Глава 6

Запросы

Эта глава будет самой насыщенной и интересной, поскольку умение писать SQL-запросы — это не только ремесло, но, пожалуй, и искусство тоже.

В предыдущих главах мы уже не раз использовали команду SELECT и формировали с ее помощью различные запросы. Эти запросы строились как на основе одной таблицы, так и на основе двух и более таблиц. Мы рассмотрели простые способы сортировки и группировки строк в полученных выборках из таблиц, использовали функцию count для подсчета числа выбранных строк. Таким образом, вы уже получили элементарное представление о том, как формировать выборки из базы данных. В этой главе мы покажем более сложные способы их получения.

С целью приведения в систему тех знаний о формировании выборок, что были получены в предыдущих главах, в этой главе мы повторим некоторые сведения, но сделаем это уже на новых примерах.

6.1. Дополнительные возможности команды SELECT

Основой для экспериментов в этом разделе будут самые маленькие (по числу строк) таблицы базы данных «Авиаперевозки»: «Самолеты» (`aircrafts`) и «Аэропорты» (`airports`).

Прежде чем перейти к конкретным запросам, просто просмотрите содержимое этих двух таблиц. Таблица «Самолеты» совсем маленькая, а таблица «Аэропорты» содержит чуть больше ста строк. Для ее просмотра можно включить расширенный режим вывода данных \x.

```
SELECT * FROM aircrafts;
SELECT * FROM airports;
```

Начнем с различных условий отбора строк в предложении WHERE. Эти условия могут конструироваться с использованием следующих **операторов сравнения**: =, <>, >, > =, <, < =. В предыдущих главах мы уже использовали ряд таких операторов, поэтому сейчас рассмотрим некоторые другие способы осуществления отбора строк.

Для начала поставим перед собой такую задачу: выбрать все самолеты компании Airbus. В этом нам поможет оператор поиска **шаблонов LIKE**:

```
SELECT * FROM aircrafts WHERE model LIKE 'Airbus%';
```

Обратите внимание на символ «%», имеющий специальное значение. Он соответствует любой последовательности символов, т. е. вместо него могут быть подставлены любые символы в любом количестве, а может и не быть подставлено ни одного символа. В результате будут выбраны строки, в которых значения атрибута `model` начинаются с символов «Airbus»:

```
 aircraft_code |      model      | range
---------------+-----------------+-------
 320           | Airbus A320-200 |  5700
 321           | Airbus A321-200 |  5600
 319           | Airbus A319-100 |  6700
(3 строки)
```

Шаблон в операторе `LIKE` всегда покрывает *всю* анализируемую строку. Поэтому если требуется отыскать некоторую последовательность символов где-то внутри строки, то шаблон должен начинаться и завершаться символом «%». Однако в этом случае нужно учитывать следующие соображения. Если по тому столбцу, к которому применяется оператор `LIKE`, создан индекс для ускорения доступа к данным, то при наличии символа «%» в начале шаблона этот индекс использоваться не будет. Из-за этого может ухудшиться производительность, т. е. запрос будет выполняться медленнее. Индексы подробно рассматриваются в главе 8, а вопросы производительности — в главе 10.

Конечно, существует и оператор `NOT LIKE`. Например, если мы захотим узнать, какими самолетами, кроме машин компаний Airbus и Boeing, располагает наша авиакомпания, то придется усложнить условие:

```
SELECT * FROM aircrafts
  WHERE model NOT LIKE 'Airbus%'
    AND model NOT LIKE 'Boeing%';
```

```
 aircraft_code |       model         | range
---------------+---------------------+-------
 SU9           | Sukhoi SuperJet-100 |  3000
 CN1           | Cessna 208 Caravan  |  1200
 CR2           | Bombardier CRJ-200  |  2700
(3 строки)
```

Кроме символа «%» в шаблоне может использоваться и символ подчеркивания — «_», который соответствует в точности одному любому символу. В качестве примера найдем в таблице «Аэропорты» те из них, которые имеют названия длиной три символа (буквы). С этой целью зададим в качестве шаблона строку, состоящую из трех символов «_».

```
SELECT * FROM airports WHERE airport_name LIKE '___';
```

```
-[ RECORD 1 ]-+------------------
 airport_code | UFA
 airport_name | Уфа
 city         | Уфа
 longitude    | 55.874417
 latitude     | 54.557511
 timezone     | Asia/Yekaterinburg
```

Существует ряд операторов для работы с **регулярными выражениями** POSIX. Эти операторы имеют больше возможностей, чем оператор LIKE. Для того чтобы выбрать, например, самолеты компаний Airbus и Boeing, можно сделать так:

```
SELECT * FROM aircrafts WHERE model ~ '^(A|Boe)';
```

```
 aircraft_code |      model      | range
---------------+-----------------+-------
 773           | Boeing 777-300  | 11100
 763           | Boeing 767-300  |  7900
 320           | Airbus A320-200 |  5700
 321           | Airbus A321-200 |  5600
 319           | Airbus A319-100 |  6700
 733           | Boeing 737-300  |  4200
(6 строк)
```

Оператор ~ ищет совпадение с шаблоном с учетом регистра символов. Символ «^» в начале регулярного выражения означает, что поиск совпадения будет привязан к началу строки. Если же требуется проверить наличие такого символа *в составе* строки, то перед ним нужно поставить символ обратной косой черты «\». Выражение в круглых скобках означает альтернативный выбор между значениями, разделяемыми символом «|». Поэтому в выборку попадут значения, начинающиеся либо на «A», либо на «Boe».

Для инвертирования смысла оператора ~ нужно перед ним добавить знак «!». В качестве примера отыщем модели самолетов, которые не завершаются числом 300.

```
SELECT * FROM aircrafts WHERE model !~ '300$';
```

В этом регулярном выражении символ «$» означает привязку поискового шаблона к концу строки. Если же требуется проверить наличие такого символа *в составе* строки, то перед ним нужно поставить символ обратной косой черты «\».

```
 aircraft_code |       model         | range
---------------+---------------------+-------
 SU9           | Sukhoi SuperJet-100 |  3000
 320           | Airbus A320-200     |  5700
 321           | Airbus A321-200     |  5600
 319           | Airbus A319-100     |  6700
 CN1           | Cessna 208 Caravan  |  1200
 CR2           | Bombardier CRJ-200  |  2700
(6 строк)
```

Использование регулярных выражений подробно рассматривается в разделе документации 9.7.3 «Регулярные выражения POSIX».

В качестве замены традиционных операторов сравнения могут использоваться **предикаты сравнения**, которые ведут себя так же, как и операторы, но имеют другой синтаксис.

Давайте ответим на вопрос: какие самолеты имеют дальность полета в диапазоне от 3 000 км до 6 000 км? Ответ получим с помощью предиката BETWEEN.

```
SELECT * FROM aircrafts WHERE range BETWEEN 3000 AND 6000;
```

```
 aircraft_code |        model        | range
---------------+---------------------+-------
 SU9           | Sukhoi SuperJet-100 |  3000
 320           | Airbus A320-200     |  5700
 321           | Airbus A321-200     |  5600
 733           | Boeing 737-300      |  4200
(4 строки)
```

Обратите внимание, что граничное значение 3 000 включено в полученную выборку.

При выборке данных можно проводить вычисления и получать в результирующей таблице **вычисляемые столбцы**. Если мы захотим представить дальность полета не только в километрах, но и в милях, то нужно вычислить это выражение и для удобства присвоить новому столбцу псевдоним с помощью ключевого слова AS.

```
SELECT model, range, range / 1.609 AS miles FROM aircrafts;
```

```
     model           | range |        miles
---------------------+-------+----------------------
 Boeing 777-300      | 11100 | 6898.6948415164698571
 Boeing 767-300      |  7900 | 4909.8819142324425109
 ...
(9 строк)
```

По всей вероятности, такая высокая точность представления значений в милях не требуется, поэтому мы можем уменьшить ее до разумного предела в два десятичных знака:

```
SELECT model, range, round( range / 1.609, 2 ) AS miles
  FROM aircrafts;
```

```
       model          | range |  miles
----------------------+-------+---------
 Boeing 777-300       | 11100 | 6898.69
 Boeing 767-300       |  7900 | 4909.88
 ...
```

Теперь обратимся к такому вопросу, как **упорядочение строк** при выводе. Если не принять специальных мер, то СУБД не гарантирует никакого конкретного порядка строк в результирующей выборке. Для упорядочения строк служит **предложение ORDER BY**, которое мы уже использовали ранее. Однако мы не говорили, что можно задать не только возрастающий, но также и убывающий порядок сортировки. Например, если мы захотим разместить самолеты в порядке убывания дальности их полета, то нужно сделать так:

```
SELECT * FROM aircrafts ORDER BY range DESC;
```

```
 aircraft_code |        model         | range
---------------+----------------------+-------
 773           | Boeing 777-300       | 11100
 763           | Boeing 767-300       |  7900
 319           | Airbus A319-100      |  6700
 320           | Airbus A320-200      |  5700
 321           | Airbus A321-200      |  5600
 733           | Boeing 737-300       |  4200
 SU9           | Sukhoi Superjet-100  |  3000
 CR2           | Bombardier CRJ-200   |  2700
 CN1           | Cessna 208 Caravan   |  1200
(9 строк)
```

Мы детально разобрались с таблицей «Самолеты» и теперь обратим наше внимание на таблицу «Аэропорты»). В ней есть столбец «Часовой пояс» (`timezone`). Давайте посмотрим, в каких различных часовых поясах располагаются аэропорты. Если сделать традиционную выборку

```
SELECT timezone FROM airports;
```

то мы получим список значений, среди которых будет много повторяющихся. Конечно, это неудобно. Для того чтобы оставить в выборке только *неповторяющиеся значения*, служит **ключевое слово `DISTINCT`**:

```
SELECT DISTINCT timezone FROM airports ORDER BY 1;
```

Обратите внимание, что столбец, по значениям которого будут упорядочены строки, указан не с помощью его имени, а с помощью его порядкового номера в предложении `SELECT`.

Получим такой результат:

```
       timezone
--------------------
 Asia/Anadyr
 Asia/Chita
 Asia/Irkutsk
 Asia/Kamchatka
 Asia/Krasnoyarsk
 Asia/Magadan
 Asia/Novokuznetsk
 Asia/Novosibirsk
 Asia/Omsk
 Asia/Sakhalin
 Asia/Vladivostok
 Asia/Yakutsk
 Asia/Yekaterinburg
 Europe/Kaliningrad
 Europe/Moscow
 Europe/Samara
 Europe/Volgograd
(17 строк)
```

Таким образом, аэропорты располагаются в семнадцати различных часовых поясах. Они описаны в базе данных часовых поясов, поддерживаемой международной организацией IANA (Internet Assigned Numbers Authority), и отличаются от традиционных

географических и административных часовых поясов, число которых в России равно одиннадцати.

В таблице «Аэропорты» более ста записей. Если мы поставим задачу найти три самых восточных аэропорта, то для ее решения подошел бы такой алгоритм: отсортировать строки в таблице по убыванию значений столбца «Долгота» (`longitude`) и включить в выборку только первые три строки. Как отсортировать строки по убыванию значений какого-либо столбца, вы уже знаете, а для того чтобы ограничить число строк, включаемых в результирующую выборку, служит **предложение `LIMIT`**.

```
SELECT airport_name, city, longitude
  FROM airports
  ORDER BY longitude DESC
  LIMIT 3;
```

```
 airport_name |          city             | longitude
--------------+---------------------------+------------
 Анадырь      | Анадырь                   | 177.741483
 Елизово      | Петропавловск-Камчатский | 158.453669
 Магадан      | Магадан                   | 150.720439
(3 строки)
```

А как найти еще три аэропорта, которые находятся немного западнее первой тройки, т. е. занимают места с четвертого по шестое? Алгоритм будет почти таким же, как в первой задаче, но он будет дополнен еще одним шагом: нужно пропустить три первые строки, прежде чем начать вывод. Для пропуска строк служит **предложение `OFFSET`**.

```
SELECT airport_name, city, longitude
  FROM airports
  ORDER BY longitude DESC
  LIMIT 3
  OFFSET 3;
```

```
 airport_name    |         city          | longitude
-----------------+-----------------------+------------
 Хомутово        | Южно-Сахалинск        | 142.717531
 Хурба           | Комсомольск-на-Амуре |    136.934
 Хабаровск-Новый | Хабаровск             | 135.188361
(3 строки)
```

В дополнение к вычисляемым столбцам, когда выводимые значения получают путем вычислений, при выборке данных из таблиц можно использовать **условные выражения**, позволяющие вывести то или иное значение в зависимости от условий.

В таблице «Самолеты» есть столбец «Максимальная дальность полета» (range). Мы можем дополнить вывод данных из этой таблицы столбцом «Класс самолета», имея в виду принадлежность каждого самолета к классу дальнемагистральных, среднемагистральных или ближнемагистральных судов.

Для этого подойдет конструкция

```
CASE WHEN условие THEN выражение
  [ WHEN ... ]
  [ ELSE выражение ]
END
```

Воспользовавшись этой конструкцией в предложении SELECT и назначив новому столбцу имя с помощью ключевого слова AS, получим следующий запрос:

```
SELECT model, range,
  CASE WHEN range < 2000 THEN 'Ближнемагистральный'
       WHEN range < 5000 THEN 'Среднемагистральный'
       ELSE 'Дальнемагистральный'
  END AS type
  FROM aircrafts
  ORDER BY model;
```

```
        model         | range | type
----------------------+-------+----------------------
 Airbus A319-100      |  6700 | Дальнемагистральный
 Airbus A320-200      |  5700 | Дальнемагистральный
 Airbus A321-200      |  5600 | Дальнемагистральный
 Boeing 737-300       |  4200 | Среднемагистральный
 Boeing 767-300       |  7900 | Дальнемагистральный
 Boeing 777-300       | 11100 | Дальнемагистральный
 Bombardier CRJ-200   |  2700 | Среднемагистральный
 Cessna 208 Caravan   |  1200 | Ближнемагистральный
 Sukhoi SuperJet-100 |  3000 | Среднемагистральный
(9 строк)
```

6.2. Соединения

В тех случаях, когда информации, содержащейся в одной таблице, недостаточно для получения требуемого результата, используют **соединение (join)** таблиц. Покажем способ выполнения соединения на примере следующего запроса: выбрать все места, предусмотренные компоновкой салона самолета Cessna 208 Caravan.

