7

SQL и реляционная алгебра II: дополнительные операции

Как я уже неоднократно говорил, оператором реляционной алгебры называется такой оператор, который принимает на входе одно или несколько отношений и порождает на выходе новое отношение. Однако в главе 1 отмечалось, что можно определить произвольное количество операторов, отвечающих такой простой характеристике. В главе 6 были описаны оригинальные операторы (соединение, проекция и т. д.), а в этой главе мы рассмотрим некоторые из многочисленных дополнительных операторов, которые были определены уже после изобретения реляционной модели. Мы также обсудим, как эти операторы лучше всего реализовать в SQL.

Полусоединение и полуразность

Соединение – один из наиболее известных реляционных операторов. Но на практике часто бывает, что в тех запросах, где вообще необходимо соединение, возникает потребность в обобщенной форме этого оператора, которая называется полусоединением (возможно, вы никогда не слышали о полусоединении, но это достаточно важный оператор).

> **Определение**: *Полусоединение* отношений *r1* и *r2* (в таком порядке), *r1* MATCHING *r2*, эквивалентно операции (*r1* JOIN *r2*){*A*,*B*,...,*C*}, где *A*, *B*, ..., *C* – все атрибуты *r1*.

Иначе говоря, *r1* MATCHING *r2* – это результат соединения *r1* и *r2*, спроецированный на атрибуты *r1*. Рассмотрим пример («Получить всех поставщиков, которые в настоящий момент поставляют хотя бы одну деталь»):

```
S MATCHING SP                |   SELECT S.* FROM S
                             |   WHERE  SNO IN
                             |        ( SELECT SNO FROM SP )
```

Заголовок результата такой же, как у S. Отметим, что выражения *r1* MATCHING *r2* и *r2* MATCHING *r1* в общем случае не эквивалентны. Отметим также, что можно было бы заменить ключевое слово IN в SQL словом MATCH; однако любопытно, что заменить NOT IN на NOT MATCH в операторе полуразности (см. ниже) не получится, потому что в SQL нет оператора «NOT MATCH».

Теперь обратимся к полуразности. Если полусоединение в каком-то отношении важнее соединения, то к полуразности это замечание относится даже в большей мере – на практике в большинстве запросов, где используется разность, на самом деле необходима полуразность.

> **Определение**: *Полуразность* между отношениями *r1* и *r2* (в таком порядке), *r1* NOT MATCHING *r2*, эквивалентна операции *r1* MINUS (*r1* MATCHING *r2*).

Вот пример («Получить всех поставщиков, который в настоящий момент не поставляют ни одной детали»):

```
S NOT MATCHING SP                |    SELECT S.* FROM S
                                 |    WHERE  SNO NOT IN
                                 |         ( SELECT SNO FROM SP )
```

Опять-таки заголовок результата такой же, как у S. *Примечание*: Если отношения *r1* и *r2* одного типа, то *r1* NOT MATCHING *r2* вырождается в *r1* MINUS *r2*; другими словами, в реляционной теории разность (MINUS) – это частный случай полуразности. Однако соединение не является частным случаем полусоединения, это совершенно разные операторы, хотя неформально можно сказать, что некоторые соединения являются полусоединениями, а некоторые полусоединения – соединениями. См. упражнение 7.19 в конце главы.

Расширение

Возможно, вы обратили внимание, что в описанной до сих пор алгебре нет традиционных средств вычислений. А в SQL они есть; например, никто не мешает написать запрос такого вида: SELECT A + B AS C ... . Однако, написав знак «+», мы сразу же вышли за рамки первоначально определенной алгебры. Поэтому, чтобы получить такого рода функциональность, нам необходимо что-то добавить в алгебру, и это «что-то» как раз и есть оператор EXTEND. Предположим, например, что веса деталей (в отношении P) выражены в фунтах, и что мы хотим получить их в граммах. Поскольку в фунте 454 грамма, то можно написать такой запрос:

```
EXTEND P                         |    SELECT P.* ,
   ADD ( WEIGHT * 454 AS GMWT )  |           WEIGHT * 454 AS GMWT
                                 |    FROM   P
```

Для наших тестовых данных получится такой результат:

PNO	PNAME	COLOR	WEIGHT	CITY	GMWT
P1	Nut	Red	12.0	London	5448.0
P2	Bolt	Green	17.0	Paris	7718.0
P3	Screw	Blue	17.0	Oslo	7718.0
P4	Screw	Red	14.0	London	6356.0
P5	Cam	Blue	12.0	Paris	5448.0
P6	Cog	Red	19.0	London	8626.0

Важно: Переменная-отношение P в базе данных не изменяется! EXTEND – не аналог SQL-предложения ALTER TABLE; выражение EXTEND – это всего лишь выражение, и, как всякое выражение, служит для обозначения некоторого значения.

