREATE INDEX ON tickets ( get_lastname( passenger_name ) );
CREATE INDEX
CREATE INDEX IF NOT EXISTS tickets_passenger_name_idx ON tickets ( passenger_name );
CREATE INDEX
```

Повторим уже известный запрос:

```
EXPLAIN ( analyze, costs off )
SELECT count( * )
FROM tickets
WHERE get_lastname( passenger_name ) = 'NOVIKOV';
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Aggregate (actual time=1.037..1.038 rows=1 loops=1)
   -> Bitmap Heap Scan on tickets (actual time=0.480..0.941 rows=2278 loops=1)
        Recheck Cond: (substr(passenger_name, (strpos(passenger_name, ' '::text) + 1))...
        Heap Blocks: exact=1897
        -> Bitmap Index Scan on tickets_get_lastname_idx (actual time=0.234..0.234 row...
             Index Cond: (substr(passenger_name, (strpos(passenger_name, ' '::text) + 1...
 Planning Time: 0.231 ms
 Execution Time: 1.060 ms
(8 строк)
```

Время выполнения запроса сократилось примерно на два порядка по сравнению с предыдущим упражнением. Обратите внимание, какой из индексов выбирает планировщик.

А если искомой фамилии нет в таблице «Билеты» (tickets), то подходящий индекс позволяет вовсе избежать обращений к таблице (в плане теперь нет строки с меткой Heap Blocks):

```
EXPLAIN ( analyze, costs off )
SELECT count( * )
FROM tickets
WHERE get_lastname( passenger_name ) = 'LEVITAN';
                                      QUERY PLAN
-----------------------------------------------------------------------------------------
 Aggregate (actual time=0.039..0.040 rows=1 loops=1)
   -> Bitmap Heap Scan on tickets (actual time=0.037..0.037 rows=0 loops=1)
       Recheck Cond: (substr(passenger_name, (strpos(passenger_name, ' '::text) + 1))...
       -> Bitmap Index Scan on tickets_get_lastname_idx (actual time=0.034..0.034 row...
           Index Cond: (substr(passenger_name, (strpos(passenger_name, ' '::text) + 1...
 Planning Time: 0.094 ms
 Execution Time: 0.059 ms
(7 строк)
```

Задание 1. В дополнение к двум предыдущим индексам создайте еще один по выражению, которое фигурирует в коде функции get_lastname:

```
CREATE INDEX ON tickets ( substr( passenger_name, strpos( passenger_name, ' ' ) + 1 ) );
CREATE INDEX
```

Проведите ряд экспериментов, изменяя условие в предложении WHERE и выражение в списке SELECT, например:

```
...
WHERE get_lastname( passenger_name ) = 'NOVIKOV';
...
WHERE substr( passenger_name, ( strpos( passenger_name, ' '::text ) + 1 ) ) = 'NOVIKOV';
...
SELECT get_lastname( passenger_name )
...
SELECT passenger_name
...
SELECT ticket_no
```

Удаляйте индексы по одному и смотрите, какие из оставшихся используются в запросах.

Задание 2. Опять создайте все три индекса, если вы их удалили. Модифицируйте код функции get_lastname таким образом, чтобы она выбирала не фамилию, а имя из полного имени пассажира:

```
CREATE OR REPLACE FUNCTION get_lastname( fullname text )
  RETURNS text AS
$$
  SELECT left( fullname, strpos( fullname, ' ' ) - 1 );
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
```

Не пересоздавая индекс, созданный с использованием этой функции, выполните тот же самый запрос, но вместо искомой фамилии задайте имя:

```
EXPLAIN ( analyze, costs off )
SELECT count( * )
FROM tickets
WHERE get_lastname( passenger_name ) = 'DARYA';
```

Посмотрите в плане, используется ли теперь индекс и какой именно. Можно ли считать модифицирование кода функции, по которой уже создан индекс, корректным приемом?

Задание 3. Создайте функцию:

```
CREATE OR REPLACE FUNCTION seats_count( a_code text )
RETURNS integer AS
$$
  SELECT count( * )
  FROM seats
  WHERE aircraft_code = a_code;
$$ LANGUAGE sql STABLE;
CREATE FUNCTION
```

Ее можно вызвать в запросе:

```
SELECT *
FROM
  aircrafts,
  seats_count( aircraft_code ) AS seats_count
ORDER BY seats_count DESC;
```

```
 aircraft_code |        model         | range | seats_count
---------------+----------------------+-------+-------------
 773           | Боинг 777-300        | 11100 |         402
 763           | Боинг 767-300        |  7900 |         222
 321           | Аэробус A321-200     |  5600 |         170
 320           | Аэробус A320-200     |  5700 |         140
 733           | Боинг 737-300        |  4200 |         130
 319           | Аэробус A319-100     |  6700 |         116
 SU9           | Сухой Суперджет-100  |  3000 |          97
 CR2           | Бомбардье CRJ-200    |  2700 |          50
 CN1           | Сессна 208 Караван   |  1200 |          12
(9 строк)
```

А можно ли создать индекс по этой функции для таблицы aircrafts_data, лежащей в основе представления «Самолеты» (aircrafts)? Сначала сделайте обоснованное предположение, а затем проверьте его практически.

24 Иллюстрация использования системного каталога pg_depend

В разделе 5.2 «Функции и зависимости между объектами базы данных» (с. 288) мы уже обращались к системному каталогу pg_depend. Давайте сделаем это еще раз. Для экспериментов обратимся к перегруженным функциям get_lastname, возвращающим фамилию пассажира. Они были созданы в упражнении 22 (с. 408).

Посмотрим, что записано о наших функциях в системном каталоге pg_proc (конечно, в выборку вошла лишь малая часть сведений). Воспользуемся типом regprocedure, представленным в разделе документации 8.19 «Идентификаторы объектов». Он позволяет вывести имя функции вместе с типами данных ее параметров.