Сначала приведем SQL-команду для выполнения запроса, а потом объясним, как мы ее придумали.

```
SELECT a.aircraft_code, a.model, s.seat_no, s.fare_conditions
  FROM seats AS s
  JOIN aircrafts AS a
    ON s.aircraft_code = a.aircraft_code
  WHERE a.model ~ '^Cessna'
  ORDER BY s.seat_no;
```

В предложении WHERE мы применили регулярное выражение, хотя в данном случае можно было с таким же успехом воспользоваться и оператором LIKE или функцией substr.

```
 aircraft_code |       model        | seat_no | fare_conditions
---------------+--------------------+---------+-----------------
 CN1           | Cessna 208 Caravan | 1A      | Economy
 CN1           | Cessna 208 Caravan | 1B      | Economy
 CN1           | Cessna 208 Caravan | 2A      | Economy
 CN1           | Cessna 208 Caravan | 2B      | Economy
 CN1           | Cessna 208 Caravan | 3A      | Economy
 CN1           | Cessna 208 Caravan | 3B      | Economy
 CN1           | Cessna 208 Caravan | 4A      | Economy
 CN1           | Cessna 208 Caravan | 4B      | Economy
 CN1           | Cessna 208 Caravan | 5A      | Economy
 CN1           | Cessna 208 Caravan | 5B      | Economy
 CN1           | Cessna 208 Caravan | 6A      | Economy
 CN1           | Cessna 208 Caravan | 6B      | Economy
(12 строк)
```

Данная команда иллюстрирует **соединение двух таблиц на основе равенства значений атрибутов**.

В этой команде в предложении FROM указаны две таблицы — aircrafts и seats, причем каждая из них получила еще и псевдоним с помощью ключевого слова AS (заметим, что оно не является обязательным). Конечно, псевдонимы могут состоять не только из одной буквы, как в нашем примере. Псевдонимы удобны в тех случаях, когда в соединяемых таблицах есть одноименные атрибуты. В таких случаях в списке атрибутов, следующих за ключевым словом SELECT, необходимо указывать либо имя таблицы, из которой выбирается значение этого атрибута, либо ее псевдоним, но псевдоним может быть коротким, что удобнее при написании команды. Псевдоним и атрибут соединяются символом «.». Псевдонимы используются и в предложениях WHERE, GROUP BY, ORDER BY, HAVING, т. е. во всех частях команды SELECT.

Итак, как мы рассуждали? Если бы в качестве исходных сведений мы получили сразу код самолета — CN1, то запрос свелся бы к выборке из одной таблицы «Места». Он был бы таким:

```
SELECT * FROM seats WHERE aircraft_code = 'CN1';
```

Но нам дано название модели, а не ее код, поэтому придется подключить к работе и таблицу «Самолеты» (aircrafts), в которой хранятся наименования моделей. Для того чтобы решить, удовлетворяет ли строка таблицы seats поставленному условию, нужно узнать, какой модели самолета соответствует эта строка.

Как это можно узнать? В каждой строке таблицы seats есть атрибут aircraft_code, такой же атрибут есть и в каждой строке таблицы aircrafts. Если с каждой строкой таблицы seats соединить такую строку таблицы aircrafts, в которой значение атрибута aircraft_code такое же, как и в строке таблицы seats, то сформированная комбинированная строка, составленная из атрибутов обеих таблиц, будет содержать не только номер места, класс обслуживания и код модели, но — что важно — и наименование модели. Поэтому с помощью условия WHERE можно будет отобрать только те результирующие строки, в которых значение атрибута model будет «Cessna 208 Caravan».

А какие столбцы оставлять в списке столбцов предложения SELECT, решать нам. Даже если мы соединяем две таблицы (или более), то совершенно не обязательно в результирующий список столбцов включать столбцы всех таблиц, перечисленных в предложении FROM. Мы могли бы оставить только атрибуты таблицы seats:

```
SELECT s.seat_no, s.fare_conditions
  FROM seats s
  JOIN aircrafts a ON s.aircraft_code = a.aircraft_code
  WHERE a.model ~ '^Cessna'
  ORDER BY s.seat_no;
```

```
 seat_no | fare_conditions
---------+-----------------
 1A      | Economy
 1B      | Economy
 2A      | Economy
 2B      | Economy
 3A      | Economy
 3B      | Economy
 4A      | Economy
 4B      | Economy
 5A      | Economy
```

```
 5B       | Economy
 6A       | Economy
 6B       | Economy
(12 строк)
```

Если подвести итог, то можно упрощенно объяснить механизм построения соединения следующим образом.

Сначала формируются все попарные комбинации строк из обеих таблиц, т. е. декартово произведение множеств строк этих таблиц. Эти комбинированные строки включают в себя все атрибуты обеих таблиц.

Затем в дело вступает условие `s.aircraft_code = a.aircraft_code`. Это означает, что в результирующем множестве строк останутся только те из них, в которых значения атрибута `aircraft_code`, взятые из таблицы `aircrafts` и из таблицы `seats`, одинаковые. Строки, не удовлетворяющие этому критерию, отфильтровываются.

Это означает на практике, что каждой строке из таблицы «Места» мы сопоставили только одну конкретную строку из таблицы «Самолеты», из которой мы теперь можем взять значение атрибута «Модель самолета», чтобы включить ее в итоговый вывод данных.

На практике описанный механизм не реализуется буквально. Специальная подсистема PostgreSQL, называемая планировщиком, строит план выполнения запроса, который является гораздо более эффективным, чем упрощенный план, представленный здесь. Детально вопросы планирования запросов рассматриваются в главе 10.

Запрос, который мы рассмотрели, можно записать немного по-другому, без использования предложения `JOIN` (обратите внимание, что мы не использовали ключевое слово `AS` для назначения псевдонимов таблицам).

```
SELECT a.aircraft_code, a.model, s.seat_no, s.fare_conditions
  FROM seats s, aircrafts a
  WHERE s.aircraft_code = a.aircraft_code
    AND a.model ~ '^Cessna'
  ORDER BY s.seat_no;
```

В этом варианте условие соединения таблиц `s.aircraft_code = a.aircraft_code` перешло из предложения `FROM` в предложение `WHERE`, а таблицы просто перечислены в предложении `FROM` через запятую. Простые запросы зачастую записывают именно в такой форме, без предложения `JOIN`, а в предложении `WHERE` указывают критерии, которым должны удовлетворять результирующие строки.

Изучая язык SQL вообще и способы выполнения соединений в частности, нужно иметь в виду, что *результатом любых реляционных операций над отношениями (таблицами, представлениями) также является отношение*. Поэтому такие операции можно произвольно комбинировать друг с другом.

В соединении одна и та же таблица может участвовать дважды, т. е. формировать **соединение таблицы с самой собой**. В качестве примера рассмотрим запрос для создания представления «Рейсы» (`flights_v`), о котором шла речь в главе 5.

Этот запрос выглядит так:

```
CREATE OR REPLACE VIEW flights_v AS
  SELECT f.flight_id,
         f.flight_no,
         f.scheduled_departure,
         timezone( dep.timezone, f.scheduled_departure )
           AS scheduled_departure_local,
         f.scheduled_arrival,
         timezone( arr.timezone, f.scheduled_arrival )
           AS scheduled_arrival_local,
         f.scheduled_arrival - f.scheduled_departure
           AS scheduled_duration,
         f.departure_airport,
         dep.airport_name AS departure_airport_name,
         dep.city AS departure_city,
         f.arrival_airport,
         arr.airport_name AS arrival_airport_name,
         arr.city AS arrival_city,
         f.status,
         f.aircraft_code,
         f.actual_departure,
         timezone( dep.timezone, f.actual_departure )
           AS actual_departure_local,
         f.actual_arrival,
         timezone( arr.timezone, f.actual_arrival )
           AS actual_arrival_local,
         f.actual_arrival - f.actual_departure AS actual_duration
  FROM flights f,
       airports dep,
       airports arr
  WHERE f.departure_airport = dep.airport_code
    AND f.arrival_airport = arr.airport_code;
```

В этом представлении используется не только таблица «Рейсы» (flights), но также и таблица «Аэропорты» (airports). Причем она используется, условно говоря, дважды. Поясним, что мы имеем в виду.

Как вы уже знаете из главы 3, при соединении двух таблиц в результирующую выборку попадают те комбинации строк из первой и второй таблиц, которые удовлетворяют условию, указанному в предложении WHERE. Будем рассуждать от противного. Пусть в предложение FROM таблица «Аэропорты» (airports) будет указана только один раз, тогда предложения FROM и WHERE будут выглядеть так:

```
FROM flights f, airports a
WHERE f.departure_airport = a.airport_code
  AND f.arrival_airport = a.airport_code;
```

Это означает, что при соединении двух таблиц PostgreSQL будет пытаться для каждой строки из таблицы flights найти такую строку в таблице airports, в которой значение атрибута airport_code будет равно не только значению атрибута departure_airport, но также и значению атрибута arrival_airport в таблице flights. Получается, что данное условие будет выполнено, если только аэропорт вылета и аэропорт назначения будет одним и тем же. Однако в сфере пассажирских авиаперевозок таких рейсов не бывает. Конечно, иногда самолеты возвращаются в пункт вылета, но это уже совсем другая ситуация, которая в нашей учебной базе данных не учитывается.

Таким образом, приходим к выводу о том, что каждую строку из таблицы «Рейсы» необходимо соединять с двумя *различными* строками из таблицы «Аэропорты»: ведь аэропорт вылета и аэропорт назначения — это *различные* аэропорты. Однако при однократном включении таблицы «Аэропорты» в предложение FROM сделать это невозможно, поэтому поступают так: к таблице airports в предложении FROM обращаются дважды, как будто это две копии одной и той же таблицы.

Конечно, на самом деле никаких копий не создается. Просто в результате поиск строк в ней будет производиться дважды: один раз для атрибута departure_airport, а второй раз — для атрибута arrival_airport. Но поскольку необходимо обеспечить однозначную идентификацию, то каждой «копии» (экземпляру) таблицы airports присваивают уникальный псевдоним, в нашем случае это dep и arr, т. е. departure и arrival. Эти псевдонимы указывают, из какой «копии» (экземпляра) таблицы airports нужно брать значение атрибута airport_code для сопоставления с атрибутами departure_airport и arrival_airport.

Рассмотрев этот пример, вновь обратимся к соединениям такого типа и покажем три способа выполнения **соединения таблицы с самой собой**, отличающиеся синтаксически, но являющиеся функционально эквивалентными. Наш запрос-иллюстрация должен выяснить: сколько всего маршрутов нужно было бы сформировать, если бы требовалось соединить каждый город со всеми остальными городами? Если в городе имеется более одного аэропорта, то договоримся рейсы из каждого из них (в каждый из них) считать отдельными маршрутами. Поэтому правильнее было бы говорить не о маршрутах из каждого города, а о маршрутах из каждого аэропорта во все другие аэропорты. Конечно, рейсов из любого города в тот же самый город быть не должно.

Первый вариант запроса использует обычное перечисление имен таблиц в предложении FROM. Поскольку имена таблиц совпадают, используются псевдонимы. В таком случае СУБД обращается к таблице дважды, как если бы это были различные таблицы.

```
SELECT count( * )
  FROM airports a1, airports a2
  WHERE a1.city <> a2.city;
```

Как мы уже говорили ранее, СУБД соединяет каждую строку первой таблицы с каждой строкой второй таблицы, т. е. формирует **декартово произведение** таблиц — все попарные комбинации строк из двух таблиц. Затем СУБД отбрасывает те комбинированные строки, которые не удовлетворяют условию, приведенному в предложении WHERE. В нашем примере условие как раз и отражает требование о том, что рейсов из одного города в тот же самый город быть не должно.

```
 count
-------
 10704
(1 строка)
```

Во втором варианте запроса мы используем **соединение таблиц на основе неравенства значений атрибутов**. Тем самым мы перенесли условие отбора результирующих строк из предложения WHERE в предложение FROM.

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

```
 count
-------
 10704
(1 строка)
```

Третий вариант предусматривает **явное использование декартова произведения таблиц**. Для этого служит предложение CROSS JOIN. Лишние строки, как и в первом варианте, отсеиваем с помощью предложения WHERE:

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

```
 count
-------
 10704
(1 строка)
```

С точки зрения СУБД эти три варианта эквивалентны и отличаются лишь синтаксисом. Для них PostgreSQL выберет один и тот же план (порядок) выполнения запроса.

Теперь обратимся к так называемым **внешним соединениям**. Зададимся вопросом: сколько маршрутов обслуживают самолеты каждого типа? Если не требовать вывода наименований моделей самолетов, тогда всю необходимую информацию можно получить из материализованного представления «Маршруты» (routes). Но мы все же будем выводить и наименования моделей, поэтому обратимся также к таблице «Самолеты» (aircrafts). Соединим эти таблицы на основе атрибута aircraft_code, сгруппируем строки и просто воспользуемся функцией count. В этом запросе внешнее соединение еще не используется.

```
SELECT r.aircraft_code, a.model, count( * ) AS num_routes
  FROM routes r
  JOIN aircrafts a ON r.aircraft_code = a.aircraft_code
  GROUP BY 1, 2
  ORDER BY 3 DESC;
```

```
 aircraft_code |        model         | num_routes
---------------+----------------------+------------
 CR2           | Bombardier CRJ-200   |        232
 CN1           | Cessna 208 Caravan   |        170
 SU9           | Sukhoi SuperJet-100  |        158
 319           | Airbus A319-100      |         46
 733           | Boeing 737-300       |         36
 321           | Airbus A321-200      |         32
 763           | Boeing 767-300       |         26
 773           | Boeing 777-300       |         10
(8 строк)
```

Обратите внимание, что таблица «Самолеты» содержит 9 моделей, а в этой выборке лишь 8 строк. Значит, какая-то модель самолета не участвует в выполнении рейсов. Как ее выявить?

С помощью такого запроса:

```
SELECT a.aircraft_code AS a_code,
       a.model,
       r.aircraft_code AS r_code,
       count( r.aircraft_code ) AS num_routes
  FROM aircrafts a
  LEFT OUTER JOIN routes r ON r.aircraft_code = a.aircraft_code
  GROUP BY 1, 2, 3
  ORDER BY 4 DESC;
```

```
 a_code |        model         | r_code | num_routes
--------+----------------------+--------+------------
 CR2    | Bombardier CRJ-200   | CR2    |        232
 CN1    | Cessna 208 Caravan   | CN1    |        170
 SU9    | Sukhoi SuperJet-100  | SU9    |        158
 319    | Airbus A319-100      | 319    |         46
 733    | Boeing 737-300       | 733    |         36
 321    | Airbus A321-200      | 321    |         32
 763    | Boeing 767-300       | 763    |         26
 773    | Boeing 777-300       | 773    |         10
 320    | Airbus A320-200      |        |          0
(9 строк)
```

В данном запросе используется **левое внешнее соединение** — об этом говорит предложение `LEFT OUTER JOIN`.