Продолжая этот пример, рассмотрим запрос «Получить номер детали и вес в граммах для тех деталей, для которых вес в граммах больше 7000 граммов»:

```
( ( EXTEND P ADD                |    SELECT PNO ,
    ( WEIGHT * 454 AS GMWT ) ) |           WEIGHT * 454 AS GMWT
        WHERE GMWT > 7000.0 )  |    FROM   P
                { PNO , GMWT } |    WHERE  WEIGHT * 454 > 7000.0
```

Как видите, в SQL-версии выражение WEIGHT * 454 встречается дважды, и мы можем лишь надеяться, что реализация достаточно изощренна, чтобы понять, что это выражение нужно вычислять не два, а один раз для каждого кортежа (или строки). В версии же для языка **Tutorial D** выражение встречается только один раз.

Проблема, которую иллюстрирует этот пример, заключается в том, что конструкция SELECT - FROM - WHERE в SQL слишком жесткая. В действительности нам здесь необходимо – из формулировки на **Tutorial D** это совершенно очевидно – выполнить ограничение расширения; в терминах SQL нужно применить оператор WHERE к результату SELECT. Но шаблон SELECT - FROM - WHERE вынуждает применять фразу WHERE к результату фразы FROM, а не фразы SELECT. Выразим ту же мысль по-другому: во многих отношениях весь смысл алгебры (благодаря замкнутости) состоит в том, чтобы реляционные операции можно было произвольным образом сочетать и вкладывать друг в друга. Но шаблон SELECT - FROM - WHERE в SQL означает по сути дела, что выразить можно только те запросы, в которых сначала выполняется произведение, потом ограничение, а потом некоторая комбинация проекции и/или расширения, и/или разгруппирования, и/или переименования. Однако существует много запросов, не укладывающихся в такую схему.

Кстати, возможно, вам интересно, почему я не записал SQL-версию в таком виде:

```
SELECT PNO , WEIGHT * 454 AS GMWT
FROM   P
WHERE  GMWT > 7000.0
```

(изменена последняя строчка). Причина в том, что GMWT – имя столбца в *конечном результате*; в таблице P такого столбца нет, поэтому фраза WHERE не имеет смысла и вызовет ошибку на этапе компиляции.

На самом деле стандарт SQL допускает формулировку рассматриваемого запроса в виде, чуть более близком к **Tutorial D** (ниже для ясности я привожу полные имена):

```
SELECT TEMP.PNO , TEMP.GMWT
FROM ( SELECT P.PNO , ( P.WEIGHT * 454 ) AS GMWT
       FROM P ) AS TEMP
WHERE  TEMP.GMWT > 7000.0
```

Но не во всех SQL-системах разрешается употреблять вложенные подзапросы во фразе FROM. Отмечу также, что такая формулировка неизбежно ведет к необходимости ссылаться на некоторые переменные (в данном случае TEMP) еще до того, как они определены, а в реальных SQL-запросах, быть может, задолго до точки определения.

Я завершаю этот раздел формальным определением:

> **Определение:** Пусть *r* – отношение. *Расширением* EXTEND *r* ADD (*exp* AS X) называется отношение, для которого (а) заголовок совпадает с заголовком *r*, дополненным атрибутом *X*, и (б) тело состоит из множества всех кортежей *t*, таких что *t* есть кортеж *r*, дополненный значением атрибута *X*, которое получается вычислением выражения *exp* для этого кортежа *r*. В отношении *r* не должно быть атрибута с именем *X* и *exp* не должно ссылаться на *X*. Отметим, что кардинальность результата равна кардинальности *r*, а степень равна степени *r* плюс 1. Типом *X* в результате является тип *exp*.

Отношения-образы

Отношение-образ – это, неформально говоря, «образ» некоторого кортежа внутри некоторого отношения (обычно кортеж принадлежит какому-то другому отношению). Например, для базы данных о поставщиках и деталях следующее отношение является образом кортежа поставщика S4 (из отношения «поставщики») в отношении «поставки»:

PNO	QTY
P2	200
P4	300
P5	400

Ясно, что это отношение-образ можно получить с помощью такого выражения на языке **Tutorial D**:

```
( SP WHERE SNO = 'S4' ) { ALL BUT SNO }
```

Приведем формальное определение отношения-образа в общем случае.

> **Определение**: Пусть *r1* и *r2* – соединяемые отношения (то есть одноименные атрибуты имеют один и тот же тип); пусть *t1* – кортеж *r1*, а *t2* – кортеж *r2*, для которого значения общих с кортежем *t1* атрибутов такие же, как в кортеже *t1*. Пусть отношение *r3* – это ограничение *r2*, которое содержит все такие и только такие кортежи *t2*, и пусть *r4* – проекция *r3* на все атрибуты, кроме общих. Тогда *r4* называется *отношением-образом* (относительно *r2*), соответствующим *t1*.

Следующий пример иллюстрирует полезность отношений-образов:

```
S WHERE ( !!SP ) { PNO } = P { PNO }
```

Отметим попутно, что здесь булево выражение во фразе WHERE является реляционным сравнением.