```
SELECT oid, proname, proargtypes, proargtypes::regtype[], oid::regprocedure
FROM pg_proc
WHERE proname = 'get_lastname';
```

```
  oid  |   proname    | proargtypes |    proargtypes    |           oid
-------+--------------+-------------+-------------------+-------------------------
 16684 | get_lastname | 25          | [0:0]={text}      | get_lastname(text)
 16686 | get_lastname | 25 25       | [0:1]={text,text} | get_lastname(text,text)
(2 строки)
```

Создадим индекс на таблице «Билеты» (tickets) по одной из этих функций:

```
CREATE INDEX tickets_func_idx ON tickets ( get_lastname( passenger_name ) );
CREATE INDEX
```

Зная имя индекса, выберем из системного каталога pg_depend все строки, описывающие зависимости этого индекса от других объектов базы данных:

```
SELECT
  classid::regclass AS classname,
  objid::regclass AS objname,
  refclassid::regclass AS refclassname,
  refobjid::regclass AS refobjname,
  ( SELECT attname
    FROM pg_attribute
    WHERE attrelid = refobjid
      AND attnum = refobjsubid
  ),
  CASE deptype
    WHEN 'n' THEN 'normal'
    WHEN 'a' THEN 'auto'
    ELSE 'other'
  END AS deptype
FROM pg_depend
WHERE objid::regclass::text = 'tickets_func_idx';
```

Для столбца refobjid мы использовали приведение типа refobjid::regclass. Оно не работает для объекта, являющегося функцией.

В столбце attname выборки показано имя столбца таблицы tickets, от которой зависит наш индекс. Обратите внимание, что хотя индекс был создан на основе функции, здесь указывается столбец, который передавался ей в качестве аргумента в команде создания индекса. Номер этого столбца хранится в столбце refobjsubid, а для вывода его имени мы воспользовались подзапросом к системному каталогу pg_attribute. Он представлен в разделе документации 51.7 «pg_attribute».

```
 classname |     objname      | refclassname | refobjname |    attname     | deptype
-----------+------------------+--------------+------------+----------------+---------
 pg_class  | tickets_func_idx | pg_class     | tickets    |                | auto
 pg_class  | tickets_func_idx | pg_class     | tickets    | passenger_name | auto
 pg_class  | tickets_func_idx | pg_proc      | 16684      |                | normal
(3 строки)
```

Чтобы вместо OID функции get_lastname вывести ее имя, нужно в списке SELECT запроса заменить тип regclass на regproc в операции приведения типа столбца refobjid, а в предложение WHERE добавить условие, сужающее выборку.

```
SELECT
  classid::regclass AS classname,
  objid::regclass AS objname,
  refclassid::regclass AS refclassname,
  refobjid::regproc AS refobjname,
  ( SELECT attname
    FROM pg_attribute
    WHERE attrelid = refobjid
      AND attnum = refobjsubid
  ),
  CASE deptype
    WHEN 'n' THEN 'normal'
    WHEN 'a' THEN 'auto'
    ELSE 'other'
  END AS deptype
FROM pg_depend
WHERE objid::regclass::text = 'tickets_func_idx'
  AND refclassid::regclass::text = 'pg_proc' \gx
-[ RECORD 1 ]+----------------------
classname    | pg_class
objname      | tickets_func_idx
refclassname | pg_proc
refobjname   | bookings.get_lastname
attname      |
deptype      | normal
```

Задание 1. Модифицируйте первый запрос к системному каталогу pg_depend таким образом, чтобы он выводил не OID функции get_lastname, а ее имя.

Задание 2. Модифицируйте запросы к системному каталогу pg_depend, воспользовавшись стандартной функцией pg_describe_object, представленной в подразделе документации 9.27.5 «Функции получения информации и адресации объектов». Эта функция выводит текстовое описание объекта базы данных.

Задание 3. Обратитесь к системному каталогу pg_index (см. раздел документации 51.26 «pg_index»). В столбце indexprs хранятся сведения об атрибутах индекса, не являющихся простыми ссылками на столбцы. Эти сведения представлены в виде деревьев специальной структуры. Здесь можно увидеть и OID функции, на основе которой построен индекс.

25 Подстановка кода функций в запрос: детальные эксперименты

В подразделе 5.7.1 «Подстановка кода функций в запрос» (с. 347) была рассмотрена функция count_passengers, подсчитывающая пассажиров, перевезенных по каждому маршруту за весь период времени, представленный в базе данных. Давайте создадим ее расширенную версию, позволяющую вывести не только номера рейсов, но и даты их выполнения:

```
CREATE OR REPLACE FUNCTION count_passengers_2(
  OUT sched_dep date,
  OUT f_no char,
  OUT pass_num bigint
) RETURNS SETOF record AS
$$
  SELECT scheduled_departure::date, flight_no, count( * )
  FROM flights AS f
    JOIN boarding_passes AS bp ON bp.flight_id = f.flight_id
  WHERE status IN ( 'Departed', 'Arrived' )
  GROUP BY scheduled_departure::date, flight_no;
$$ LANGUAGE sql STABLE;
```

```
CREATE FUNCTION
```

Задание. Проведите эксперименты, аналогичные тем, что были показаны в тексте вышеупомянутого раздела. Например, выполните такой запрос:

```
SELECT sched_dep, f_no, pass_num
FROM count_passengers_2()
WHERE f_no IN ( 'PG0149', 'PG0148' )
  AND sched_dep > '2017-08-05'::date
  AND sched_dep < '2017-08-10'::date
ORDER BY sched_dep, f_no;
```

```
 sched_dep  |  f_no  | pass_num
------------+--------+----------
 2017-08-06 | PG0148 |       50
 2017-08-06 | PG0149 |       45
 2017-08-07 | PG0148 |       49
 2017-08-07 | PG0149 |       47
 2017-08-08 | PG0148 |       42
 2017-08-08 | PG0149 |       47
 2017-08-09 | PG0148 |       39
 2017-08-09 | PG0149 |       49
(8 строк)
```

Посмотрите план запроса. В каком узле плана оказывается условие из предложения WHERE запроса?

Проводя эксперименты, изменяйте условие так, чтобы даты отправления ограничивались не только двумя неравенствами, но и одним, а также равенством. Обращайте внимание на время выполнения запросов в сравнении с выполнением полной выборки рейсов, когда в запросе есть только вызов функции без дополнительных условий в предложении WHERE.

Попробуйте задавать более сложные условия, например:

```
SELECT sched_dep, f_no, pass_num
FROM count_passengers_2()
WHERE ( f_no = 'PG0149' AND sched_dep > '2017-08-05'::date
      )
   OR ( f_no = 'PG0148' AND sched_dep < '2017-08-10'::date
      )
ORDER BY sched_dep, f_no;
```

26 Как отобрать из Парето-оптимального множества единственную альтернативу?

Из теории принятия решений известно, что формирование множества Парето, рассмотренное в тексте главы, является лишь первым этапом выбора из множества альтернатив. Для определения единственной оптимальной альтернативы из оставшихся используются различные способы:

Указание нижних (верхних) границ критериев. Для позитивных критериев задаются нижние границы, а для негативных — верхние. Все Парето-оптимальные альтернативы подвергаются проверке на соответствие этим границам. Чем более жесткими будут границы, тем меньшее число альтернатив будет им удовлетворять. Стремятся отобрать единственную альтернативу.

Субоптимизация. Выделяется один из критериев в качестве главного, а по всем остальным назначают нижние (верхние) границы. Оптимальной считается альтернатива, имеющая максимальное (для негативного критерия — минимальное) значение выделенного критерия среди всех альтернатив, удовлетворяющих назначенным границам. Фактически задача многокритериальной оптимизации превращается в задачу скалярной оптимизации на суженном допустимом множестве.

Лексикографическая оптимизация. Критерии упорядочиваются по их относительной важности. На первом шаге отбираются альтернативы, имеющие максимальную оценку по важнейшему критерию. Если такая альтернатива всего одна, то ее и считают оптимальной; если же их несколько, то из них отбираются те, которые имеют максимальную оценку по следующему по важности критерию, и т. д. Оставшаяся альтернатива будет оптимальной. При этом подходе слишком преувеличивается роль первого по важности критерия: если по нему отбирается всего одна альтернатива, то остальные критерии вообще не учитываются.

Конечно, окончательное решение при использовании этих методов имеет субъективный характер, так как относительную важность критериев и значения границ задает лицо, принимающее решение.

Задание. Напишите функции, реализующие представленные методы.

27 Функция pareto: результат попарного сравнения альтернатив в другой форме

В разделе 5.8 «Элементы теории принятия решений» (с. 357) была представлена функция `compare_pairwise` для попарного сравнения альтернатив. Она использовалась в коде функции `pareto`, формирующей множество Парето-оптимальных альтернатив. Результат сравнения представлял собой массив из двух элементов. В том же разделе была предложена и другая версия этой функции — `compare_pairwise_2`, — возвращающая целое число.

Задание. Модифицируйте функцию `pareto`, заменив в ее коде вызов функции `compare_pairwise` на вызов `compare_pairwise_2`, и проведите эксперименты из упомянутого раздела главы.

28 Составные значения в качестве аргументов функций

В разделе 5.8 «Элементы теории принятия решений» (с. 357) была представлена функция `compare_pairwise` для попарного сравнения альтернатив. Ее параметрами являются составные значения типа `meal`. Давайте проведем эксперимент, в котором функция будет получать один из аргументов как результат выполнения подзапроса. В качестве базовой альтернативы возьмем альтернативу *D*.