В качестве базовой таблицы выбирается таблица `aircrafts`, указанная в запросе слева от предложения `LEFT OUTER JOIN`, и для каждой строки, находящейся в ней, из таблицы `routes` подбираются строки, в которых значение атрибута `aircraft_code` такое же, как и в текущей строке таблицы `aircrafts`. Если в таблице `routes` нет ни одной соответствующей строки, то при отсутствии ключевых слов `LEFT OUTER` результирующая комбинированная строка просто не будет сформирована и не попадет в выборку. Но при наличии ключевых слов `LEFT OUTER` результирующая строка все равно будет сформирована.

Это происходит таким образом: если для строки из левой таблицы (левой относительно предложения `LEFT OUTER JOIN`) не находится ни одной соответствующей строки

в правой таблице, тогда в результирующую строку вместо значений столбцов правой таблицы будут помещены значения NULL. Получается, что для строки из таблицы `aircrafts`, в которой значение атрибута `aircraft_code` равно `320`, в таблице `routes` нет ни одной строки с таким же значением этого атрибута. В результате при выводе выборки в столбце `a_code`, взятом из таблицы `aircrafts`, будет значение `320`, а в столбце `r_code`, взятом из таблицы `routes`, будет значение NULL. Этот столбец включен в выборку лишь для повышения наглядности, в реальном запросе он не нужен.

Обратите внимание, что параметром функции `count` является столбец из таблицы `routes`, поэтому count и выдает число 0 для самолета с кодом `320`. Если заменить его на одноименный столбец из таблицы `aircrafts`, тогда count выдаст 1, что будет противоречить цели нашей задачи — подсчитать число рейсов, выполняемых на самолетах каждого типа. Напомним, что если функция `count` в качестве параметра получает не символ «*», а имя столбца, тогда она подсчитывает число строк, в которых значение в этом столбце определено (не равно NULL).

Кроме левого внешнего соединения существует также и **правое внешнее соединение** — `RIGHT OUTER JOIN`.

В этом случае в качестве базовой выбирается таблица, имя которой указано справа от предложения `RIGHT OUTER JOIN`, а механизм получения результирующих строк в случае, когда для строки базовой таблицы не находится пары во второй таблице, точно такой же, как и для левого внешнего соединения. Как сказано в документации, правое внешнее соединение является лишь синтаксическим приемом, поскольку всегда можно заменить его левым внешним соединением, поменяв при этом имена таблиц местами.

Важно учитывать, что порядок следования таблиц в предложениях `LEFT (RIGHT) OUTER JOIN` никак не влияет на порядок столбцов в предложении `SELECT`. В вышеприведенном запросе мы написали

```
SELECT a.aircraft_code AS a_code,
       a.model,
       r.aircraft_code AS r_code,
       ...
```

Но если бы нам это было нужно, то мы могли бы поменять столбцы местами:

```
SELECT r.aircraft_code AS r_code,
       a.model,
       a.aircraft_code AS a_code,
       ...
```

Комбинацией этих двух видов внешних соединений является **полное внешнее соединение** — FULL OUTER JOIN.

В этом случае в выборку включаются строки из левой таблицы, для которых не нашлось соответствующих строк в правой таблице, и строки из правой таблицы, для которых не нашлось соответствующих строк в левой таблице.

В практической работе при выполнении выборок зачастую выполняются **многотабличные запросы**, включающие три таблицы и более. В качестве примера рассмотрим такую задачу: определить число пассажиров, не пришедших на регистрацию билетов и, следовательно, не вылетевших в пункт назначения. Будем учитывать только рейсы, у которых фактическое время вылета не пустое, т. е. рейсы, имеющие статус Departed или Arrived.

```
SELECT count( * )
  FROM ( ticket_flights t
         JOIN flights f ON t.flight_id = f.flight_id
       )
  LEFT OUTER JOIN boarding_passes b
    ON t.ticket_no = b.ticket_no AND t.flight_id = b.flight_id
  WHERE f.actual_departure IS NOT NULL AND b.flight_id IS NULL;
```

Оказывается, таких пассажиров нет.

```
 count
-------
     0
(1 строка)
```

При формировании запроса надо вспомнить, что таблица «Посадочные талоны» (boarding_passes) связана с таблицей «Перелеты» (ticket_flights) по внешнему ключу, а тип связи — 1:1, т. е. каждой строке из таблицы ticket_flights соответствует *не более* одной строки в таблице boarding_passes: ведь строка в таблицу boarding_passes добавляется только тогда, когда пассажир прошел регистрацию на рейс. Однако теоретически, да и практически тоже, пассажир может на регистрацию не явиться, тогда строка в таблицу boarding_passes добавлена не будет.

Поскольку нас интересуют только рейсы с непустым временем вылета, нам придется обратиться к таблице «Рейсы» (flights) и соединить ее с таблицей ticket_flights по атрибуту flight_id. А затем для подключения таблицы boarding_passes мы используем левое внешнее соединение, т. к. в этой таблице может не оказаться строки, соответствующей строке из таблицы ticket_flights.

В предложении WHERE второе условие — `b.flight_id IS NULL`. Оно и позволяет выявить те комбинированные строки, в которых столбцам таблицы `boarding_passes` были назначены значения NULL из-за того, что в ней не нашлось строки, для которой выполнялось бы условие `t.ticket_no = b.ticket_no AND t.flight_id = b.flight_id`. Конечно, для проверки на NULL мы могли использовать любой столбец таблицы `boarding_passes`, а не только `b.flight_id`.

При формировании соединений подключение таблиц выполняется слева направо, т. е. берется самая первая таблица в предложении FROM и с ней соединяется вторая таблица, затем с полученным набором строк соединяется третья таблица и т. д. Если требуется изменить порядок соединения таблиц, то могут использоваться круглые скобки. В приведенном запросе мы использовали круглые скобки для наглядности, однако в данном случае они не были обязательными. Необходимо различать описанный выше логический порядок соединения таблиц, т. е. взгляд с позиции программиста, пишущего запрос, и тот фактический порядок выполнения запроса, который будет сформирован планировщиком. Они могут различаться. Подробно о планах выполнения запросов сказано в главе 10.

Теперь рассмотрим более сложный пример. Известно, что в компьютерных системах бывают сбои. Предположим, что возможна такая ситуация: при бронировании билета пассажир выбрал один класс обслуживания, например, `Business`, а при регистрации на рейс ему выдали посадочный талон на то место в салоне самолета, где класс обслуживания — `Economy`. Необходимо выявить все случаи несовпадения классов обслуживания.

Сведения о классе обслуживания, который пассажир выбрал при бронировании билета, содержатся в таблице «Перелеты» (`ticket_flights`). Однако в таблице «Посадочные талоны» (`boarding_passes`), которая «отвечает» за посадку на рейс, сведений о классе обслуживания, который пассажир получил при регистрации, нет. Эти сведения можно получить только из таблицы «Места» (`seats`). Причем сделать это можно, зная код модели самолета, выполняющего рейс, и номер места в салоне самолета. Номер места можно взять из таблицы `boarding_passes`, а код модели самолета можно получить из таблицы «Рейсы» (`flights`), связав ее с таблицей `boarding_passes`.

Для полноты информационной картины необходимо получить еще фамилию и имя пассажира из таблицы «Билеты» (`tickets`), связав ее с таблицей `ticket_flights` по атрибуту «Номер билета» (`ticket_no`). При формировании запроса выберем в качестве, условно говоря, базовой таблицы таблицу `boarding_passes`, а затем будем поэтапно подключать остальные таблицы. В предложении WHERE будет только одно условие: несовпадение требуемого и фактического классов обслуживания.

В результате получим запрос, включающий пять таблиц. Он не выдаст ни одной строки, значит, пассажиров, получивших неправильный класс обслуживания, не было.

```
SELECT f.flight_no,
       f.scheduled_departure,
       f.flight_id,
       f.departure_airport,
       f.arrival_airport,
       f.aircraft_code,
       t.passenger_name,
       tf.fare_conditions AS fc_to_be,
       s.fare_conditions AS fc_fact,
       b.seat_no
  FROM boarding_passes b
  JOIN ticket_flights tf
    ON b.ticket_no = tf.ticket_no AND b.flight_id = tf.flight_id
  JOIN tickets t ON tf.ticket_no = t.ticket_no
  JOIN flights f ON tf.flight_id = f.flight_id
  JOIN seats s
    ON b.seat_no = s.seat_no AND f.aircraft_code = s.aircraft_code
 WHERE tf.fare_conditions <> s.fare_conditions
 ORDER BY f.flight_no, f.scheduled_departure;
```

Чтобы все же удостовериться в работоспособности этого запроса, можно в таблице `boarding_passes` изменить в одной строке номер места таким образом, чтобы этот пассажир переместился из салона экономического класса в салон бизнес-класса.

```
UPDATE boarding_passes
  SET seat_no = '1A'
  WHERE flight_id = 1 AND seat_no = '17A';
```

```
UPDATE 1
```

Выполним запрос еще раз, и теперь он выдаст одну строку.

В предложении `FROM` можно использовать виртуальные таблицы, сформированные с помощью **ключевого слова `VALUES`**. Предположим, что для выработки финансовой стратегии нашей авиакомпании требуется распределение количества бронирований по диапазонам сумм с шагом в 100 тысяч рублей. Максимальная сумма в одном бронировании составляет 1 204 500 рублей. Учтем это при формировании диапазонов.

Виртуальной таблице, создаваемой с помощью ключевого слова `VALUES`, присваивают имя с помощью ключевого слова `AS`. После имени в круглых скобках приводится список имен столбцов этой таблицы.

```
SELECT r.min_sum, r.max_sum, count( b.* )
  FROM bookings b
  RIGHT OUTER JOIN
   ( VALUES (       0,  100000 ), (  100000,  200000 ),
            (  200000,  300000 ), (  300000,  400000 ),
            (  400000,  500000 ), (  500000,  600000 ),
            (  600000,  700000 ), (  700000,  800000 ),
            (  800000,  900000 ), (  900000, 1000000 ),
            ( 1000000, 1100000 ), ( 1100000, 1200000 ),
            ( 1200000, 1300000 )
   ) AS r ( min_sum, max_sum )
   ON b.total_amount >= r.min_sum AND b.total_amount < r.max_sum
  GROUP BY r.min_sum, r.max_sum
  ORDER BY r.min_sum;
```

В этом запросе мы использовали внешнее соединение. Сделано это для того, чтобы в случаях, когда в каком-то диапазоне не окажется ни одного бронирования, результирующая строка выборки все же была бы сформирована. А правое соединение было выбрано только потому, что в качестве первой, базовой, таблицы мы выбрали таблицу «Бронирования» (`bookings`), но именно в ней может не оказаться ни одной строки для соединения с какой-либо строкой виртуальной таблицы. А все строки виртуальной таблицы, стоящей справа от предложения `RIGHT OUTER JOIN`, должны быть обязательно представлены в выборке: это позволит сразу увидеть «пустые» диапазоны, если они будут.

Можно было использовать и левое внешнее соединение, поменяв таблицы местами.

```
 min_sum | max_sum |  count
---------+---------+--------
       0 |  100000 | 198314
  100000 |  200000 |  46943
  200000 |  300000 |  11916
  300000 |  400000 |   3260
  400000 |  500000 |   1357
  500000 |  600000 |    681
  600000 |  700000 |    222
  700000 |  800000 |     55
  800000 |  900000 |     24
  900000 | 1000000 |     11
 1000000 | 1100000 |      4
 1100000 | 1200000 |      0
 1200000 | 1300000 |      1
(13 строк)
```

Обратите внимание, что для диапазона от 1 100 до 1 200 тысяч рублей значение счетчика бронирований равно нулю. Если бы мы не использовали внешнее соединение, то эта строка вообще не попала бы в выборку. Конечно, информация была бы получена та же самая, но воспринимать ее было бы сложнее.

В команде SELECT предусмотрены средства для выполнения операций с выборками, как с множествами, а именно:

- UNION для вычисления объединения множеств строк из двух выборок;
- INTERSECT для вычисления пересечения множеств строк из двух выборок;
- EXCEPT для вычисления разности множеств строк из двух выборок.

Запросы должны возвращать одинаковое число столбцов, типы данных у столбцов также должны совпадать.

Рассмотрим эти операции, используя материализованное представление «Маршруты» (routes).

Начнем с операции **объединения множеств строк — UNION**. Строка включается в итоговое множество (выборку), если она присутствует хотя бы в одном из них. Строки-дубликаты в результирующее множество не включаются. Для их включения нужно использовать UNION ALL.

Вопрос: в какие города можно улететь либо из Москвы, либо из Санкт-Петербурга?

```
SELECT arrival_city FROM routes
  WHERE departure_city = 'Москва'
UNION
SELECT arrival_city FROM routes
  WHERE departure_city = 'Санкт-Петербург'
ORDER BY arrival_city;
```

```
     arrival_city
--------------------------
 Абакан
 Анадырь
 Анапа
 ...
(87 строк)
```

Рассмотрим операцию **пересечения множеств строк — INTERSECT**. Строка включается в итоговое множество (выборку), если она присутствует в каждом из них. Строки-дубликаты в результирующее множество не включаются. Для их включения нужно использовать INTERSECT ALL.

Вопрос: в какие города можно улететь как из Москвы, так и из Санкт-Петербурга?

```
SELECT arrival_city FROM routes
  WHERE departure_city = 'Москва'
INTERSECT
SELECT arrival_city FROM routes
  WHERE departure_city = 'Санкт-Петербург'
ORDER BY arrival_city;
```

```
  arrival_city
-----------------
 Воркута
 Воронеж
 Казань
 ...
(15 строк)
```

В завершение рассмотрим операцию **разности множеств строк — EXCEPT**. Строка включается в итоговое множество (выборку), если она присутствует в первом множестве (выборке), но отсутствует во втором. Строки-дубликаты в результирующее множество не включаются. Для их включения нужно использовать EXCEPT ALL.

Вопрос: в какие города можно улететь из Санкт-Петербурга, но нельзя из Москвы?

```
SELECT arrival_city FROM routes
  WHERE departure_city = 'Санкт-Петербург'
EXCEPT
SELECT arrival_city FROM routes
  WHERE departure_city = 'Москва'
ORDER BY arrival_city;
```

```
 arrival_city
--------------
 Иркутск
 Калуга
 Москва
 ...
(7 строк)
```

Конечно, при выполнении этих операций можно соединять не только две таблицы, но и большее их число. При этом нужно либо учитывать приоритеты выполнения операций, либо использовать скобки. Согласно документации INTERSECT связывает свои подзапросы сильнее, чем UNION, а EXCEPT связывает свои подзапросы так же сильно, как UNION.

6.3. Агрегирование и группировка

Среди множества функций, имеющихся в PostgreSQL, важное место занимают агрегатные функции. С одной из них, функцией `count`, мы уже работали довольно много. Давайте рассмотрим еще ряд функций из этой группы и сделаем это на примере таблицы «Бронирования».