Объяснение:

- Во-первых, в качестве *r1* и *r2* из определения выше выступают отношения «поставщики» и «поставки» соответственно (под «отношением “поставщики”» я понимаю текущее значение переменной-отношения S, и аналогично для «отношения “поставки”»).
- Далее, мы можем представить себе, что булево выражение во фразе WHERE вычисляется для каждого кортежа *t1* из *r1* (то есть для каждого кортежа в отношении «поставщики») по очереди.
- Рассмотрим один такой кортеж, скажем, для поставщика S*x*. Тогда для этого кортежа выражение !!SP обозначает соответствующее отношение-образ *r4* внутри *r2*; иными словами, это множество пар (PNO,QTY) отношения SP для деталей, поставляемых поставщиком S*x*. Выражение !!SP представляет собой *ссылку на отношение-образ*.
- Выражение (!!SP){PNO}, то есть проекция отношения-образа на {PNO}, обозначает, следовательно, множество номеров деталей для деталей, поставляемых поставщиком S*x*.
- Таким образом, все выражение в целом (то есть S WHERE ...) обозначает поставщиков из S, для которых это множество номеров деталей равно множеству всех номеров деталей в проекции P на {PNO}. Иначе говоря, оно представляет запрос «Получить поставщиков, которые поставляют все детали» (допуская некоторую вольность формулировки).

Примечание

Поскольку понятие отношения-образа определяется в терминах некоторого заданного кортежа (в формальном определении он назван *t1*), то понятно, что ссылка на отношение-образ может встречаться не во всех контекстах, где допустимы общие реляционные выражения, а только в некоторых, точнее в тех, где кортеж *t1* имеет смысл. Примером такого контекста могут служить фразы WHERE, как показывает приведенный выше пример. Другой пример мы увидим в разделе «Еще об отношениях-образах».

Отступление

В SQL нет прямой поддержки отношений-образов как таковых. Но для интересующихся покажу аналог приведенного выше выражения **Tutorial D** (я включил его только для справки и не собираюсь вдаваться в детали, замечу лишь, что, очевидно (?), есть много способов улучшить его):

```
SELECT *
FROM   S
WHERE  NOT EXISTS
     ( SELECT PNO
       FROM   SP
       WHERE  SP.SNO = S.SNO
       EXCEPT
       SELECT PNO
       FROM   P )
AND    NOT EXISTS
     ( SELECT PNO
       FROM   P
       EXCEPT
       SELECT PNO
       FROM   P
       WHERE  SP.SNO = S.SNO )
```

Но вернемся к отношениям-образам. Стоит отметить, что оператор «!!» можно определить в терминах MATCHING. Например, для рассмотренного выше примера выражение

```
S WHERE ( !!SP ) { PNO } = P { PNO }
```

логически эквивалентно такому выражению:

```
S WHERE ( SP MATCHING RELATION { TUPLE { SNO SNO } } ) { PNO } = P { PNO }
```

Объяснение: Снова рассмотрим некоторый кортеж S, скажем, для поставщика S*x*. Для этого кортежа выражение TUPLE{SNO SNO} – вызов селектора кортежа – обозначает кортеж, содержащий только значение атрибута SNO кортежа S*x* (первое вхождение SNO – имя атрибута, второе – значение атрибута с этим именем в кортеже S*x* переменной-отношения S). Таким образом, выражение

```
RELATION { TUPLE { SNO SNO } }
```

представляет собой вызов селектора отношения и обозначает отношение, содержащее только этот кортеж. Следовательно, выражение

```
SP MATCHING RELATION { TUPLE { SNO SNO } }
```

обозначает некоторое ограничение SP, а именно ограничение, которое содержит только те кортежи отношения «поставки», в которых значение SNO такое же, как в кортеже поставщика S*x* из отношения «поставщики». Что и требовалось доказать.

Предположим теперь, что база данных о поставщиках несколько изменена (одновременно обобщена и упрощена) и выглядит следующим образом (в общих чертах):

```
S   { SNO }       /* поставщики                       */
SP  { SNO, PNO }  /* поставщик поставляет деталь      */
PJ  { PNO, JNO }  /* деталь используется в проекте */
J   { JNO }       /* проекты                          */
```

Здесь переменная-отношение J представляет проекты (JNO – номер проекта), а переменная-отношение PJ описывает, какие детали в каких проектах используются. Рассмотрим запрос «Получить все пары (*sno*,*jno*), такие что *sno* – значение SNO, входящее в данный момент времени в переменную-отношение S, *jno* – значение JNO, входящее в данный момент времени в переменную-отношение J, и поставщик *sno* поставляет все детали, использующиеся в проекте *jno*». Это сложный запрос! Но с применением отношений-образов он записывается почти тривиально:

```
( S JOIN J ) WHERE !!PJ ⊆ !!SP
```

Вернемся к обычной базе данных о поставщиках и деталях и рассмотрим еще один запрос («Удалить поставки от поставщиков, находящихся в Лондоне», и на этот раз я приведу также аналог на SQL):

```
DELETE SP WHERE IS_NOT_EMPTY   |   DELETE FROM SP
  ( !!(S WHERE                 |   WHERE  SNO IN
        CITY = 'London') ) ;   |         ( SELECT SNO FROM S
                               |           WHERE CITY = 'London' ) ;
```

Для данной поставки заданное отношение-образ !!(S WHERE ...) либо пусто, либо содержит ровно один кортеж.