```
SELECT *
FROM
  meal AS m,
  compare_pairwise( ( SELECT meal FROM meal WHERE meal_code = 'D' ), m ) AS score
WHERE m.meal_code <> 'D'
ORDER BY m.meal_code;
```

```
 meal_code | price  | calories | variety | pareto_optimal | score
-----------+--------+----------+---------+----------------+-------
 A         | 550.00 |     1500 |       3 | t              | {2,1}
 B         | 490.00 |     1300 |       4 | t              | {2,1}
 C         | 600.00 |     1400 |       4 | f              | {3,0}
 E         | 570.00 |     1380 |       5 | f              | {1,1}
 F         | 520.00 |     1450 |       3 | f              | {2,1}
 G         | 580.00 |     1580 |       4 | f              | {2,0}
 H         | 570.00 |     1380 |       4 | f              | {2,1}
 I         | 510.00 |     1450 |       3 | t              | {2,1}
 J         | 530.00 |     1450 |       6 | t              | {1,2}
(9 строк)
```

Обратите внимание: поскольку функция compare_pairwise ожидает в качестве аргументов значения составного типа meal, то в списке SELECT подзапроса присутствует именно значение meal, а не meal.* или просто *. Если подставить meal.*, будет выведена ошибка, ведь подзапрос возвратит четыре столбца:

```
SELECT *
FROM
  meal AS m,
  compare_pairwise( ( SELECT meal.* FROM meal WHERE meal_code = 'D' ), m ) AS score
WHERE m.meal_code <> 'D'
ORDER BY m.meal_code;
```

```
ОШИБКА:  подзапрос должен вернуть только один столбец
СТРОКА 4:   compare_pairwise( ( SELECT meal.* FROM meal WHERE meal_cod...
                              ^
```

Здесь речь идет о столбце составного типа meal, а не об одном из столбцов meal_code, price, calories, variety.

Однако при непосредственной передаче аргумента (не используя подзапрос) в вызове функции можно написать как m, так и m.*. Это объясняется тем, что запись вида m.* приводит к развертыванию составного значения в виде множества столбцов, когда она фигурирует на верхнем уровне в списке SELECT и в ряде других ситуаций. Однако такого развертывания не происходит при использовании m.* в качестве аргумента функции. Подробно об этом можно прочитать в подразделе документации 8.16.5 «Использование составных типов

в запросах». Там же сказано, что использование записи вида m.* считается более правильным стилем, поскольку в таком случае синтаксический анализатор воспримет идентификатор m как ссылку на имя или псевдоним таблицы, а не на имя столбца, что избавляет от неоднозначности.

Задание. Воспроизведите ситуации, описанные в упомянутом разделе документации, в которых происходит развертывание составного значения в виде множества столбцов.

29 ROWS — характеристика количества строк, возвращаемых функцией

Планировщик строит план выполнения запроса, исходя из предположений о количестве строк, отбираемых в каждом узле плана. Если оценки будут неточными, то и план получится неоптимальным. Функции, возвращающие множество строк, усложняют планирование. Функция, код которой не встроен в запрос, — черный ящик для планировщика, но он обязан как-то оценить число строк, возвращаемых ею. По умолчанию он считает это число равным 1000. К сожалению, не всегда этот выбор оказывается наилучшим. Но у программиста есть возможность помочь планировщику, задавая для функции характеристику ROWS.

Предположим, что служба контроля качества нашей авиакомпании выбирает несколько рейсов и тщательно проверяет, что экипаж следует всем инструкциям при подготовке к полету. Нам нужна функция, которая сформирует список рейсов случайным образом. Она будет получать один параметр — количество отбираемых рейсов. Это не только сделает функцию более гибкой, но и позволит проводить эксперименты с нею без модификации исходного текста.

Воспользуемся возможностью функции random формировать целое число в заданном диапазоне:

```
CREATE OR REPLACE FUNCTION generate_flight_ids( ids_count integer )
RETURNS TABLE ( id integer ) AS
$$
  SELECT random( 1, ( SELECT max( flight_id ) FROM flights ) )
  FROM generate_series( 1, ids_count );
$$ LANGUAGE sql;
CREATE FUNCTION
```

Давайте сформируем 100 идентификаторов рейсов:

```
SELECT
  f.flight_id,
  f.flight_no,
  f.scheduled_departure AS dep,
  f.departure_airport AS da,
  f.arrival_airport AS aa
FROM flights AS f
  JOIN generate_flight_ids( 100 ) AS gfi ON f.flight_id = gfi.id
ORDER BY dep;
```

```
 flight_id | flight_no |          dep           | da  | aa
-----------+-----------+------------------------+-----+-----
     14036 | PG0626    | 2017-07-17 18:35:00+10 | KJA | UIK
     30944 | PG0602    | 2017-07-17 18:50:00+10 | AER | PEZ
 ...
     12073 | PG0274    | 2017-09-13 22:50:00+10 | CEK | DME
      4678 | PG0235    | 2017-09-14 02:50:00+10 | VKO | BZK
(100 строк)
```

Обратите внимание на узел плана Function Scan. Фактическое значение показателя rows здесь равно 100, что вполне объяснимо: ведь мы просили функцию generate_flight_ids сформировать 100 строк. А вот плановое значение равно 1000, то есть принято по умолчанию.

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Sort  (cost=1199.93..1202.43 rows=1000 width=27)
   (actual time=12.424..12.429 rows=100 loops=1)
   Sort Key: f.scheduled_departure
   Sort Method: quicksort  Memory: 29kB
   -> Hash Join  (cost=1137.47..1150.10 rows=1000 width=27)
       (actual time=12.364..12.401 rows=100 loops=1)
       Hash Cond: (gfi.id = f.flight_id)
       -> Function Scan on generate_flight_ids gfi
           (cost=0.25..10.25 rows=1000 width=4)
           (actual time=0.149..0.154 rows=100 loops=1)
       -> Hash  (cost=723.21..723.21 rows=33121 width=27)
           (actual time=12.172..12.173 rows=33121 loops=1)
           Buckets: 65536  Batches: 1  Memory Usage: 2583kB
           -> Seq Scan on flights f  (cost=0.00..723.21 rows=33121 width=27)
               (actual time=0.009..6.743 rows=33121 loops=1)
 Planning Time: 0.102 ms
 Execution Time: 12.460 ms
(17 строк)
```