Для расчета среднего значения по столбцу используется функция `avg` (от слова average).

```
SELECT avg( total_amount ) FROM bookings;
```

```
       avg
--------------------
 79025.605811528685
(1 строка)
```

Для получения максимального значения по столбцу используется функция `max`.

```
SELECT max( total_amount ) FROM bookings;
```

```
    max
------------
 1204500.00
(1 строка)
```

Для получения минимального значения по столбцу используется функция `min`.

```
SELECT min( total_amount ) FROM bookings;
```

```
   min
---------
 3400.00
(1 строка)
```

Мы уже много раз выполняли **группировку строк** в выборке при помощи предложения `GROUP BY`, поэтому рассмотрим только два примера.

Первый будет таким: давайте подсчитаем, сколько маршрутов предусмотрено из Москвы в другие города. При формировании запроса не будем учитывать частоту рейсов в неделю, т. е. независимо от того, выполняется какой-то рейс один раз в неделю или семь раз, он учитывается только однократно. Воспользуемся материализованным представлением «Маршруты».

```
SELECT arrival_city, count( * )
  FROM routes
  WHERE departure_city = 'Москва'
  GROUP BY arrival_city
  ORDER BY count DESC;
```

```
 arrival_city             | count
--------------------------+-------
 Санкт-Петербург          |    12
 Брянск                   |     9
 Ульяновск                |     5
 ...
```

В качестве второго примера рассмотрим ситуацию, когда руководству компании потребовалась обобщенная информация по частоте выполнения рейсов, а именно: сколько рейсов выполняется ежедневно, сколько рейсов — шесть дней в неделю, пять и т. д. Опять обратимся к материализованному представлению «Маршруты». Но теперь при формировании запроса, в отличие от первого примера, воспользуемся столбцом `days_of_week`, в котором содержатся *массивы* номеров дней недели, когда выполняется данный рейс.

```
SELECT array_length( days_of_week, 1 ) AS days_per_week,
       count( * ) AS num_routes
  FROM routes
  GROUP BY days_per_week
  ORDER BY 1 desc;
```

```
 days_per_week | num_routes
---------------+------------
             7 |        482
             3 |         54
             2 |         88
             1 |         86
(4 строки)
```

В этом запросе используется функция `array_length`, возвращающая количество элементов в указанном измерении массива. Поскольку массив одномерный, то вторым параметром функции будет число 1 — первое измерение.

При выполнении выборок можно с помощью условий, заданных в предложении `WHERE`, сузить множество выбираемых строк. Аналогичная возможность существует и при выполнении группировок: можно включить в результирующее множество не все строки, а лишь те, которые удовлетворяют некоторому условию. Это условие

можно задать в **предложении HAVING**. Важно помнить, что предложение WHERE работает с отдельными строками еще до выполнения группировки с помощью GROUP BY, а предложение HAVING — уже после выполнения группировки.

В качестве примера приведем такой запрос: определить, сколько существует маршрутов из каждого города в другие города, и вывести названия городов, из которых в другие города существует не менее 15 маршрутов.

```
SELECT departure_city, count( * )
  FROM routes
  GROUP BY departure_city
  HAVING count( * ) >= 15
  ORDER BY count DESC;
```

```
 departure_city  | count
-----------------+-------
 Москва          |   154
 Санкт-Петербург |    35
 Новосибирск     |    19
 Екатеринбург    |    15
(4 строки)
```

В подавляющем большинстве городов только один аэропорт, но есть и такие города, в которых более одного аэропорта. Давайте их выявим.

```
SELECT city, count( * )
  FROM airports
  GROUP BY city
  HAVING count( * ) > 1;
```

```
 city      | count
-----------+-------
 Ульяновск |     2
 Москва    |     3
(2 строки)
```

Кроме обычных агрегатных функций существуют и так называемые **оконные функции (window functions)**, технология использования которых описана в документации в разделе 3.5 «Оконные функции». Эти функции предоставляют возможность производить вычисления на множестве строк, логически связанных с текущей строкой, т. е. имеющих то или иное отношение к ней.

При работе с оконными функциями используются концепции *раздела* (partition) и *оконного кадра* (window frame). Сначала объясним эти понятия на примере.

Предположим, что руководство нашей компании хочет усовершенствовать тарифную политику и с этой целью просит нас предоставить сведения о распределении количества проданных билетов на некоторые рейсы во времени. Количество проданных билетов должно выводиться в виде накопленного показателя, суммирование должно производиться в пределах каждого календарного месяца.

Более детально, в столбцах `book_ref` и `book_date` выборки должны приводиться номер и время бронирования соответственно. В столбцах `month` и `day` должны указываться порядковый номер месяца и день этого месяца. Столбец `count` должен содержать суммарные (накопленные) количества билетов, проданных на каждый момент времени. С первого дня нового месяца подсчет числа проданных билетов начинается сначала.

Таким образом, в нашем примере в качестве раздела (partition) будет выступать множество строк, у которых даты продажи билета (т. е. даты бронирования) относятся к одному и тому же месяцу. В результате в полученной выборке, пример которой приведен ниже, будет сформировано два раздела.

```
 book_ref |        book_date       | month | day | count
----------+------------------------+-------+-----+-------
 A60039   | 2016-08-22 12:02:00+08 |     8 |  22 |     1
 554340   | 2016-08-23 23:04:00+08 |     8 |  23 |     2
 854C4C   | 2016-08-24 10:52:00+08 |     8 |  24 |     5
 854C4C   | 2016-08-24 10:52:00+08 |     8 |  24 |     5
 854C4C   | 2016-08-24 10:52:00+08 |     8 |  24 |     5
 81D8AF   | 2016-08-25 10:22:00+08 |     8 |  25 |     6
 ...
 8D6873   | 2016-08-31 17:09:00+08 |     8 |  31 |    59
 E82829   | 2016-08-31 20:56:00+08 |     8 |  31 |    60
 ECA0D7   | 2016-09-01 00:48:00+08 |     9 |   1 |     1
 E3BD32   | 2016-09-01 04:44:00+08 |     9 |   1 |     2
 ...
 EB11BB   | 2016-09-03 12:02:00+08 |     9 |   3 |    14
 19FE38   | 2016-09-03 17:42:00+08 |     9 |   3 |    16
 19FE38   | 2016-09-03 17:42:00+08 |     9 |   3 |    16
 536A3D   | 2016-09-03 19:19:00+08 |     9 |   3 |    18
 536A3D   | 2016-09-03 19:19:00+08 |     9 |   3 |    18
 02E6B6   | 2016-09-04 01:39:00+08 |     9 |   4 |    19
(79 строк)
```

Здесь для примера был выбран рейс с идентификатором 1.

Понятие оконного кадра (window frame) является важным, поскольку многие оконные функции работают не со всеми строками раздела, а только с теми, которые образуют оконный кадр текущей строки. Если строки в разделе не упорядочены, то оконным кадром текущей строки по умолчанию считается множество всех строк раздела. Однако в том случае, когда строки в разделе упорядочены по какому-то критерию, тогда в состав оконного кадра по умолчанию включаются строки, начиная с первой строки раздела и заканчивая текущей строкой. Если же существуют строки, имеющие такое же значение критерия сортировки, что и текущая строка, и расположенные *после* нее, то они также включаются в состав оконного кадра текущей строки.

Обратите внимание на первые строки в представленной выборке. В строках с третьей по пятую значения в столбце `count` одинаковые и равны 5. Равенство значений имеет следующее объяснение. В рамках одного бронирования с номером `854C4C` были проданы сразу три билета на этот рейс, поэтому в этих трех строках значения в столбце `book_date` одинаковые. Строки в выборке упорядочены по значениям столбца `book_date`. Таким образом, для каждой из этих трех строк, т. е. для третьей, четвертой и пятой, значения критерия сортировки одинаковые, поэтому оконным кадром для каждой из них будут являться первые пять строк первого раздела выборки. Подсчет числа проданных билетов выполняется в пределах оконного кадра. В результате и появляется значение 5 в каждой из этих трех строк, а значений 3 и 4 нет вообще.

В приведенной выборке отражены также и случаи одновременного бронирования двух билетов на данный рейс. Вы можете найти соответствующие строки самостоятельно.

Теперь посмотрим, с помощью какого запроса был получен этот результат, и на его примере объясним синтаксические конструкции, используемые для работы с оконными функциями.

```
SELECT b.book_ref,
       b.book_date,
       extract( 'month' from b.book_date ) AS month,
       extract( 'day'   from b.book_date ) AS day,
       count( * ) OVER (
         PARTITION BY date_trunc( 'month', b.book_date )
         ORDER BY b.book_date
       ) AS count
  FROM ticket_flights tf
  JOIN tickets  t ON tf.ticket_no = t.ticket_no
  JOIN bookings b ON t.book_ref = b.book_ref
  WHERE tf.flight_id = 1
  ORDER BY b.book_date;
```

Рассмотрим конструкцию, предназначенную для вызова оконной функции:

```
count( * ) OVER (
  PARTITION BY date_trunc( 'month', b.book_date )
  ORDER BY b.book_date
) AS count
```

В этой конструкции обязательным является ключевое слово OVER. Функция count — это обычная агрегатная функция, но если вслед за ней идет это ключевое слово, то она становится оконной функцией. Предложение PARTITION BY задает правило разбиения строк выборки на разделы. Предложение ORDER BY предписывает порядок сортировки строк в разделах.

Обобщая приведенные объяснения, можно сказать, что раздел включает в себя все строки выборки, имеющие в некотором смысле одинаковые свойства, например, одинаковые значения определенных выражений, задаваемых с помощью предложения PARTITION BY. Это могут быть выражения, построенные на основе одного или нескольких столбцов таблицы (или таблиц, участвующих в соединении).

Оконный кадр состоит из подмножества строк данного раздела и привязан к текущей строке. Для определения границ кадра важным является наличие предложения ORDER BY при формировании раздела. В рассмотренном примере границы оконного кадра определялись по умолчанию. Однако для указания этих границ предусмотрены различные способы. Подробно о них сказано в разделе документации 4.2.8 «Вызовы оконных функций».

Не только функция count, но и другие агрегатные функции (например, sum, avg) тоже могут применяться в качестве оконных функций. Полный перечень собственно оконных функций приведен в документации в разделе 9.21 «Оконные функции».

Оконные функции, в отличие от обычных агрегатных функций, не требуют группировки строк, а работают на уровне отдельных (несгруппированных) строк. Однако если в запросе присутствуют предложения GROUP BY и HAVING, тогда оконные функции вызываются уже *после* них. В таком случае оконные функции будут работать со строками, являющимися результатом группировки.

Рассмотрим еще один пример. Покажем, как с помощью оконной функции rank можно проранжировать аэропорты в пределах каждого часового пояса на основе их географической широты. Причем будем присваивать более высокий ранг тому аэропорту, который находится севернее.

```
SELECT airport_name,
       city,
       round( latitude::numeric, 2 ) AS ltd,
       timezone,
       rank() OVER (
         PARTITION BY timezone
         ORDER BY latitude DESC
       )
  FROM airports
  WHERE timezone IN ( 'Asia/Irkutsk', 'Asia/Krasnoyarsk' )
  ORDER BY timezone, rank;
```

В этом запросе в предложении OVER (PARTITION BY timezone ...) указывается, что строки относятся к одному разделу на основе совпадения значений в столбце timezone. Обратите внимание, что хотя в предложении OVER задан порядок сортировки, действующий в пределах каждого окна, тем не менее, с помощью предложения ORDER BY указан также и порядок сортировки на уровне всего запроса.

```
 airport_name  |      city     |  ltd  |    timezone      | rank
---------------+---------------+-------+------------------+------
 Усть-Илимск   | Усть-Илимск   | 58.14 | Asia/Irkutsk     |    1
 Усть-Кут      | Усть-Кут      | 56.85 | Asia/Irkutsk     |    2
 Братск        | Братск        | 56.37 | Asia/Irkutsk     |    3
 Иркутск       | Иркутск       | 52.27 | Asia/Irkutsk     |    4
 ...
 Абакан        | Абакан        | 53.74 | Asia/Krasnoyarsk |    5
 Барнаул       | Барнаул       | 53.36 | Asia/Krasnoyarsk |    6
 Горно-Алтайск | Горно-Алтайск | 51.97 | Asia/Krasnoyarsk |    7
 Кызыл         | Кызыл         | 51.67 | Asia/Krasnoyarsk |    8
(13 строк)
```

Усложним запрос — для каждого аэропорта будем вычислять разницу между его географической широтой и широтой, на которой находится самый северный аэропорт в этом же часовом поясе. Поскольку в запросе используются три конструкции с оконными функциями и при этом способ формирования разделов и порядок сортировки строк в разделах один и тот же, то вводится предложение WINDOW. Оно позволяет создать определение раздела, а затем ссылаться на него при вызове оконных функций. Самый северный аэропорт в каждом часовом поясе, т. е. самая первая строка в каждом разделе, выбирается с помощью оконной функции first_value. Строго говоря, эта функция получает доступ к первой строке оконного кадра, а не раздела. Однако когда используются правила формирования оконного кадра по умолчанию, тогда его начало совпадает с началом раздела.

Обратите внимание, что в этом запросе в каждой конструкции OVER используется ссылка на одно и то же окно, т. е. имеет место один и тот же порядок разбиения на разделы и сортировки строк, поэтому данные будут обработаны за один проход по таблице.

```
SELECT airport_name, city, timezone, latitude,
       first_value( latitude )            OVER tz AS first_in_timezone,
       latitude - first_value( latitude ) OVER tz AS delta,
       rank()                             OVER tz
  FROM airports
  WHERE timezone IN ( 'Asia/Irkutsk', 'Asia/Krasnoyarsk' )
  WINDOW tz AS ( PARTITION BY timezone ORDER BY latitude DESC )
  ORDER BY timezone, rank;
```

```
...
--[ RECORD 5 ]-----+-----------------
 airport_name      | Байкал
 city              | Улан-Удэ
 timezone          | Asia/Irkutsk
 latitude          | 51.807764
 first_in_timezone | 58.135
 delta             | -6.327236
 rank              | 5
--[ RECORD 6 ]-----+-----------------
 airport_name      | Норильск
 city              | Норильск
 timezone          | Asia/Krasnoyarsk
 latitude          | 69.311053
 first_in_timezone | 69.311053
 delta             | 0
 rank              | 1
...
```

Более подробно использование оконных функций описано в документации. Мы рекомендуем начать с раздела 3.5 «Оконные функции», в котором приводятся примеры их использования. В разделе 9.21 «Оконные функции» приводятся описания всех оконных функций, предлагаемых PostgreSQL. В разделе 4.2.8 «Вызовы оконных функций» детально рассматривается синтаксис вызова оконных функций. В разделе 7.2.5 «Обработка оконных функций» говорится о том, на каком этапе выполнения запроса производится обработка этих функций.