Деление

Я включил в эту главу обсуждение операции деления только для того, чтобы показать, почему (быть может, вразрез с общепринятым мнением) я не считаю ее очень важной; я даже думаю, что от нее следовало бы отказаться. Если хотите, можете пропустить этот раздел.

У меня есть несколько причин (по меньшей мере, три) желать исключения операции деления. Во-первых, любой запрос, который можно

выразить в терминах деления, можно сформулировать, причем гораздо проще, в терминах отношений-образов, что я скоро и продемонстрирую. Во-вторых, существует по меньшей мере семь различных операторов деления, то есть по меньшей мере семь различных операторов, которые претендуют на название «деление», и я, безусловно, не хочу рассказывать о каждом. Вместо этого я ограничусь базовым, самым простым определением.

> **Определение:** Пусть отношения *r1* и *r2* таковы, что заголовок {*Y*} отношения *r2* является некоторым подмножеством заголовка *r1*, и множество {*X*} состоит из остальных атрибутов *r1*. Тогда результатом *деления r1* на *r2*, *r1* DIVIDEBY *r2*,[1] называется следующее отношение:
>
> ```
> r1 { X } NOT MATCHING ( ( r1 { X } JOIN r2 ) NOT MATCHING r1 )
> ```

Например, выражение

```
SP { SNO , PNO } DIVIDEBY P { PNO }
```

(на наших обычных тестовых данных) дает:

SNO
S1

Таким образом, неформально это выражение можно охарактеризовать как представление запроса «Получить номера тех поставщиков, которые поставляют все детали» (чуть ниже я объясню, почему «неформально»). На практике, однако, нам обычно интересна вся информация о поставщиках, а не только номера, и, следовательно, деление должно сопровождаться соединением:

```
( SP { SNO , PNO } DIVIDEBY P { PNO } ) JOIN S
```

Но ведь мы уже знаем, как этот запрос сформулировать проще, воспользовавшись отношениями-образами:

```
S WHERE ( !!SP ) { PNO } = P { PNO }
```

Эта формулировка (а) более лаконична, (б) проще для понимания (по крайней мере, мне так кажется) и (в) *корректна*. Разумеется, последний пункт важнее всего, и ниже я остановлюсь на нем подробнее. Но сначала я хочу объяснить, почему эта операция называется делением. А дело в том, что если *r1* и *r2* – отношения, не имеющие общих имен атрибутов, и мы образуем произведение *r1* TIMES *r2*, а затем разделим его на *r2*, то получим снова *r1*.[2] Другими словами, произведение и деление – в некотором смысле взаимно обратные операции.

---

[1] Язык **Tutorial D** не поддерживает этот оператор напрямую, поэтому запись *r1* DIVIDEBY *r2* в этом языке синтаксически недопустима.

[2] При условии, что *r2* не пусто. А если пусто, что произойдет?

Как я уже сказал, выражение

```
SP { SNO , PNO } DIVIDEBY P { PNO }
```

можно неформально охарактеризовать как формулировку запроса «Получить номера тех поставщиков, которые поставляют все детали». На самом деле, именно этот пример часто используют для объяснения и обоснования необходимости оператора деления. Но, к сожалению, такая характеристика не вполне корректна. В действительности это выражение соответствует запросу «Получить номера поставщиков, которые поставляют хотя бы одну деталь и фактически поставляют все детали».[1] Иначе говоря, оператор деления не только сложен и недостаточно лаконичен, но еще и не решает ту задачу, для которой изначально задумывался.

Агрегатные операторы

В каком-то смысле этот раздел можно назвать отвлечением от темы, потому что обсуждаемые здесь операторы не реляционные, а скалярные – они возвращают скаляр в качестве результата.[2] Но я все же хочу сказать о них несколько слов, прежде чем вернусь в основной теме главы.

В реляционной модели агрегатным называется оператор, который порождает единственное значение из «агрегата» (то есть множества или мультимножества) значений некоторого атрибута какого-то отношения. Для COUNT(*), представляющего особый случай, таким «агрегатом» является все отношение. Приведу два примера:

```
X := COUNT ( S ) ;              |   SELECT COUNT ( * ) AS X
                                |   FROM   S

Y := COUNT ( S { STATUS } ) ;   |   SELECT COUNT ( DISTINCT STATUS )
                                |          AS Y
                                |   FROM   S
```

Сначала рассмотрим предложения языка **Tutorial D** в левой части. Для наших тестовых данных в первом предложении переменной X присва-

---

[1] Если вы не понимаете, в чем тут логическое различие, то давайте рассмотрим чуть измененный запрос «Получить поставщиков, которые поставляют все фиолетовые детали» (смысл, конечно, в том, что фиолетовых деталей в базе нет). Если фиолетовых деталей вообще не существует, то каждый поставщик поставляет их все! – даже поставщик S5, которые не поставляет никаких деталей и, следовательно, не представлен в переменной-отношении SP, а потому не может быть возвращен аналогичным выражением DIVIDEBY. Если что-то все же осталось непонятно, подождите до главы 11, где мы продолжим обсуждение этого примера.