В этом плане для соединения наборов строк используется метод хеширования (Hash Join). Формирование хеш-таблицы на основе таблицы flights потребовало много времени, но число обращений к ней будет невелико. Окупаются ли в процессе соединения наборов строк затраты на ее создание? Это трудно сказать без проведения дополнительных экспериментов.

Пусть планировщик узнает о том, что функция возвращает 100 строк, а не 1000:

```
ALTER FUNCTION generate_flight_ids ROWS 100;
ALTER FUNCTION
```

Увидеть текущее значение характеристики ROWS можно в системном каталоге pg_proc. Его описание приведено в разделе документации 51.39 «pg_proc».

```
SELECT proname, prorows
FROM pg_proc
WHERE proname = 'generate_flight_ids';
       proname       | prorows
---------------------+---------
 generate_flight_ids |     100
(1 строка)
```

Снова сформируем 100 номеров и посмотрим, что изменилось в плане запроса. Теперь не только фактическое, но и плановое значение показателя rows в узле Function Scan стало равным 100. Но это не самое интересное. Важнее то, что изменилась структура плана, оценки и — самое главное — выполнение ускорилось примерно на порядок. Планировщик решил отказаться от соединения хешированием и выбрал вложенные циклы (Nested Loop):

```
                                    QUERY PLAN
-------------------------------------------------------------------------------------------
 Sort  (cost=655.32..655.57 rows=100 width=27) (actual time=0.511..0.515 rows=100 loo...
   Sort Key: f.scheduled_departure
   Sort Method: quicksort  Memory: 29kB
   -> Nested Loop  (cost=0.54..652.00 rows=100 width=27)
      (actual time=0.186..0.482 rows=100 loops=1)
      -> Function Scan on generate_flight_ids gfi  (cost=0.25..1.25 rows=100 width=4)
         (actual time=0.178..0.184 rows=100 loops=1)
      -> Index Scan using flights_pkey on flights f
         (cost=0.29..6.51 rows=1 width=27) (actual time=0.003..0.003 rows=1 loops=100)
         Index Cond: (flight_id = gfi.id)
 Planning Time: 0.099 ms
 Execution Time: 0.536 ms
(12 строк)
```

Конечно, при однократном выполнении запроса ускорение — даже на порядок — может оказаться не таким важным, как при частом его выполнении. Но тем не менее положительный эффект, полученный путем простого указания характеристики функции ROWS, оказался существенным.

Обратите внимание на разность второй и первой оценок cost в узле Function Scan. Она равна произведению параметра cpu_tuple_cost (по умолчанию — 0,01) на значение планового показателя rows. Узнать о настройках планировщика можно из раздела документации 19.7 «Планирование запросов».

В разделе документации 36.11 «Информация для оптимизации функций» говорится в том числе о *вспомогательных функциях для планировщика* (planner support functions). Такую функцию можно связать с целевой SQL-функцией. Через вспомогательную функцию планировщику можно передать ту информацию о целевой функции, которая слишком сложна для представления в декларативном виде, в частности в виде характеристики ROWS. Вспомогательная функция должна быть написана на языке C, при этом язык целевой функции может быть любым. Для целевой функции, возвращающей множество строк, вспомогательная функция позволяет получать переменную оценку числа выдаваемых строк.

Давайте посмотрим, какие вспомогательные функции связаны, скажем, с перегруженной стандартной функцией generate_series:

```
SELECT proname, prosupport, proargtypes::regtype[], prorettype::regtype
FROM pg_proc
WHERE proname = 'generate_series' \gx
-[ RECORD 1 ]----------------------------
proname     | generate_series
prosupport  | generate_series_int4_support
proargtypes | [0:2]={integer,integer,integer}
prorettype  | integer
-[ RECORD 2 ]----------------------------
proname     | generate_series
prosupport  | generate_series_int4_support
proargtypes | [0:1]={integer,integer}
prorettype  | integer
 ...
-[ RECORD 5 ]----------------------------
proname     | generate_series
prosupport  | -
proargtypes | [0:2]={numeric,numeric,numeric}
prorettype  | numeric
 ...
```

Попутно заметим, что порядок индексирования элементов в массивах, принятый по умолчанию, можно изменить, что и сделано в столбце proargtypes.