6.4. Подзапросы

Прежде чем приступить к рассмотрению столь сложной темы, как подзапросы, опишем, как в общем случае работает команда SELECT. Согласно описанию этой команды, приведенному в документации, дело, в несколько упрощенном виде, обстоит так.

1. Сначала вычисляются все элементы, приведенные в списке после ключевого слова FROM. Под такими элементами подразумеваются не только реальные таблицы, но также и виртуальные таблицы, создаваемые с помощью ключевого слова VALUES. Если таблиц больше одной, то формируется декартово произведение из множеств их строк. Например, в случае двух таблиц будут сформированы попарные комбинации каждой строки из одной таблицы с каждой строкой из другой таблицы. При этом в комбинированных строках сохраняются все атрибуты из каждой исходной таблицы.

2. Если в команде присутствует условие WHERE, то из полученного декартова произведения исключаются строки, которые этому условию не соответствуют. Таким образом, первоначальное множество строк, сформированное без всяких условий, сужается.

3. Если присутствует предложение GROUP BY, то результирующие строки группируются на основе совпадения значений одного или нескольких атрибутов, а затем вычисляются значения агрегатных функций. Если присутствует предложение HAVING, то оно отфильтровывает результирующие строки (группы), не удовлетворяющие критерию.

4. Ключевое слово SELECT присутствует всегда. Но в списке выражений, идущих после него, могут быть не только простые имена атрибутов, но и их комбинации, созданные с использованием арифметических и других операций, а также вызовы функций. Причем эти функции могут быть не только встроенные, но и созданные пользователем. В списке выражений не обязаны присутствовать *все* атрибуты, представленные в строках используемых таблиц. Например, атрибуты, на основе которых формируются условия в предложении WHERE, могут отсутствовать в списке выражений после ключевого слова SELECT. Предложение SELECT DISTINCT удаляет дубликаты строк.

5. Если присутствует предложение ORDER BY, то результирующие строки сортируются на основе значений одного или нескольких атрибутов. По умолчанию сортировка производится по возрастанию значений.

6. Если присутствует предложение LIMIT или OFFSET, то возвращается только подмножество строк из выборки.

Приведенная схема описывает работу команды SELECT на логическом уровне, а на уровне реализации запросов в дело вступает планировщик, который и формирует план выполнения запроса.

А теперь перейдем непосредственно к теме этого раздела — подзапросам.

Предположим, что сотрудникам аналитического отдела потребовалось провести статистическое исследование финансовых результатов работы авиакомпании. В качестве первого шага они решили подсчитать количество операций бронирования, в которых общая сумма превышает среднюю величину по всей выборке.

```
SELECT count( * ) FROM bookings
  WHERE total_amount >
    ( SELECT avg( total_amount ) FROM bookings );
```

```
 count
-------
 87224
(1 строка)
```

В приведенном запросе присутствует два предложения SELECT, но при этом только одно из них является главным в этом запросе, а другое представляет собой **подзапрос**. Он заключается в круглые скобки и является частью более общего запроса. Подзапросы могут присутствовать в предложениях SELECT, FROM, WHERE и HAVING, а также в предложении WITH, о котором мы расскажем позднее.

В приведенном примере в предложении WHERE используется так называемый **скалярный подзапрос**. Это означает, что в результате его выполнения возвращается только одно скалярное значение (один столбец и одна строка), с которым можно сравнивать другие скалярные значения.

Если подзапрос выдает множество скалярных значений (или даже только одно), можно использовать такой **подзапрос в предикате IN**. Этот предикат позволяет организовать проверку на предмет принадлежности какого-либо значения определенному множеству значений.

В качестве примера давайте выясним, какие маршруты существуют между городами часового пояса Asia/Krasnoyarsk. Подзапрос будет выдавать список городов из этого часового пояса, а в предложении WHERE главного запроса с помощью предиката IN будет выполняться проверка на принадлежность города этому списку. При этом подзапрос выполняется *только один раз* для всего внешнего запроса, а не при обработке каждой строки из таблицы routes во внешнем запросе. Повторного выполнения подзапроса не требуется, т. к. его результат не зависит от значений, хранящихся в таблице routes. Такие подзапросы называются **некоррелированными**.

```
SELECT flight_no, departure_city, arrival_city
  FROM routes
  WHERE departure_city IN (
      SELECT city
        FROM airports
        WHERE timezone ~ 'Krasnoyarsk'
    )
    AND arrival_city IN (
      SELECT city
        FROM airports
        WHERE timezone ~ 'Krasnoyarsk'
    );
```

```
 flight_no | departure_city | arrival_city
-----------+----------------+--------------
 PG0070    | Абакан         | Томск
 PG0071    | Томск          | Абакан
 PG0313    | Абакан         | Кызыл
 PG0314    | Кызыл          | Абакан
 PG0653    | Красноярск     | Барнаул
 PG0654    | Барнаул        | Красноярск
(6 строк)
```

Можно сформировать множество значений для предиката `IN` с помощью скалярных подзапросов. Если мы захотим найти самый западный и самый восточный аэропорты и представить полученные сведения в наглядной форме, то запрос может быть таким:

```
SELECT airport_name, city, longitude
  FROM airports
  WHERE longitude IN (
   ( SELECT max( longitude ) FROM airports ),
   ( SELECT min( longitude ) FROM airports )
  )
  ORDER BY longitude;
```

```
 airport_name |    city     | longitude
--------------+-------------+------------
 Храброво     | Калининград |  20.592633
 Анадырь      | Анадырь     | 177.741483
(2 строки)
```

Конечно, в случае, когда необходимо, наоборот, исключить какие-либо значения из рассмотрения, можно использовать конструкцию `NOT IN`.

Иногда возникают ситуации, когда от подзапроса требуется лишь установить сам факт наличия или отсутствия строк в конкретной таблице, удовлетворяющих определенному условию, а непосредственные значения атрибутов в этих строках интереса не представляют. В подобных случаях используют **предикат EXISTS** (или NOT EXISTS).

В качестве примера выясним, в какие города нет рейсов из Москвы.

```
SELECT DISTINCT a.city
  FROM airports a
  WHERE NOT EXISTS (
    SELECT * FROM routes r
      WHERE r.departure_city = 'Москва'
        AND r.arrival_city = a.city
    )
    AND a.city <> 'Москва'
  ORDER BY city;
```

В этом запросе мы не можем ограничиться только лишь материализованным представлением «Маршруты» (routes), поскольку в нем представлены лишь *существующие* маршруты. Полный список городов можно найти в таблице «Аэропорты» (airports). Для каждой строки (каждого города) из таблицы airports выполняется поиск строки в представлении routes, в которой значение атрибута arrival_city такое же, как в текущей строке таблицы airports. Если такой строки не найдено, значит, в этот город маршрута из Москвы нет.

Поскольку от подзапроса в предикате EXISTS требуется только установить факт наличия или отсутствия строк, соответствующих критерию отбора, то в документации рекомендуется вместо списка столбцов (или символа «*») в предложении SELECT делать так:

```
WHERE NOT EXISTS ( SELECT 1 FROM routes r ...
```

Обратите внимание на ключевое слово DISTINCT в запросе. Оно необходимо, т. к. кроме Москвы могут быть другие города, в которых есть более одного аэропорта. Один такой город уже существует — Ульяновск. Если не использовать DISTINCT, то в принципе возможно появление строк-дубликатов в выборке.

И еще одна важная деталь. В представленном запросе мы использовали так называемый **коррелированный (связанный) подзапрос**. В подзапросах такого типа присутствует ссылка (ссылки) на таблицу из внешнего запроса, как здесь:

```
WHERE ...
  AND r.arrival_city = a.city
```

В теории это означает, что подзапрос выполняется не один раз для всего внешнего запроса, а *для каждой строки*, обрабатываемой во внешнем запросе. Однако на практике важную роль играет умение планировщика (это специальная подсистема в СУБД) оптимизировать подобные запросы с тем, чтобы по возможности избегать выполнения подзапроса для каждой строки из внешнего запроса.

Получаем такой результат:

```
      city
----------------------
 Благовещенск
 Иваново
 ...
 Якутск
 Ярославль
(20 строк)
```

Рассмотрим использование подзапросов в предложениях SELECT, FROM и HAVING.

Предположим, что для выработки ценовой политики авиакомпании необходимо знать, как распределяются места разных классов в самолетах всех типов. Первый вариант решения этой задачи основан на включении **подзапросов в предложение SELECT**.

```
SELECT a.model,
  ( SELECT count( * )
      FROM seats s
      WHERE s.aircraft_code = a.aircraft_code
        AND s.fare_conditions = 'Business'
  ) AS business,
  ( SELECT count( * )
      FROM seats s
      WHERE s.aircraft_code = a.aircraft_code
        AND s.fare_conditions = 'Comfort'
  ) AS comfort,
  ( SELECT count( * )
      FROM seats s
      WHERE s.aircraft_code = a.aircraft_code
        AND s.fare_conditions = 'Economy'
  ) AS economy
  FROM aircrafts a
  ORDER BY 1;
```

Обратите внимание, что в этом запросе мы использовали коррелированные подзапросы. Все они ссылаются на столбец таблицы «Самолеты» (`aircrafts`), которая обрабатывается во внешнем запросе. Для каждой обрабатываемой строки таблицы `aircrafts` подсчитывается число строк в таблице `seats`, в которых атрибут `aircraft_code` имеет такое же значение, что и в строке таблицы `aircrafts`. Подзапросы отличаются друг от друга только условием `fare_conditions`.

Поскольку все эти подзапросы не зависят друг от друга, то, хотя все они обращаются к таблице «Места» (`seats`), не требуется использовать для нее различные псевдонимы в этих подзапросах.

```
       model          | business | comfort | economy
----------------------+----------+---------+---------
 Airbus A319-100      |       20 |       0 |      96
 Airbus A320-200      |       20 |       0 |     120
 Airbus A321-200      |       28 |       0 |     142
 Boeing 737-300       |       12 |       0 |     118
 Boeing 767-300       |       30 |       0 |     192
 Boeing 777-300       |       30 |      48 |     324
 Bombardier CRJ-200   |        0 |       0 |      50
 Cessna 208 Caravan   |        0 |       0 |      12
 Sukhoi SuperJet-100  |       12 |       0 |      85
(9 строк)
```

А в этом варианте решения задачи используется **подзапрос в предложении `FROM`**.

```
SELECT s2.model,
  string_agg(
     s2.fare_conditions || ' (' || s2.num || ')',
     ', '
  )
  FROM (
    SELECT a.model,
        s.fare_conditions,
        count( * ) AS num
      FROM aircrafts a
      JOIN seats s ON a.aircraft_code = s.aircraft_code
      GROUP BY 1, 2
      ORDER BY 1, 2
  ) AS s2
  GROUP BY s2.model
  ORDER BY s2.model;
```

Подзапрос формирует временную таблицу в таком виде:

```
       model          | fare_conditions | num
----------------------+-----------------+-----
 Airbus A319-100      | Business        |  20
 Airbus A319-100      | Economy         |  96
 ...
 Sukhoi SuperJet-100 | Business        |  12
 Sukhoi SuperJet-100 | Economy         |  85
(17 строк)
```

А в главном (внешнем) запросе используется агрегатная функция `string_agg` для формирования результирующего значения на основе сгруппированных строк. Эта функция отличается от агрегатных функций `avg`, `min`, `max`, `sum` и `count` тем, что возвращает не числовое значение, а строку символов, составленную из значений атрибутов, указанных в качестве ее параметров. Эти значения берутся из сгруппированных строк.

```
        model         |           string_agg
----------------------+-----------------------------------------
 Airbus A319-100      | Business (20), Economy (96)
 Airbus A320-200      | Business (20), Economy (120)
 Airbus A321-200      | Business (28), Economy (142)
 Boeing 737-300       | Business (12), Economy (118)
 Boeing 767-300       | Business (30), Economy (192)
 Boeing 777-300       | Business (30), Comfort (48), Economy (324)
 Bombardier CRJ-200   | Economy (50)
 Cessna 208 Caravan   | Economy (12)
 Sukhoi SuperJet-100 | Business (12), Economy (85)
(9 строк)
```

В качестве еще одного примера использования подзапроса в предложении `FROM` решим такую задачу: получить перечень аэропортов в тех городах, в которых больше одного аэропорта.

```
SELECT aa.city, aa.airport_code, aa.airport_name
  FROM (
    SELECT city, count( * )
      FROM airports
      GROUP BY city
      HAVING count( * ) > 1
    ) AS a
  JOIN airports AS aa ON a.city = aa.city
  ORDER BY aa.city, aa.airport_name;
```

Благодаря использованию предложения HAVING, подзапрос выбирает города, в которых имеется более одного аэропорта, и формирует временную таблицу в следующем виде:

```
   city    | count
-----------+-------
 Ульяновск |     2
 Москва    |     3
(2 строки)
```

А в главном запросе выполняется соединение временной таблицы с таблицей «Аэропорты» (airports).

```
    city   | airport_code |    airport_name
-----------+--------------+---------------------
 Москва    | VKO          | Внуково
 Москва    | DME          | Домодедово
 Москва    | SVO          | Шереметьево
 Ульяновск | ULV          | Баратаевка
 Ульяновск | ULY          | Ульяновск-Восточный
(5 строк)
```

Для иллюстрации использования **подзапросов в предложении HAVING** решим такую задачу: определить число маршрутов, исходящих из тех аэропортов, которые расположены восточнее географической долготы 150°.

```
SELECT departure_airport, departure_city, count( * )
  FROM routes
  GROUP BY departure_airport, departure_city
  HAVING departure_airport IN (
    SELECT airport_code
      FROM airports
      WHERE longitude > 150
  )
  ORDER BY count DESC;
```

Подзапрос формирует список аэропортов, которые и будут отобраны с помощью предложения HAVING после выполнения группировки.

```
 departure_airport |      departure_city      | count
-------------------+--------------------------+-------
 DYR               | Анадырь                  |     4
 GDX               | Магадан                  |     3
 PKC               | Петропавловск-Камчатский |     1
(3 строки)
```

В сложных запросах могут использоваться **вложенные подзапросы**. Это означает, что один подзапрос находится внутри другого. Давайте в качестве примера рассмотрим такую ситуацию: руководство авиакомпании хочет выяснить степень заполнения самолетов на всех рейсах, ведь отправлять полупустые самолеты не очень выгодно. Таким образом, запрос должен не только выдавать число билетов, проданных на данный рейс, и общее число мест в самолете, но должен также вычислять отношение этих двух показателей.