[2] Можно определить и нескалярные агрегатные операторы, но они выходят за рамки этой книги.

ивается значение 5 (количество кортежей в текущем значении переменной-отношения S), а во втором – переменной Y присваивается значение 3 (количество кортежей в проекции текущего значения переменной-отношения S на множество {STATUS}, то есть количество различных значений атрибута STATUS в текущем значении переменной-отношения).

В общем случае вызов агрегатного оператора в **Tutorial D** выглядит так:

```
<agg op name> ( <relation exp> [, <exp> ] )
```

В качестве операции *<agg op name>* допустимы COUNT, SUM, AVG, MAX, MIN, AND, OR и XOR (последние три применяются к агрегатам, состоящим из булевых значений). В выражении *<exp>* в любом месте, где допустим литерал, может встречаться ссылка на атрибут *<attribute ref>*. Части *<exp>* не должно быть, если в роли *<agg op name>* выступает COUNT; в противном случае ее можно опускать, только если реляционное выражение *<relation exp>* обозначает отношение степени 1, и тогда предполагается, что *<exp>* состоит из ссылки на единственный атрибут этого отношения. Примеры:

1. `SUM ( SP , QTY )`

   Это выражение обозначает сумму всех значений QTY в переменной-отношении SP (на наших тестовых данных получится 3100).

2. `SUM ( SP { QTY } )`

   Это сокращенная запись SUM(SP{QTY},QTY), которая обозначает сумму всех различных значений QTY в переменной-отношении SP (то есть 1000).

3. `AVG ( SP , 3 * QTY )`

   Это выражение отвечает на вопрос, какова была бы средняя величина поставки, если бы количество деталей в каждой поставке увеличилось втрое (ответ 775). Более общо, выражение

   ```
   agg ( rx , x )
   ```

   (где *x* – некоторое выражение, более сложное, чем просто *<attribute ref>*) по существу является сокращенной записью для:

   ```
   agg ( EXTEND rx ADD ( x AS y ) , y )
   ```

Теперь я возвращаюсь к SQL. Для удобства повторю приведенные выше примеры:

```
X := COUNT ( S ) ;              |   SELECT COUNT ( * ) AS X
                                |   FROM   S

Y := COUNT ( S { STATUS } ) ;   |   SELECT COUNT ( DISTINCT STATUS )
                                |          AS Y
                                |   FROM   S
```

Возможно, вам будет странно слышать от меня, что SQL в действительности вообще не поддерживает агрегатные операторы! Я заявляю об этом, прекрасно понимая, что многие сочтут выражения справа в точности вызовами агрегатных операторов в SQL.[1] Но это не так. И я готов объяснить. Как мы знаем, значения этих счетчиков равны соответственно 5 и 3. Но эти SQL-выражения не вычисляют сами счетчики, как должны были бы делать настоящие агрегатные операторы, а вычисляют таблицы, содержащие значения счетчиков. Точнее, каждый вызов порождает таблицу с одной строкой и одним столбцом, и единственным значением в этой строке будет счетчик:

X
5

Y
3

Как видите, выражения SELECT не являются вызовами агрегатных операторов; в лучшем случае можно сказать, что они дают аппроксимацию таких вызовов. На самом деле агрегирование в SQL рассматривается как частный случай *обобщения*. Правда, я еще не касался обобщения в этой главе, но пока вы можете считать, что оно представляется в SQL выражением SELECT с фразой GROUP BY. Конечно, в приведенных выше SQL-выражениях фразы GROUP BY нет, но по определению они являются сокращенной записью таких выражений (и потому все же являются частными случаями обобщения, как и утверждалось):

```
SELECT COUNT ( * ) AS X
FROM   S
GROUP  BY ( )

SELECT COUNT ( DISTINCT STATUS ) AS Y
FROM   S
GROUP  BY ( )
```

Примечание

На случай, если эти выражения вас смущают, объясню, что в SQL разрешается (а) заключать списки операндов GROUP BY в скобки и (б) включать фразы GROUP BY без операндов. Задание GROUP BY без операндов трактуется так, будто этой фразы нет вовсе.

SQL поддерживает обобщение, но не агрегирование как таковое. Печально, но эти понятия часто смешивают, и, наверное, теперь вы понимаете почему. Но картина запутывается еще больше из-за того, что

[1] Можно сказать, с несколько большим основанием, что *вызовы COUNT в этих выражениях* – это вызовы агрегатных операторов. Но суть в том, что в SQL такой вызов не может выступать в роли «независимого» выражения; он обязательно должен быть частью какого-то табличного выражения.