Задание 1. Проведите этот же эксперимент, формируя 1000 и 10 000 идентификаторов рейсов. Следите за тем, чтобы характеристика ROWS и параметр ids_count функции generate_flight_ids имели одинаковые значения. Для того чтобы убедиться, что планировщик выбирает оптимальный план, «предложите» ему временно отказаться сначала от способа соединения строк хешированием, а затем, посмотрев полученный план, и от способа соединения слиянием. Команды, которые действуют в пределах текущего сеанса подключения к серверу, таковы:

```
SET enable_hashjoin = off;
SET enable_mergejoin = off;
```

Таким образом, вы увидите, что в ряде ситуаций достаточно лишь правильно задать значение ROWS, чтобы планировщик выбрал наилучший способ соединения наборов строк в плане, а в других случаях может потребоваться использовать параметры *enable_hashjoin* и *enable_mergejoin*.

Для возврата значений параметров *enable_hashjoin* и *enable_mergejoin* в исходное состояние можно воспользоваться командой RESET.

Посмотреть текущие установки этих параметров позволит команда SHOW. Например:

```
SHOW enable_hashjoin;
 enable_hashjoin
-----------------
 on
(1 строка)
```

Задание 2. Конечно, характеристика ROWS полезна, только если число строк, возвращаемых функцией, известно заранее и остается примерно постоянным при многократных вызовах. Но так бывает далеко не всегда. Тем не менее подумайте, имеет ли смысл задать значение ROWS для функций, рассмотренных в этой главе. Проведите необходимые эксперименты, аналогичные тем, что были выполнены в этом упражнении.

Задание 3. Выясните, для каких стандартных функций PostgreSQL определены вспомогательные функции для планировщика.

30 COST — характеристика стоимости вычисления функции

Существует еще одна характеристика, которую можно задать для функции и настроить таким образом план запроса — COST. Эта характеристика задает примерную стоимость выполнения функции в единицах *cpu_operator_cost*. Точнее говоря, первая оценка cost в узле, в котором вызывается функция, равна произведению значения COST на значение параметра планировщика *cpu_operator_cost*, которое по умолчанию равно 0,0025. Значение COST для функций, написанных не на языке C, по умолчанию составляет 100.

Задание. Самостоятельно ознакомьтесь с характеристикой COST, проведя необходимые эксперименты. Воспользуйтесь функцией generate_flight_ids и запросом из предыдущего упражнения.

Если количество возвращаемых функцией строк более или менее стабильно и заранее известно, можно попытаться ускорить выполнение запроса, изменяя эту характеристику. Конечно, если количество строк при многократных вызовах количества может отличаться, скажем, на порядок, то трудно выбрать обоснованное значение характеристики COST. В таком случае может помочь создание вспомогательной функции для планировщика. О них речь идет в разделе документации 36.11 «Информация для оптимизации функций».

Указание. Информацию о характеристике COST можно найти в описании команды CREATE FUNCTION, а подробные сведения о настройках планировщика представлены в разделе документации 19.7 «Планирование запросов».

Увидеть текущее значение характеристики COST можно в системном каталоге pg_proc:

```
SELECT proname, procost
FROM pg_proc
WHERE proname = 'generate_flight_ids';
       proname       | procost
---------------------+---------
 generate_flight_ids |     100
(1 строка)
```

Изменить это значение можно командой ALTER FUNCTION. Например:

```
ALTER FUNCTION generate_flight_ids COST 1000;
ALTER FUNCTION
```

31 Вычисляемые столбцы, сохраняемые в таблице

В тексте главы было рассмотрено множество примеров использования функций в запросах. Однако этим их применение не исчерпывается. С помощью постоянных (IMMUTABLE) функций можно определить вычисляемый (генерируемый) столбец при создании или модификации таблиц. Подробно о вычисляемых столбцах можно прочитать в описании команды CREATE TABLE, приведенном в документации.

Давайте сделаем копию таблицы «Билеты» (tickets). Во временной таблице-копии не будем создавать первичный и внешний ключи, поскольку это не влияет на наши эксперименты.

```
CREATE TEMP TABLE tickets_2 AS
SELECT * FROM tickets;
SELECT 366733
```

Предположим, что при эксплуатации базы данных часто выполняются запросы, в которых имя и фамилия пассажира выводятся в виде отдельных значений. Если бы эти значения хранились в отдельных столбцах, такие запросы выполнялись бы значительно быстрее (читатель может проверить это самостоятельно). Время, затрачиваемое на заполнение этих столбцов при вводе данных, многократно компенсировалось бы при выборках большого числа строк. Давайте напишем две функции, которые будут извлекать эти элементы данных из единого значения. Эти функции будут автоматически вызываться как при создании вычисляемых столбцов, так и при добавлении и обновлении табличных строк.

Напомним, что в столбце passenger_name таблицы «Билеты» (tickets) имя и фамилия пассажира хранятся именно в таком порядке: сначала — имя, а потом — фамилия.

```
CREATE OR REPLACE FUNCTION get_first_name( pass_name text )
RETURNS text AS
$$
  SELECT left( pass_name, strpos( pass_name, ' ' ) - 1 );
$$
LANGUAGE sql IMMUTABLE;
```

```
CREATE FUNCTION
```

```
CREATE OR REPLACE FUNCTION get_last_name( pass_name text )
  RETURNS text AS
$$
SELECT substr( pass_name, strpos( pass_name, ' ' ) + 1 );
$$
LANGUAGE sql IMMUTABLE;
```

```
CREATE FUNCTION
```

Добавим в таблицу вычисляемые столбцы для хранения имени и фамилии. Для этого воспользуемся предложением GENERATED ALWAYS. Во включенном в него выражении для вычисления значений столбца допускаются вызовы функций и ссылки на столбцы этой же таблицы, но на другие таблицы ссылаться нельзя.