Вот какой запрос получился:

```
SELECT ts.flight_id,
       ts.flight_no,
       ts.scheduled_departure_local,
       ts.departure_city,
       ts.arrival_city,
       a.model,
       ts.fact_passengers,
       ts.total_seats,
       round( ts.fact_passengers::numeric /
              ts.total_seats::numeric, 2 ) AS fraction
  FROM (
    SELECT f.flight_id,
           f.flight_no,
           f.scheduled_departure_local,
           f.departure_city,
           f.arrival_city,
           f.aircraft_code,
           count( tf.ticket_no ) AS fact_passengers,
           ( SELECT count( s.seat_no )
               FROM seats s
               WHERE s.aircraft_code = f.aircraft_code
           ) AS total_seats
      FROM flights_v f
      JOIN ticket_flights tf ON f.flight_id = tf.flight_id
      WHERE f.status = 'Arrived'
      GROUP BY 1, 2, 3, 4, 5, 6
  ) AS ts
  JOIN aircrafts AS a ON ts.aircraft_code = a.aircraft_code
  ORDER BY ts.scheduled_departure_local;
```

Самый внутренний подзапрос — `total_seats` — выдает общее число мест в самолете. Этот подзапрос — коррелированный, т. к. он выполняется для каждой строки,

обрабатываемой во внешнем подзапросе, т. е. для каждой модели самолета. Для подсчета числа проданных билетов мы использовали соединение представления «Рейсы» (`flights_v`) с таблицей «Перелеты» (`ticket_flights`) с последующей группировкой строк и вызовом функции `count`. Конечно, можно было бы вместо такого решения использовать еще один коррелированный подзапрос:

```
( SELECT count( tf.ticket_no )
    FROM ticket_flights tf
    WHERE tf.flight_id = f.flight_id
) AS fact_passengers
```

В таком случае уже не потребовалось бы соединять представление `flights_v` с таблицей `ticket_flights` и группировать строки, достаточно было бы сделать:

```
  FROM flights_v
  WHERE f.status = 'Arrived'
) AS ts JOIN aircrafts AS a
```

Внешний запрос вместо кода самолета выводит наименование модели, выбирает остальные столбцы из подзапроса без изменений и дополнительно производит вычисление степени заполнения самолета пассажирами, а также сортирует результирующие строки.

```
-[ RECORD 1 ]--------------+-------------------------
 flight_id                 | 28205
 flight_no                 | PG0032
 scheduled_departure_local | 2016-09-13 08:00:00
 departure_city            | Пенза
 arrival_city              | Москва
 model                     | Cessna 208 Caravan
 fact_passengers           | 2
 total_seats               | 12
 fraction                  | 0.17
-[ RECORD 2 ]--------------+-------------------------
 flight_id                 | 9467
 flight_no                 | PG0360
 scheduled_departure_local | 2016-09-13 08:00:00
 departure_city            | Санкт-Петербург
 arrival_city              | Оренбург
 model                     | Bombardier CRJ-200
 fact_passengers           | 6
 total_seats               | 50
 fraction                  | 0.12
...
```

Рассмотренный сложный запрос можно сделать более наглядным за счет выделения подзапроса в отдельную конструкцию, которая называется **общее табличное выражение (Common Table Expression — CTE)**.

```
WITH ts AS
( SELECT f.flight_id,
         f.flight_no,
         f.scheduled_departure_local,
         f.departure_city,
         f.arrival_city,
         f.aircraft_code,
         count( tf.ticket_no ) AS fact_passengers,
         ( SELECT count( s.seat_no )
             FROM seats s
             WHERE s.aircraft_code = f.aircraft_code
         ) AS total_seats
    FROM flights_v f
    JOIN ticket_flights tf ON f.flight_id = tf.flight_id
    WHERE f.status = 'Arrived'
    GROUP BY 1, 2, 3, 4, 5, 6
)
SELECT ts.flight_id,
       ts.flight_no,
       ts.scheduled_departure_local,
       ts.departure_city,
       ts.arrival_city,
       a.model,
       ts.fact_passengers,
       ts.total_seats,
       round( ts.fact_passengers::numeric /
              ts.total_seats::numeric, 2 ) AS fraction
  FROM ts
  JOIN aircrafts AS a ON ts.aircraft_code = a.aircraft_code
  ORDER BY ts.scheduled_departure_local;
```

Конструкция `WITH ts AS (...)` и представляет собой общее табличное выражение (CTE). Такие конструкции удобны тем, что позволяют упростить основной запрос, сделать его менее громоздким. В общем табличном выражении может присутствовать больше одного подзапроса. Каждый подзапрос формирует временную таблицу с указанным именем. Если имена столбцов этой таблицы не заданы явным образом в виде списка, тогда они определяются на основе списка столбцов в предложении `SELECT`. В нашем примере это будет именно так. Теперь мы можем в главном запросе обращаться к временной таблице `ts` так, как если бы она существовала постоянно.

Но важно учитывать, что временная таблица, создаваемая в общем табличном выражении, существует только во время выполнения запроса.

В этой главе мы уже решали задачу распределения сумм бронирований по диапазонам с шагом в 100 тысяч рублей. Тогда мы использовали предложение VALUES для формирования виртуальной таблицы. Можно решить эту задачу более рациональным способом с использованием конструкции `WITH ... AS (...)`.

Сначала покажем, как можно сформировать диапазоны сумм бронирований с помощью **рекурсивного общего табличного выражения**:

```
WITH RECURSIVE ranges ( min_sum, max_sum ) AS
  ( VALUES ( 0, 100000 )
    UNION ALL
    SELECT min_sum + 100000, max_sum + 100000
      FROM ranges
      WHERE max_sum <
       ( SELECT max( total_amount ) FROM bookings )
  )
SELECT * FROM ranges;
```

В этом примере мы явно указали имена столбцов для временной таблицы `ranges` — это `min_sum` и `max_sum`. Рекурсивный алгоритм работает следующим образом:

– сначала выполняется предложение VALUES `(0, 100000)` и результат записывается во временную область памяти;

– затем к этой временной области памяти применяется запрос

```
SELECT min_sum + 100000, max_sum + 100000
...
```

и в результате его выполнения формируется только одна строка, поскольку в исходном предложении VALUES была сформирована только одна строка и только одна строка была помещена во временную область памяти;

– вновь сформированная строка вместе с исходной строкой помещаются в другую временную область, в которой происходит накапливание результирующих строк;

– к той строке, которая была на предыдущем шаге сформирована с помощью команды SELECT, опять применяется эта же команда и т. д.;

– работа завершится, когда перестанет выполняться условие

```
max_sum < ( SELECT max( total_amount ) FROM bookings )
```

Важную роль в этом процессе играет предложение UNION ALL, благодаря которому происходит объединение сформированных строк в единую таблицу. Поскольку в нашем примере в рекурсивном алгоритме участвует только одна строка, то строк-дубликатов не возникает, поэтому мы используем не UNION, а UNION ALL. При использовании предложения UNION выполняется устранение строк-дубликатов, но в данном случае необходимости в выполнении этой операции нет, следовательно, целесообразно использовать именно UNION ALL.

Получим такую таблицу:

```
 min_sum | max_sum
---------+--------
       0 |  100000
  100000 |  200000
  200000 |  300000
 ...
 1000000 | 1100000
 1100000 | 1200000
 1200000 | 1300000
(13 строк)
```

Здесь в предложении WHERE используется скалярный подзапрос. С результатом его выполнения сравнивается значение столбца max_sum:

```
WHERE max_sum < ( SELECT max( total_amount ) FROM bookings )
```

Теперь давайте скомбинируем рекурсивное общее табличное выражение с выборкой из таблицы bookings:

```
WITH RECURSIVE ranges ( min_sum, max_sum ) AS
  ( VALUES( 0, 100000 )
    UNION ALL
    SELECT min_sum + 100000, max_sum + 100000
      FROM ranges
      WHERE max_sum <
        ( SELECT max( total_amount ) FROM bookings )
  )
SELECT r.min_sum, r.max_sum, count( b.* )
  FROM bookings b
  RIGHT OUTER JOIN ranges r
    ON b.total_amount >= r.min_sum
    AND b.total_amount < r.max_sum
  GROUP BY r.min_sum, r.max_sum
  ORDER BY r.min_sum;
```

```
 min_sum | max_sum | count
---------+---------+--------
       0 |  100000 | 198314
  100000 |  200000 |  46943
  200000 |  300000 |  11916
  300000 |  400000 |   3260
  400000 |  500000 |   1357
  500000 |  600000 |    681
  600000 |  700000 |    222
  700000 |  800000 |     55
  800000 |  900000 |     24
  900000 | 1000000 |     11
 1000000 | 1100000 |      4
 1100000 | 1200000 |      0
 1200000 | 1300000 |      1
(13 строк)
```

Обратите внимание, что для диапазона от 1 100 до 1 200 тысяч рублей значение числа бронирований равно нулю. Для того чтобы была выведена строка с нулевым значением столбца `count`, мы использовали внешнее соединение.

В заключение рассмотрим команду для создания материализованного представления «Маршруты» (`routes`), которое было описано в главе 5. Но тогда мы не стали рассматривать эту команду, т. к. еще не ознакомились с подзапросами, которые в ней используются.

Описание атрибута	**Имя атрибута**	**Тип PostgreSQL**
Номер рейса	flight_no	char(6)
Код аэропорта отправления	departure_airport	char(3)
Название аэропорта отправления	departure_airport_name	text
Город отправления	departure_city	text
Код аэропорта прибытия	arrival_airport	char(3)
Название аэропорта прибытия	arrival_airport_name	text
Город прибытия	arrival_city	text
Код самолета, IATA	aircraft_code	char(3)
Продолжительность полета	duration	interval
Дни недели, когда выполняются рейсы	days_of_week	integer[]

Эта команда выглядит так:

```
CREATE MATERIALIZED VIEW routes AS
  WITH f3 AS
    (
      SELECT f2.flight_no,
             f2.departure_airport,
             f2.arrival_airport,
             f2.aircraft_code,
             f2.duration,
             array_agg( f2.days_of_week ) AS days_of_week
        FROM
        (      SELECT f1.flight_no,
                      f1.departure_airport,
                      f1.arrival_airport,
                      f1.aircraft_code,
                      f1.duration,
                      f1.days_of_week
                 FROM
                 (     SELECT flights.flight_no,
                              flights.departure_airport,
                              flights.arrival_airport,
                              flights.aircraft_code,
                            ( flights.scheduled_arrival -
                              flights.scheduled_departure
                            ) AS duration,
                            ( to_char( flights.scheduled_departure,
                                       'ID'::text
                                     )
                            )::integer AS days_of_week
                       FROM flights
                 ) f1
                 GROUP BY f1.flight_no, f1.departure_airport,
                          f1.arrival_airport, f1.aircraft_code,
                          f1.duration, f1.days_of_week
                 ORDER BY f1.flight_no, f1.departure_airport,
                          f1.arrival_airport, f1.aircraft_code,
                          f1.duration, f1.days_of_week
        ) f2
        GROUP BY f2.flight_no, f2.departure_airport,
                 f2.arrival_airport, f2.aircraft_code,
                 f2.duration
    )
```

```
SELECT f3.flight_no,
       f3.departure_airport,
       dep.airport_name AS departure_airport_name,
       dep.city AS departure_city,
       f3.arrival_airport,
       arr.airport_name AS arrival_airport_name,
       arr.city AS arrival_city,
       f3.aircraft_code,
       f3.duration,
       f3.days_of_week
  FROM f3,
       airports dep,
       airports arr
  WHERE f3.departure_airport = dep.airport_code
    AND f3.arrival_airport   = arr.airport_code;
```

Начнем ознакомление с запросом с его верхней части. Здесь мы видим конструкцию `WITH f3 AS (...)`, т. е. общее табличное выражение. В результате его выполнения будет сформирована временная таблица `f3`. Запрос, который ее формирует, содержит в предложении `FROM` подзапрос, формирующий временную таблицу `f2`. А этот подзапрос, в свою очередь, также содержит в предложении `FROM` подзапрос, формирующий временную таблицу `f1`. Таким образом, в этой команде используется вложенный подзапрос.

Во вложенном подзапросе используется функция `to_char`. Второй ее параметр — `ID` — указывает на то, что из значения даты/времени вылета будет извлечен номер дня недели. При этом нумерация дней недели соответствует стандарту ISO 8601: понедельник — 1, воскресенье — 7. Поскольку номер дня недели представлен в виде символьной строки, он преобразуется в тип данных `integer`. Таким образом, вложенный подзапрос вычисляет плановую длительность полета (столбец `duration`) и извлекает номер дня недели из даты/времени вылета по расписанию (столбец `days_of_week`).

Подзапрос следующего, более высокого уровня, получив результат вложенного подзапроса, просто группирует строки, готовя столбец `days_of_week` к объединению отдельных номеров дней недели в массивы целых чисел. При этом в предложение `GROUP BY` включен столбец `days_of_week`, чтобы заменить дубликаты дней недели одним значением. Ведь таблица `flights` содержит расписание рейсов на длительный период. Поэтому рейс, который отправляется, скажем, по вторникам, появится в этом расписании несколько раз, следовательно, день недели с номером 2 также появится в столбце `days_of_week` для этого номера рейса несколько раз. В результате,

если не прибегнуть к группировке по этому столбцу, то при формировании массива дней недели в этом массиве будут многократные вхождения каждого дня недели, когда этот рейс летает. В этом подзапросе присутствует и предложение ORDER BY, в которое включен столбец days_of_week. Это необходимо для того, чтобы агрегатная функция array_agg собрала номера дней недели в массив в возрастающем порядке этих номеров.

Во внешнем запросе вызывается функция array_agg, которая агрегирует номера дней недели, содержащиеся в сгруппированных строках, в массивы целых чисел. На этом работа конструкции WITH f3 AS (...) завершается. В результате вместо нескольких строк в таблице flights, соответствующих вылетам конкретного рейса в различные дни недели, формируется одна строка в представлении routes, в этой строке все дни недели, в которые выполняется конкретный рейс, собраны в массив целых чисел.

И, наконец, главный запрос выполняет соединение временной таблицы f3 с таблицей «Аэропорты» (airports), причем дважды. Это нужно потому, что в таблице f3 есть столбец f3.departure_airport (аэропорт отправления) и столбец f3.arrival_airport (аэропорт прибытия), для каждого из них нужно выбрать наименование аэропорта и наименование города из таблицы airports. О том, как нужно рассуждать при двукратном использовании одной и той же таблицы в соединении, мы уже говорили ранее в разделе 5.4 «Представления».

Контрольные вопросы и задания

1. В документации сказано, что служебный символ «%» в шаблоне оператора LIKE соответствует любой последовательности символов, в том числе и пустой последовательности, однако ничего не сказано насчет правил обработки пробелов.

   В таблице «Билеты» (tickets) столбец passenger_name содержит имя и фамилию пассажира, записанные заглавными латинскими буквами и разделенные одним пробелом.