в SQL нередко приводят таблицу, являющуюся результатом «агрегирования» к одной строке, которую она содержит, или даже к тому единственному значению, которое имеется в этой строке. Две разных ошибки (по крайней мере, концептуальных) – а в результате странное «агрегирование» становится больше похоже на настоящее! Такое двойное приведение типа происходит, в частности, когда выражение SELECT заключено в скобки и образует скалярный подзапрос, как, например, в следующих присваиваниях:

```
SET X = ( SELECT COUNT ( * ) FROM S ) ;

SET Y = ( SELECT COUNT ( DISTINCT STATUS ) FROM S ) ;
```

Но присваивание – далеко не единственный контекст, в котором возможно подобное приведение типа (см. главы 2 и 12).

Отступление

На самом деле, с агрегированием в духе SQL связана еще одна странность (я поместил это замечание сюда, так как оно сюда логически относится, однако оно опирается на понимание идеи обобщения в SQL, поэтому сейчас можете его пропустить):

- В общем случае выражение вида SELECT - FROM *T* - WHERE - GROUP BY - HAVING порождает результат, содержащий по одной строке для каждой «группы» в *G*, где *G* – «сгруппированная таблица», получающаяся путем применения фраз WHERE, GROUP BY и HAVING к таблице *T*.
- Отсутствие фраз WHERE и HAVING, характерное для типичного агрегирования в SQL, эквивалентно заданию фраз WHERE TRUE и HAVING TRUE соответственно. Поэтому сейчас нам достаточно рассмотреть только влияние фразы GROUP BY на вычисление сгруппированной таблицы *G*.
- Предположим, что таблица *T* состоит из *nT* строк. При объединении этих строк в группы может получиться не более *nT* групп. Иными словами, сгруппированная таблица *G* состоит из *nG* групп, где $nG \leq nT$, а окончательный результат применения фразы SELECT к *G* состоит из *nG* строк.
- Теперь предположим, что *nT* равно нулю (то есть таблица *T* пуста); тогда *nG*, очевидно, также должно быть равно нулю (то есть таблица *G* и, стало быть, результат выражения SELECT тоже будут пусты).
- Поэтому, в частности, выражение

```
SELECT COUNT ( * ) AS X
FROM   S
GROUP  BY ( )
```

которое, напомню, является полной записью SELECT COUNT(*) AS X FROM S, по логике вещей, должно порождать результат, показанный слева, а не справа, если таблица S пуста.

X

X
0

Но на самом деле получается результат, показанный справа. Почему? *Ответ:* это особый случай. Вот прямая цитата из стандарта: «Если не заданы столбцы, по которым производится группировка, то результатом фразы <group by clause> является сгруппированная таблица, содержащая *T* в качестве единственной группы». Иными словами, хотя группировка пустой таблицы в SQL действительно (как я и доказывал выше) порождает в общем случае пустое множество групп, однако *случай, когда множество столбцов группировки пусто, считается особым*; в этом случае порождается множество, содержащее ровно одну группу, которая идентична пустой таблице T. Таким образом, в нашем примере оператор COUNT применяется к пустой группе и, значит, «корректно» возвращает значение 0.

Возможно, вам кажется, что в этом несоответствии нет ничего ужасного; может быть, вы даже думаете, что правый результат чем-то «лучше» левого. Но (и это очевидно) между тем и другим есть логическое различие, а – снова цитируя Виттгенштейна, – «*любое логическое различие является существенным различием*». Такого рода логические ошибки попросту недопустимы в системе, которая, подобно реляционной системе, должна покоиться на твердых логических основаниях.

Еще об отношениях-образах

В этом разделе я хотел бы привести ряд примеров, демонстрирующих полезность агрегатных операторов, рассмотренных в предыдущем разделе, в применении к отношениям-образам. *Примечание*: Во всех примерах в каком-то виде, пусть неявном, присутствует обобщение, но пока я еще не готов обсуждать его детально (это будет темой следующих двух разделов).

Пример 1. Получить поставщиков, для которых количество деталей в поставке, просуммированное по всем поставкам данного поставщика, меньше 1000.

```
S WHERE SUM ( !!SP , QTY ) < 1000
```

Для любого поставщика выражение SUM(!!SP,QTY) обозначает в точности общее количество деталей, поставляемых этим поставщиком. Эквивалентная формулировка без использования отношения-образа такова:

```
S WHERE SUM ( SP MATCHING RELATION { TUPLE { SNO SNO } } , QTY ) < 1000
```

Ради интереса приведу «аналог» на SQL – я взял слово «аналог» в кавычки, потому что в этом примере есть подводный камень: показанное SQL-выражение не является точным эквивалентом выражений **Tutorial D** выше (почему?):

```
SELECT S.*
FROM   S , SP
WHERE  S.SNO = SP.SNO
```

```
GROUP  BY S.SNO , S.SNAME , S.STATUS , S.CITY
HAVING SUM ( SP.QTY ) < 1000
```

Примечание

Не могу не заметить мимоходом, что (как видно из примера выше) SQL допускает запись «S.*» во фразе SELECT, но не во фразе GROUP BY, где она имела бы ничуть не меньший смысл.