В вычисляемый столбец нельзя записать значение явным образом (задав его, скажем, в виде константы), а при чтении этого столбца выводится результат вычисления выражения, которое и включено в предложение GENERATED ALWAYS. Ключевое слово STORED означает, что значение столбца будет вычисляться при записи и сохраняться на диске.

```
ALTER TABLE tickets_2
ADD COLUMN passenger_fname text GENERATED ALWAYS AS
  ( get_first_name( passenger_name ) ) STORED;
```

```
ALTER TABLE
```

```
ALTER TABLE tickets_2
ADD COLUMN passenger_lname text GENERATED ALWAYS AS
  ( get_last_name( passenger_name ) ) STORED;
```

```
ALTER TABLE
```

При добавлении столбцов в таблицу должны быть вычислены их новые значения. Да, новые столбцы содержат значения:

```
SELECT ticket_no, passenger_name, passenger_fname, passenger_lname
FROM tickets_2
LIMIT 3;
```

```
   ticket_no   |   passenger_name   | passenger_fname | passenger_lname
---------------+--------------------+-----------------+-----------------
 0005432000987 | VALERIY TIKHONOV   | VALERIY         | TIKHONOV
 0005432000988 | EVGENIYA ALEKSEEVA | EVGENIYA        | ALEKSEEVA
 0005432000989 | ARTUR GERASIMOV    | ARTUR           | GERASIMOV
(3 строки)
```

Попробуем вставить новую строку. Важно помнить, что задавать явно значения для вычисляемых столбцов нельзя: их вычислят соответствующие функции:

```
INSERT INTO tickets_2 VALUES
  ( '1234567890123', '123ABC', '1234 123456', 'ANTON GOROSHKIN',
    '{"phone": "+71234567890"}'
  );
INSERT 0 1
```

Все в порядке, при вставке новой строки имя и фамилия записываются в соответствующие столбцы автоматически — функции работают:

```
SELECT ticket_no, passenger_name, passenger_fname, passenger_lname
FROM tickets_2
WHERE passenger_lname = 'GOROSHKIN';
   ticket_no   | passenger_name  | passenger_fname | passenger_lname
---------------+-----------------+-----------------+-----------------
 1234567890123 | ANTON GOROSHKIN | ANTON           | GOROSHKIN
(1 строка)
```

Вопрос. Мы назначили функциям категорию изменчивости IMMUTABLE, потому что так требует документация. Эти функции действительно соответствуют этой категории? Почему?

Задание. Предложенные функции — очень простые, и тем не менее создание вычисляемых столбцов на их основе дает выигрыш по времени при выборках с участием имени и фамилии пассажира. Проверить это можно, например, с помощью таких запросов:

```
SELECT count( * )
FROM tickets_2
WHERE substr( passenger_name, strpos( passenger_name, ' ' ) + 1 ) = 'IVANOV';
SELECT count( * )
FROM tickets_2
WHERE passenger_lname = 'IVANOV';
```

Предложите для базы данных «Авиаперевозки» ваши модификации таблиц, возможно, с использованием более сложных выражений для вычисляемых столбцов, которые дадут еще больший эффект. В качестве возможной идеи можно рассмотреть добавление в ту же таблицу «Билеты» (`tickets`) столбцов для хранения номера телефона и адреса электронной почты пассажиров. Эти данные можно извлекать из столбца `contact_data`.

32 Сортировка массивов разной размерности

В разделе документации 9.19 «Функции и операторы для работы с массивами» представлено много разнообразных функций для обработки массивов, однако функции сортировки среди них нет. Давайте ее напишем, но сначала уточним, что мы понимаем под сортировкой массива. В случае одномерного массива речь идет о сортировке его элементов. Элементами двухмерного массива являются одномерные массивы, поэтому сначала сортируются элементы каждого одномерного массива, а потом — сами одномерные массивы (в лексикографическом порядке). Аналогично понимается и сортировка трехмерных массивов, учитывая, что их элементы — двухмерные массивы.

Начнем с сортировки одномерного массива.

```
CREATE OR REPLACE FUNCTION array_sort( arr integer[] )
RETURNS integer[] AS
$$
  SELECT array_agg( elem ORDER BY elem ) AS sorted_array
  FROM unnest( arr ) AS elem;
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
```

Вот что получается:

```
SELECT *
FROM array_sort( ARRAY[ 8, 3, 1, 4, 2, 9 ] );
  array_sort
---------------
 {1,2,3,4,8,9}
(1 строка)
```