   Выясните правила обработки пробелов самостоятельно, выполнив следующие команды и сравнив полученные результаты:

```
SELECT count( * ) FROM tickets;
SELECT count( * ) FROM tickets WHERE passenger_name LIKE '% %';
SELECT count( * ) FROM tickets WHERE passenger_name LIKE '% % %';
SELECT count( * ) FROM tickets WHERE passenger_name LIKE '% %%';
```

2. Этот запрос выбирает из таблицы «Билеты» (`tickets`) всех пассажиров с именами, состоящими из трех букв (в шаблоне присутствуют три символа «_»):

```
SELECT passenger_name
  FROM tickets
  WHERE passenger_name LIKE '___ %';
```

   Предложите шаблон поиска в операторе `LIKE` для выбора из этой таблицы всех пассажиров с фамилиями, состоящими из пяти букв.

3. В разделе документации 9.7.2 «Регулярные выражения SIMILAR TO» рассматривается оператор `SIMILAR TO`. Он работает аналогично оператору `LIKE`, но использует шаблоны, соответствующие определению регулярных выражений, приведенному в стандарте SQL. Регулярные выражения SQL представляют собой комбинацию синтаксиса `LIKE` с синтаксисом обычных регулярных выражений. Самостоятельно ознакомьтесь с оператором `SIMILAR TO`.

4. В разделе документации 9.2 «Функция и операторы сравнения» представлены различные предикаты сравнения, кроме предиката `BETWEEN`, рассмотренного в этой главе. Самостоятельно ознакомьтесь с ними.

5. В разделе документации 9.17 «Условные выражения» представлены условные выражения, которые поддерживаются в PostgreSQL. В тексте главы была рассмотрена конструкция `CASE`. Самостоятельно ознакомьтесь с функциями `COALESCE`, `NULLIF`, `GREATEST` и `LEAST`.

6. Выясните, на каких маршрутах используются самолеты компании Boeing. В выборке вместо кода модели должно выводиться ее наименование, например, вместо кода 733 должно быть Boeing 737-300.

   Указание: можно воспользоваться соединением представления «Маршруты» (`routes`) и таблицы «Самолеты» (`aircrafts`).

7. Самые крупные самолеты в нашей авиакомпании — это Boeing 777-300. Выяснить, между какими парами городов они летают, поможет запрос:

```
SELECT DISTINCT departure_city, arrival_city
  FROM routes r
  JOIN aircrafts a ON r.aircraft_code = a.aircraft_code
  WHERE a.model = 'Boeing 777-300'
  ORDER BY 1;
```

```
 departure_city | arrival_city
----------------+--------------
 Екатеринбург   | Москва
 Москва         | Екатеринбург
 Москва         | Новосибирск
 Москва         | Пермь
 Москва         | Сочи
 Новосибирск    | Москва
 Пермь          | Москва
 Сочи           | Москва
(8 строк)
```

К сожалению, в этой выборке информация дублируется. Пары городов приведены по два раза: для рейса «туда» и для рейса «обратно». Модифицируйте запрос таким образом, чтобы каждая пара городов была выведена только один раз:

```
 departure_city | arrival_city
----------------+--------------
 Москва         | Екатеринбург
 Новосибирск    | Москва
 Пермь          | Москва
 Сочи           | Москва
(4 строки)
```

8. В тексте главы мы рассматривали различные примеры использования левого и правого внешних соединений: `LEFT OUTER JOIN` и `RIGHT OUTER JOIN`. Напишите запрос, в котором использовалось бы полное внешнее соединение — `FULL OUTER JOIN`.

9. Для ответа на вопрос, сколько рейсов выполняется из Москвы в Санкт-Петербург, можно написать совсем простой запрос:

```
SELECT count( * )
  FROM routes
  WHERE departure_city = 'Москва'
    AND arrival_city   = 'Санкт-Петербург';
```

```
 count
-------
    12
(1 строка)
```

А с помощью какого запроса можно получить результат в таком виде?

```
 departure_city |  arrival_city   | count
----------------+-----------------+-------
 Москва         | Санкт-Петербург |    12
(1 строка)
```

10. Выяснить, сколько различных рейсов выполняется из каждого города, без учета частоты рейсов в неделю, можно с помощью обращения к представлению «Маршруты» (`routes`):

```
SELECT departure_city, count( * )
  FROM routes
  GROUP BY departure_city
  ORDER BY count DESC;
```

```
     departure_city       | count
--------------------------+-------
 Москва                   |   154
 Санкт-Петербург          |    35
 Новосибирск              |    19
 ...
 Благовещенск             |     1
 Братск                   |     1
(101 строка)
```

Модифицируйте этот запрос так, чтобы он выводил число направлений, по которым летают самолеты из каждого города. Например, из Москвы в Санкт-Петербург летает несколько различных рейсов, но все эти рейсы относятся к одному направлению.

Указание: нужно передать параметр в функцию `count`.

11. В материализованном представлении «Маршруты» (`routes`) имеется столбец `days_of_week`, который содержит списки (массивы) номеров дней недели, когда выполняется каждый рейс.

Для оптимизации расписания вылетов из Москвы нужно выявить пять городов, в которые из столицы отправляется наибольшее число ежедневных рейсов (маршрутов). Строки в выборке следует расположить в убывающем порядке числа выполняемых рейсов.

Указание: воспользуйтесь функцией `array_length`.

12.* Предположим, что служба материального снабжения нашей авиакомпании запросила информацию о числе рейсов, выполняющихся из Москвы в каждый день недели.

Результат можно получить путем выполнения семи аналогичных запросов: по одному для каждого дня недели. Начнем с понедельника:

```
SELECT 'Понедельник' AS day_of_week, count( * ) AS num_flights
  FROM routes
  WHERE departure_city = 'Москва'
    AND days_of_week @> '{ 1 }'::integer[];
```

В этом запросе используется оператор @>, который проверяет, содержатся ли все элементы массива, стоящего справа от него, в том массиве, который находится слева. В правом массиве всего один элемент — номер интересующего нас дня недели.

```
 day_of_week | num_flights
-------------+-------------
 Понедельник |         131
(1 строка)
```

Запрос для вторника отличается лишь номером дня недели в массиве.

```
SELECT 'Вторник' AS day_of_week, count( * ) AS num_flights
  FROM routes
  WHERE departure_city = 'Москва'
    AND days_of_week @> '{ 2 }'::integer[];
```

```
 day_of_week | num_flights
-------------+-------------
 Вторник     |         134
(1 строка)
```

Нужно выполнить еще пять аналогичных команд, чтобы получить результаты для всех дней недели. Очевидно, что это нерациональный способ.

Получить требуемый результат можно с помощью одного запроса:

```
SELECT unnest( days_of_week ) AS day_of_week,
       count( * ) AS num_flights
  FROM routes
  WHERE departure_city = 'Москва'
  GROUP BY day_of_week
  ORDER BY day_of_week;
```

```
 day_of_week | num_flights
-------------+-------------
           1 |         131
           2 |         134
           3 |         126
           4 |         136
           5 |         124
           6 |         133
           7 |         124
(7 строк)
```

Задание 1. Самостоятельно разберитесь, как работает приведенный запрос. Выясните, что делает функция unnest. Для того чтобы найти ее описание, можно воспользоваться теми разделами документации, которые были указаны в главе 4. Однако можно воспользоваться и предметным указателем (Index), ссылка на который находится в самом низу оглавления документации.

В качестве вспомогательного запроса, проясняющего работу функции unnest, можно выполнить следующий:

```
SELECT flight_no, unnest( days_of_week ) AS day_of_week
  FROM routes
  WHERE departure_city = 'Москва'
  ORDER BY flight_no;
```

Задание 2. Использование номеров дней недели в предыдущей выборке не должно вызывать затруднений. Но все-таки предположим, что нас попросили модифицировать запрос, чтобы результат выводился в таком виде:

```
 name_of_day | num_flights
-------------+-------------
 Пн.         |         131
 Вт.         |         134
 Ср.         |         126
 Чт.         |         136
 Пт.         |         124
 Сб.         |         133
 Вс.         |         124
(7 строк)
```

Покажем одно из возможных решений задачи. Оно основано на использовании специальной табличной функции unnest в предложении FROM. Подробно об этом написано в документации в разделе 7.2.1.4 «Табличные функции». Функция может принимать любое число параметров-массивов, а возвращает набор

строк, которые могут использоваться в запросах как обычные таблицы. В этих наборах строк столбцы формируются из значений, содержащихся в массивах.

```
SELECT dw.name_of_day, count( * ) AS num_flights
  FROM (
    SELECT unnest( days_of_week ) AS num_of_day
      FROM routes
      WHERE departure_city = 'Москва'
    ) AS r,
    unnest( '{ 1, 2, 3, 4, 5, 6, 7 }'::integer[],
      '{ "Пн.", "Вт.", "Ср.", "Чт.", "Пт.", "Сб.", "Вс."}'::text[]
    ) AS dw( num_of_day, name_of_day )
  WHERE r.num_of_day = dw.num_of_day
  GROUP BY r.num_of_day, dw.name_of_day
  ORDER BY r.num_of_day;
```

Этот запрос можно упростить. Предложение `WITH ORDINALITY` позволяет в нашем примере избавиться от массива целых чисел, обозначающих дни недели, поскольку автоматически формируется столбец целых чисел, нумерующих строки результирующего набора. По умолчанию этот столбец называется `ordinality`. Это имя можно использовать в запросе. Самостоятельно модифицируйте запрос с применением предложения `WITH ORDINALITY`.

13. Ответить на вопрос о том, каковы максимальные и минимальные цены билетов на все направления, может такой запрос:

```
SELECT f.departure_city, f.arrival_city,
    max( tf.amount ), min( tf.amount )
  FROM flights_v f
  JOIN ticket_flights tf ON f.flight_id = tf.flight_id
  GROUP BY 1, 2
  ORDER BY 1, 2;
```

```
   departure_city    |     arrival_city     |    max    |   min
---------------------+----------------------+-----------+----------
 Абакан              | Москва               | 101000.00 | 33700.00
 Абакан              | Новосибирск          |   5800.00 |  5800.00
 Абакан              | Томск                |   4900.00 |  4900.00
 Анадырь             | Москва               | 185300.00 | 61800.00
 Анадырь             | Хабаровск            |  92200.00 | 30700.00
 ...
 Якутск              | Мирный               |   8900.00 |  8100.00
 Якутск              | Санкт-Петербург      | 145300.00 | 48400.00
(367 строк)
```

А как выявить те направления, на которые не было продано ни одного билета? Один из вариантов решения такой: если на рейсы, отправляющиеся по какому-то направлению, не было продано ни одного билета, то максимальная и минимальная цены будут равны NULL. Нужно получить выборку в таком виде:

```
   departure_city    |    arrival_city     |    max    |    min
---------------------+---------------------+-----------+----------
 Абакан              | Архангельск         |           |
 Абакан              | Грозный             |           |
 Абакан              | Кызыл               |           |
 Абакан              | Москва              | 101000.00 | 33700.00
 Абакан              | Новосибирск         |   5800.00 |  5800.00
 ...
```

Модифицируйте запрос, приведенный выше.

14. Предположим, что маркетологи нашей авиакомпании хотят знать, как часто встречаются различные имена среди пассажиров? Получить распределение частот имен пассажиров в таблице «Билеты» (`tickets`) поможет такой запрос:

```
SELECT left( passenger_name, strpos( passenger_name, ' ' ) - 1 )
       AS firstname, count( * )
  FROM tickets
  GROUP BY 1
  ORDER BY 2 DESC;
```

```
 firstname | count
-----------+-------
 ALEKSANDR | 20328
 SERGEY    | 15133
 VLADIMIR  | 12806
 TATYANA   | 12058
 ELENA     | 11291
 OLGA      |  9998
 ...
 MAGOMED   |    14
 ASKAR     |    13
 RASUL     |    11
(363 строки)
```

Напишите запрос для ответа на аналогичный вопрос насчет распределения частот фамилий пассажиров.

Подробные сведения о других функциях для работы со строковыми данными приведены в документации в разделе 9.4 «Строковые функции и операторы».

15.* В тексте главы были кратко рассмотрены оконные функции. Самостоятельно прочитайте разделы документации, которые рекомендуется изучить для более детального ознакомления с этим классом функций.

Подумайте, в какой ситуации, связанной с базой данных «Авиаперевозки», было бы полезно применить оконные функции, и напишите запрос.

16.* Вместе с агрегатными функциями может использоваться предложение `FILTER`. Самостоятельно ознакомьтесь с этой темой, обратившись к разделу документации 4.2.7 «Агрегатные выражения». Напишите запрос с использованием предложения `FILTER` с агрегатной функцией.

17. В тексте главы в разделе 6.4 мы рассмотрели два способа получения ответа на вопрос: как распределяются места с разными классами обслуживания в самолетах всех типов?

А с помощью какого запроса можно получить результат в таком виде?

```
 aircraft_code |       model         | fare_conditions | count
---------------+---------------------+-----------------+-------
 319           | Airbus A319-100     | Business        |    20
 319           | Airbus A319-100     | Economy         |    96
...
 CR2           | Bombardier CRJ-200  | Economy         |    50
 SU9           | Sukhoi SuperJet-100 | Business        |    12
 SU9           | Sukhoi SuperJet-100 | Economy         |    85
(17 строк)
```

18. В разделе 6.2 мы находили ответ на вопрос: сколько маршрутов обслуживают самолеты каждого типа? Но для повышения наглядности получаемых результатов необходимо еще рассчитывать относительные величины, т. е. доли от общего числа маршрутов.

Вот что требуется получить:

```
 a_code |         model       | r_code | num_routes | fraction
--------+---------------------+--------+------------+----------
 CR2    | Bombardier CRJ-200  | CR2    |        232 |    0.327
 CN1    | Cessna 208 Caravan  | CN1    |        170 |    0.239
 ...
 773    | Boeing 777-300      | 773    |         10 |    0.014
 320    | Airbus A320-200     |        |          0 |    0.000
(9 строк)
```

19.* В разделе 6.4 мы использовали рекурсивный алгоритм в общем табличном выражении. Изучите этот пример, чтобы лучше понять работу рекурсивного алгоритма:

```
WITH RECURSIVE ranges ( min_sum, max_sum )
AS (
     VALUES( 0,      100000 ),
           ( 100000, 200000 ),
           ( 200000, 300000 )
     UNION ALL
     SELECT min_sum + 100000, max_sum + 100000
       FROM ranges
       WHERE max_sum < ( SELECT max( total_amount ) FROM bookings )
   )
SELECT * FROM ranges;
```

```
 min_sum | max_sum
---------+---------
       0 |  100000       исходные строки
  100000 |  200000
  200000 |  300000

  100000 |  200000       результат первой итерации
  200000 |  300000
  300000 |  400000

  200000 |  300000       результат второй итерации
  300000 |  400000
  400000 |  500000

  300000 |  400000
  400000 |  500000
  500000 |  600000
  ...
 1000000 | 1100000       результат (n-3)-й итерации
 1100000 | 1200000
 1200000 | 1300000

 1100000 | 1200000       результат (n-2)-й итерации
 1200000 | 1300000

 1200000 | 1300000       результат (n-1)-й итерации (предпоследней)
(36 строк)
```

Здесь мы с помощью предложения VALUES специально создали виртуальную таблицу из трех строк, хотя для получения требуемого результата достаточно только одной строки (0, 100000). Еще важно то, что предложение UNION ALL не удаляет строки-дубликаты, поэтому мы можем видеть весь рекурсивный процесс порождения новых строк.