Пример 2. Получить поставщиков, для которых существует меньше трех поставок.

```
S WHERE COUNT ( !!SP ) < 3
```

Пример 3. Получить поставщиков, для которых максимальное поставленное количество менее чем в два раза превышает минимальное поставленное количество (в обоих случаях речь идет обо всех поставках данного поставщика).

```
S WHERE MAX ( !!SP , QTY ) < 2 * MIN ( !!SP , QTY )
```

Пример 4. Получить поставки, для которых существует еще по меньшей мере две поставки с таким же количеством.

```
SP WHERE COUNT ( !!(SP RENAME ( SNO AS SN , PNO AS PN )) ) > 2
```

Признаю, что это очень искусственный пример, но смысл его в том, чтобы показать, что иногда при использовании отношений-образов возникает необходимость в переименовании атрибутов. В этом примере переименование необходимо, чтобы нужное нам отношение-образ, связанное с данным кортежем поставки, было определено только в терминах атрибута QTY. Имена SN и PN выбраны произвольно.

Отмечу попутно, что этот пример иллюстрирует также «множественную» форму RENAME:

```
SP RENAME ( SNO AS SN , PNO AS PN )
```

Это выражение – сокращенная запись такого:

```
( SP RENAME ( SNO AS SN ) ) RENAME ( PNO AS PN )
```

Похожие сокращения определены и для разных других операторов, в том числе EXTEND (пример будет приведен ниже).

Пример 5. Для всех поставщиков, для которых общее количество, просуммированное по всем поставкам данного поставщика, меньше 1000, уменьшить значение статуса в два раза.

```
UPDATE S WHERE SUM ( !!SP , QTY ) < 1000 :
                    { STATUS := 0.5 * STATUS } ;
```

Обобщение

> **Определение:** Пусть отношения *r1* и *r2* таковы, что заголовок *r2* совпадает с заголовком некоторой проекции *r1*, и пусть *A, B, ..., C* – атрибуты *r2*. Тогда *обобщением* SUMMARIZE *r1* PER (*r2*) ADD (*summary* AS X) называется отношение, для которого (а) заголовком является заголовок *r2*, дополненный атрибутом *X*, и (б) тело состоит из множества всех кортежей *t* таких, что *t* является кортежем *r2*, дополненным значением *x* атрибута *X*. Это значение *x* подсчитывается путем вычисления некоторого *обобщающего выражения* по всем кортежам *r1*, которые имеют те же значения атрибутов *A, B, ..., C*, что и кортеж *t*. Отношение *r2* не должно содержать атрибута с именем *X*, а *обобщающее выражение* не должно ссылаться на *X*. Отметим, что кардинальность результата равна кардинальности *r2*, а степень результата равна степени *r2* плюс единица. Типом *X* в этом случае будет тип выражения *exp*.

Вот пример (который я для последующих ссылок назову SX1 – «SUMMARIZE Example 1»):

```
SUMMARIZE SP PER ( S { SNO } ) ADD ( COUNT ( PNO ) AS PCT )
```

Для наших тестовых данных получается такой результат:

SNO	PCT
S1	6
S2	2
S3	1
S4	3
S5	0

Иными словами, результат содержит по одному кортежу для каждого кортежа в отношении PER – то есть в этом примере по одному кортежу для каждого из пяти номеров поставщиков, – дополненному соответствующим счетчиком.

Отступление

Обратите особое внимание, что синтаксическая конструкция COUNT(PNO) – я сознательно не называю ее выражением, потому что она таковым не является (по крайней мере, не в смысле языка **Tutorial D**), – в предыдущем вызове SUMMARIZE – это *не* вызов агрегатного оператора с именем COUNT. Агрегатный оператор принимает в качестве аргумента отношение, а аргумент COUNT – атрибут; конечно, это атрибут некоторого отношения, но само отношение определено лишь косвенно. На самом деле синтаксическая конструкция COUNT(PNO) представляет собой особый случай – она не имеет никакого

смысла вне контекста подходящего вызова SUMMARIZE и не может встречаться вне такого контекста. Все это наводит на мысль, что операция SUMMARIZE в каком-то смысле неполноценна и хорошо было бы заменить ее чем-то получше... См. раздел «Еще об обобщении» ниже.