Для сортировки двухмерного массива можно воспользоваться функцией, представленной в разделе документации 9.26 «Функции, возвращающие множества»: generate_subscripts. Она формирует в виде таблицы список действительных индексов в указанном измерении массива. В предложении FROM подзапроса sort_subarrays получим все комбинации обоих индексов, а затем с их помощью сформируем одномерные массивы и отсортируем каждый из них. Для получения окончательного результата соберем в двухмерный массив отсортированные одномерные массивы.

```
CREATE OR REPLACE FUNCTION array_sort_2d( arr integer[][] )
RETURNS integer[][] AS
$$
  WITH sort_subarrays AS
  ( SELECT
      array_agg( arr[ i ][ j ] ORDER BY arr[ i ][ j ] ) AS subarray
    FROM
      generate_subscripts( arr, 1 ) AS i,
      generate_subscripts( arr, 2 ) AS j
    GROUP BY i
  )
  SELECT array_agg( subarray ORDER BY subarray )
  FROM sort_subarrays;
$$
LANGUAGE sql IMMUTABLE;
```

```
CREATE FUNCTION
```

Проверяем функцию в работе:

```
SELECT *
FROM array_sort_2d(
    ARRAY [
      [ 12, 3 ], [ 4, 9 ],
      [ 11, 7 ], [ 17, 3 ],
      [ 8, 14 ], [ 22, 4 ]
    ]
  );
```

```
              array_sort_2d
-----------------------------------------
 {{3,12},{3,17},{4,9},{4,22},{7,11},{8,14}}
(1 строка)
```

Задание. Напишите функцию сортировки трехмерных массивов. Начало одного из вариантов может быть таким:

```
CREATE OR REPLACE FUNCTION array_sort_3d( arr integer[][][] )
RETURNS integer[][][] AS
$$
  WITH sort_subarrays_3rd_dim AS
  ( SELECT
      i,
      j,
      array_agg( arr[ i ][ j ][ k ] ORDER BY arr[ i ][ j ][ k ] ) AS subarray_3rd_dim
    FROM
      generate_subscripts( arr, 1 ) AS i,
      generate_subscripts( arr, 2 ) AS j,
      generate_subscripts( arr, 3 ) AS k
    GROUP BY i, j
  ),
...
```

Ожидается получить такой результат:

```
SELECT *
FROM array_sort_3d(
    ARRAY [
      [ [ 12, 3 ], [ 4, 9 ] ],
      [ [ 11, 7 ], [ 17, 3 ] ],
      [ [ 8, 14 ], [ 22, 4 ] ]
    ]
  );
                  array_sort_3d
-----------------------------------------------
 {{{3,12},{4,9}},{{3,17},{7,11}},{{4,22},{8,14}}}
(1 строка)
```

33 Формирование русского или английского алфавита

Иногда, например при обработке фамилий и имен пассажиров, может оказаться полезной функция, формирующая алфавит в виде массива или столбца таблицы.

В качестве примера представим функцию, формирующую таблицу с заглавными буквами английского алфавита. Обратите внимание, что тело функции написано в стиле стандарта SQL.

```
CREATE OR REPLACE FUNCTION make_abc_english()
RETURNS SETOF text
LANGUAGE sql
BEGIN ATOMIC
  SELECT chr( ascii( 'A' ) + delta )
  FROM generate_series( 0, ascii( 'Z' ) - ascii( 'A' ) ) AS delta;
END;
CREATE FUNCTION
SELECT * FROM make_abc_english();
 make_abc_english
------------------
 A
 B
 C
 ...
 X
 Y
 Z
(26 строк)
```

Задание 1. Предложите другое решение задачи для английского алфавита.

Задание 2. Напишите такую функцию для русского алфавита (учтите, что бывают разные кодировки символов).

Указание. Можно воспользоваться, например, функцией `regexp_matches`, передав ей в качестве первого аргумента строку, содержащую все буквы алфавита.

34 Функция для определения степени заполнения самолетов

Важным экономическим показателем работы авиакомпании является степень заполнения самолетов, выполняющих рейсы. Напишите функцию, вычисляющую этот показатель для заданного рейса с учетом класса обслуживания.

Результат может быть примерно таким:

```
SELECT f.flight_id, o.*
FROM
  flights AS f,
  occupancy_rate( flight_id ) AS o
WHERE f.flight_id IN ( 1, 1162 )
ORDER BY f.flight_id, o.f_cond;
```

```
 flight_id |  f_cond  | occupied_seats | total_seats | occupancy_rate
-----------+----------+----------------+-------------+----------------
         1 | Business |             12 |          28 |           0.43
         1 | Economy  |             67 |         142 |           0.47
      1162 | Business |              7 |          20 |           0.35
      1162 | Economy  |             44 |          96 |           0.46
(4 строки)
```

Предметный указатель

Книги издательства «ДМК Пресс»
можно купить оптом и в розницу на складе издательства
по адресу: г. Москва, ул. Электродная, д. 2, стр. 12, офис 7,
тел. +7 (499) 322-19-38,
а также заказать на сайте www.dmkpress.com
с доставкой в любой регион РФ

Моргунов Евгений Павлович

PostgreSQL. Профессиональный SQL

Учебное пособие

При поддержке Postgres Professional
https://postgrespro.ru

Главный редактор *Мовчан Д. А.*
Зам. главного редактора *Яценков В. С.*
editor@dmkpress.com
Корректор *Синяева Г. И.*
Дизайн обложки *Лукашенко М. А.*

Формат 70×100 1/16. Печать цифровая.
Гарнитура ПТ (Паратайп) и Iosevka (Renzhi Li).
Усл. печ. л. 36,08. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com