При рекурсивном выполнении запроса

```
SELECT min_sum + 100000, max_sum + 100000
  FROM ranges
  WHERE max_sum < ( SELECT max( total_amount ) FROM bookings )
```

каждый раз выполняется проверка в условии WHERE. И на $(n-2)$-й итерации это условие отсеивает одну строку, т. к. после $(n-3)$-й итерации значение атрибута max_sum в третьей строке было равно 1 300 000.

Ведь запрос

```
SELECT max( total_amount ) FROM bookings;
```

выдаст значение

```
    max
------------
 1204500.00
(1 строка)
```

Таким образом, после $(n-2)$-й итерации во временной области остается всего две строки, после $(n-1)$-й итерации во временной области остается только одна строка.

Заключительная итерация уже не добавляет строк в результирующую таблицу, поскольку единственная строка, поданная на вход команде SELECT, будет отклонена условием WHERE. Работа алгоритма завершается.

Задание 1. Модифицируйте запрос, добавив в него столбец level (можно назвать его и iteration). Этот столбец должен содержать номер текущей итерации, поэтому нужно увеличивать его значение на единицу на каждом шаге. Не забудьте задать начальное значение для добавленного столбца в предложении VALUES.

Задание 2. Для завершения экспериментов замените UNION ALL на UNION и выполните запрос. Сравните этот результат с предыдущим, когда мы использовали UNION ALL.

20.* В тексте главы есть такой запрос, вычисляющий распределение сумм бронирований по диапазонам в 100 тысяч рублей:

```
WITH RECURSIVE ranges ( min_sum, max_sum )
AS (
     VALUES( 0, 100000 )
     UNION ALL
     SELECT min_sum + 100000, max_sum + 100000
       FROM ranges
       WHERE max_sum < ( SELECT max( total_amount ) FROM bookings )
   )
SELECT r.min_sum,
       r.max_sum,
       count( b.* )
  FROM bookings b
  RIGHT OUTER JOIN ranges r
    ON  b.total_amount >= r.min_sum
    AND b.total_amount <  r.max_sum
  GROUP BY r.min_sum, r.max_sum
  ORDER BY r.min_sum;
```

Как вы думаете, почему функция `count` получает в качестве параметра выражение `b.*`, а не просто `*`? Что изменится, если оставить только `*`, и почему?

21. В тексте главы был приведен запрос, выводящий список городов, в которые нет рейсов из Москвы.

```
SELECT DISTINCT a.city
  FROM airports a
  WHERE NOT EXISTS (
    SELECT * FROM routes r
      WHERE r.departure_city = 'Москва'
        AND r.arrival_city = a.city
    )
    AND a.city <> 'Москва'
  ORDER BY city;
```

Можно предложить другой вариант, в котором используется одна из операций над множествами строк: объединение, пересечение или разность.

Вместо знака «?» поставьте в приведенном ниже запросе нужное ключевое слово — `UNION`, `INTERSECT` или `EXCEPT` — и обоснуйте ваше решение.

```
SELECT city
  FROM airports
  WHERE city <> 'Москва'
?
SELECT arrival_city
  FROM routes
  WHERE departure_city = 'Москва'
ORDER BY city;
```

22. В тексте главы мы рассматривали такой запрос: получить перечень аэропортов в тех городах, в которых больше одного аэропорта.

```
SELECT aa.city, aa.airport_code, aa.airport_name
  FROM (
    SELECT city, count( * )
      FROM airports
      GROUP BY city
      HAVING count( * ) > 1
    ) AS a
  JOIN airports AS aa ON a.city = aa.city
  ORDER BY aa.city, aa.airport_name;
```

Как вы думаете, обязательно ли наличие функции `count` в подзапросе в предложении `SELECT` или можно написать просто

```
SELECT city FROM airports
```

Сначала попробуйте дать ответ теоретически, а потом проверьте вашу гипотезу на компьютере.

23. Предположим, что департамент развития нашей авиакомпании задался вопросом: каким будет общее число различных маршрутов, которые теоретически можно проложить между всеми городами?

Если в каком-то городе имеется более одного аэропорта, то это учитывать не будем, т. е. маршрутом будем считать путь между *городами*, а не между *аэропортами*. Здесь мы используем соединение таблицы с самой собой на основе неравенства значений атрибутов.

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

```
 count
-------
 10100
(1 строка)
```

Задание. Перепишите этот запрос с общим табличным выражением.

24. В тексте главы мы рассмотрели использование подзапросов в предикатах EXISTS и IN. Существуют также предикаты многократного сравнения ANY и ALL. Они представлены в документации в разделе 9.22 «Выражения подзапросов». Самостоятельно ознакомьтесь с этими предикатами и напишите несколько запросов с их применением.

    Предикаты ANY и ALL имеют некоторую связь с предикатом IN. В частности, использование IN эквивалентно использованию конструкции = ANY, а использование NOT IN эквивалентно использованию конструкции <> ALL.

    Пример двух эквивалентных запросов, выбирающих аэропорты в часовых поясах Asia/Novokuznetsk и Asia/Krasnoyarsk:

```
SELECT * FROM airports
  WHERE timezone IN ( 'Asia/Novokuznetsk', 'Asia/Krasnoyarsk' );

SELECT * FROM airports
  WHERE timezone = ANY (
    VALUES ( 'Asia/Novokuznetsk' ), ( 'Asia/Krasnoyarsk' )
  );
```

    Еще один пример. В тексте главы мы рассматривали запрос, подсчитывающий количество маршрутов, проложенных из самых восточных аэропортов.

```
SELECT departure_city, count( * )
  FROM routes
  GROUP BY departure_city
  HAVING departure_city IN (
    SELECT city
      FROM airports
      WHERE longitude > 150
  )
  ORDER BY count DESC;
```

    В этом запросе можно заменить IN на ANY таким образом:

```
HAVING departure_city = ANY ( ... )
```

25.* При планировании новых маршрутов и оценке экономической эффективности уже существующих может потребоваться информация о том, какова усредненная степень заполнения самолетов на всех направлениях.

Будем учитывать только уже прибывшие рейсы.

```
WITH tickets_seats
AS (
     SELECT f.flight_id,
            f.flight_no,
            f.departure_city,
            f.arrival_city,
            f.aircraft_code,
            count( tf.ticket_no ) AS fact_passengers,
            ( SELECT count( s.seat_no )
                FROM seats s
                WHERE s.aircraft_code = f.aircraft_code
            ) AS total_seats
       FROM flights_v f
       JOIN ticket_flights tf ON f.flight_id = tf.flight_id
       WHERE f.status = 'Arrived'
       GROUP BY 1, 2, 3, 4, 5
   )
SELECT ts.departure_city,
       ts.arrival_city,
       sum( ts.fact_passengers ) AS sum_pass,
       sum( ts.total_seats ) AS sum_seats,
       round( sum( ts.fact_passengers )::numeric /
              sum( ts.total_seats )::numeric, 2 ) AS frac
  FROM tickets_seats ts
  GROUP BY ts.departure_city, ts.arrival_city
  ORDER BY ts.departure_city;
```

```
 departure_city |  arrival_city   | sum_pass | sum_seats | frac
----------------+-----------------+----------+-----------+------
 Абакан         | Tomsk           |      258 |       360 | 0.72
 Абакан         | Novosibirsk     |      217 |       348 | 0.62
 Абакан         | Moscow          |      466 |      1044 | 0.45
 ...
 Якутск         | Санкт-Петербург |      352 |      3596 | 0.10
(361 строка)
```

Для того чтобы лучше уяснить, как работает запрос в целом, вычлените из него отдельные подзапросы и выполните их, посмотрите, что они выводят.

Как вы считаете, равносильно ли в данном запросе

```
SELECT count( s.seat_no )
```

и

```
SELECT count( s.* )
```

Почему?

Задание. Модифицируйте этот запрос, чтобы он выводил те же отчетные данные, но с учетом классов обслуживания, т. е. `Business`, `Comfort` и `Economy`.

26.* Предположим, что некая контролирующая организация потребовала информацию о размещении пассажиров одного из рейсов Кемерово — Москва в салоне самолета. Для определенности выберем конкретный рейс из тех рейсов, которые уже прибыли на момент времени, соответствующий текущему моменту. Текущий момент времени в базе данных «Авиаперевозки» определяется с помощью функции `bookings.now`.

Выполним запрос:

```
SELECT *
  FROM flights_v
  WHERE departure_city = 'Кемерово'
    AND arrival_city = 'Москва'
    AND actual_arrival < bookings.now();
```

Выберем для дальнейшей работы рейс, у которого значения атрибутов `flight_id` — 27584, `aircraft_code` — SU9.

Получим список пассажиров этого рейса с местами, которые им были назначены в салоне самолета.

```
SELECT t.passenger_name, b.seat_no
  FROM (
    ticket_flights tf
    JOIN tickets t ON tf.ticket_no = t.ticket_no
  )
  JOIN boarding_passes b
    ON  tf.ticket_no = b.ticket_no
    AND tf.flight_id = b.flight_id
  WHERE tf.flight_id = 27584
  ORDER BY t.passenger_name;
```

```
 passenger_name      | seat_no
---------------------+---------
 ALEKSANDR ABRAMOV   | 1A
 ALEKSANDR GRIGOREV  | 5C
 ALEKSANDR SERGEEV   | 6F
 ALEKSEY FEDOROV     | 11D
 ALEKSEY MELNIKOV    | 18A
...
 VLADIMIR POPOV      | 11A
 YAROSLAV KUZMIN     | 18F
 YURIY ZAKHAROV      | 10F
(44 строки)
```

Отсортируем строки по фамилиям пассажиров:

```
SELECT t.passenger_name,
    substr(   t.passenger_name,
      strpos( t.passenger_name, ' ' ) + 1
    ) AS lastname,
    left(     t.passenger_name,
      strpos( t.passenger_name, ' ' ) - 1
    ) AS firstname,
    b.seat_no
  FROM (
    ticket_flights tf
    JOIN tickets t ON tf.ticket_no = t.ticket_no
  )
  JOIN boarding_passes b
    ON  tf.ticket_no = b.ticket_no
    AND tf.flight_id = b.flight_id
  WHERE tf.flight_id = 27584
  ORDER BY 2, 3;
```

```
    passenger_name   | lastname  | firstname | seat_no
---------------------+-----------+-----------+---------
 ALEKSANDR ABRAMOV   | ABRAMOV   | ALEKSANDR | 1A
 NIKITA ANDREEV      | ANDREEV   | NIKITA    | 6D
 ANTONINA ANISIMOVA  | ANISIMOVA | ANTONINA  | 11F
 ...
 YURIY ZAKHAROV      | ZAKHAROV  | YURIY     | 10F
 ELENA ZOTOVA        | ZOTOVA    | ELENA     | 20E
(44 строки)
```

Получим список мест в салоне самолета и пассажиров, которые сидели на этих местах. При этом незанятые места также должны быть выведены (поэтому используем левое внешнее соединение `LEFT OUTER JOIN`).

```
SELECT s.seat_no, p.passenger_name
  FROM seats s
  LEFT OUTER JOIN (
    SELECT t.passenger_name, b.seat_no
      FROM (
        ticket_flights tf
        JOIN tickets t ON tf.ticket_no = t.ticket_no
      )
      JOIN boarding_passes b
        ON  tf.ticket_no = b.ticket_no
        AND tf.flight_id = b.flight_id
      WHERE tf.flight_id = 27584
  ) AS p
     ON s.seat_no = p.seat_no
  WHERE s.aircraft_code = 'SU9'
  ORDER BY s.seat_no;
```

```
 seat_no | passenger_name
---------+---------------------
 10A     |
 10C     |
 10D     | NATALYA POPOVA
 10E     |
 10F     | YURIY ZAKHAROV
 11A     | VLADIMIR POPOV
 11C     | ANNA KUZMINA
 ...
 8F      |
 9A      | MAKSIM CHERNOV
 9C      |
 9D      | LYUDMILA IVANOVA
 9E      |
 9F      | SOFIYA KULIKOVA
(97 строк)
```

Предположим, что нас попросили отсортировать места в порядке их расположения в салоне самолета и вывести также адреса электронной почты пассажиров (у кого они были указаны при бронировании). Для выполнения второго требования воспользуемся столбцом `contact_data`. В нем содержатся JSON-объекты,

содержащие контактные данные пассажиров. Ряд из них имеет ключ `email`. Модифицированный запрос будет таким:

```
SELECT s.seat_no, p.passenger_name, p.email
  FROM seats s
  LEFT OUTER JOIN (
    SELECT t.passenger_name, b.seat_no,
           t.contact_data->'email' AS email
      FROM (
        ticket_flights tf
        JOIN tickets t ON tf.ticket_no = t.ticket_no
      )
      JOIN boarding_passes b
        ON  tf.ticket_no = b.ticket_no
        AND tf.flight_id = b.flight_id
      WHERE tf.flight_id = 27584
  ) AS p
     ON s.seat_no = p.seat_no
  WHERE s.aircraft_code = 'SU9'
  ORDER BY
    left( s.seat_no, length( s.seat_no ) - 1 )::integer,
    right( s.seat_no, 1 );
```

```
 seat_no |   passenger_name  | email
---------+-------------------+-----------------------------------
 1A      | ALEKSANDR ABRAMOV |
 1C      |                   |
 1D      | DENIS PETROV      |
 1F      | LEONID BARANOV    | "baranov.l.1967@postgrespro.ru"
 2A      |                   |
 2C      |                   |
 ...
 9F      | SOFIYA KULIKOVA   | "sofiya.kulikova_041963@postgre..."
 10A     |                   |
 10C     |                   |
 10D     | NATALYA POPOVA    | "popova.n_13031976@postgrespro.ru"
 ...
 20E     | ELENA ZOTOVA      |
 20F     | LILIYA OSIPOVA    |
(97 строк)
```

Задание. Перепишите последний запрос с использованием общего табличного выражения и добавьте столбец «Класс обслуживания» (`fare_conditions`).