Если отношение *r2* не просто имеет такой же заголовок, как некоторая проекция отношения *r1*, а является такой проекцией, то обобщение можно записать короче, заменив спецификацию PER спецификацией BY, как в следующем примере («Пример SX2»):

```
SUMMARIZE SP BY { SNO } ADD ( COUNT ( PNO ) AS PCT )
```

Результат получается таким:

SNO	PCT
S1	6
S2	2
S3	1
S4	3

Как видите, результат отличается от предыдущего – он не содержит кортежа для поставщика S5. Объясняется это тем, что BY {SNO} в этом примере, по определению, является сокращением для PER (SP{SNO}) – SP, потому что именно по SP мы и хотим произвести обобщение, – а проекция SP{SNO} не содержит кортежа для поставщика S5.

Пример SX2 можно записать на SQL следующим образом:

```
SELECT SNO , COUNT ( ALL PNO ) AS PCT
FROM   SP
GROUP  BY SNO
```

Мы видим, что обобщения – в противоположность «агрегированию» – обычно формулируются на SQL посредством выражения SELECT с явной фразой GROUP BY (однако см. ниже!). Вот на какие мысли наводит этот пример.

- Можно считать, что такие выражения вычисляются в следующем порядке. Сначала таблица, заданная фразой FROM, разбивается на множество непересекающихся «групп» – фактически таблиц, – в соответствии со столбцами группировки во фразе GROUP BY. Затем формируются результирующие строки, по одной для каждой группы: вычисляется заданное обобщающее выражение (или выражения) для этой группы и в конец дописываются другие элементы, заданные в списке SELECT. *Примечание*: В SQL аналогом термина *обобщающее выражение* служит «функция множества» (set function) – термин вдвойне неподходящий, потому что (а) аргументом такой функции является не множество, а мультимножество, и (б) результатом тоже является не множество.

- В данном примере можно без опаски употреблять SELECT, а не SELECT DISTINCT, так как (а) гарантируется, что результирующая таблица содержит в точности одну строку для каждой группы (по определению) и (б) каждая группа содержит по одному значению для каждого столбца группировки (опять-таки по определению).
- В данном примере при вызове COUNT спецификатор ALL можно опустить, так как для функций множеств его наличие предполагается по умолчанию. (Фактически, в этом примере неважно, указан ли спецификатор ALL или DISTINCT, потому что сочетание номера поставщика и номера детали является ключом для таблицы SP).
- Функция множества COUNT(*) представляет собой особый случай – она применяется не к значениям в некотором столбце (как, например, SUM), а к строкам таблицы. (В данном примере COUNT(PNO) можно было бы заменить на COUNT(*), и результат не изменился бы.)

А теперь вернемся к примеру SX1. На SQL его можно записать следующим образом:

```
SELECT S.SNO , ( SELECT COUNT ( ALL PNO )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS PCT
FROM   S
```

Здесь важно отметить, что теперь результат содержит строку для поставщика S5, потому что, по определению, результат должен содержать по одной строке для каждого номера поставщика, присутствующего в таблице S. И, как легко видеть, эта формулировка отличается от примера SX2 – того, где поставщик S5 был пропущен, – тем, что отсутствует фраза GROUP BY и не производится никакая группировка (по крайней мере, явно).

Отступление

Кстати, читателей, не искушенных в SQL, тут поджидает некий подвох. Как видите, второй элемент списка SELECT в приведенном выше SQL-выражении – то есть подвыражение (SELECT ... S.SNO) AS PCT – имеет вид *подзапрос* AS *имя* (и сам подзапрос скалярный). Если бы точно такое же выражение встретилось во фразе FROM, то спецификатор AS *имя* следовало бы понимать как определение переменной кортежа (range variable) (см. главу 10). Во фразе же SELECT этот спецификатор интерпретируется как определение имени столбца результата. Отсюда следует, что следующая формулировка этого примера логически не эквивалентна приведенной выше:

```
SELECT S.SNO , ( SELECT COUNT ( ALL PNO ) AS PCT
                 FROM   SP
                 WHERE  SP.SNO = S.SNO )
FROM   S
```

При такой формулировке таблица *t*, которая возвращается в результате вычисления подзапроса, имеет столбец PCT. Для этой таблицы *t* производится

двойное приведение типа к тому единственному скалярному значению, которое в ней содержится, в результате чего порождается значение столбца в конечной таблице, и – хотите верьте, хотите нет – этот столбец не называется PCT, он вообще не имеет имени.

Но вернемся к основной теме обсуждения. Фактически, пример SX2 можно было бы записать и без использования GROUP BY следующим образом:

```
SELECT DISTINCT SPX.SNO , ( SELECT COUNT ( ALL SPY.PNO )
                            FROM   SP AS SPY
                            WHERE  SPY.SNO = SPX.SNO ) AS PCT
FROM   SP AS SPX
```

Как следует из этих примеров, фраза GROUP BY в SQL логически избыточна – любое реляционное выражение, которое можно представить с ее помощью, можно записать и без нее. Однако в связи с этим следует сделать еще одно замечание. Предположим, что в примере SX1 требуется подсчитать не общее число номеров деталей, а суммарное количество, поставленное каждым поставщиком:

```
SUMMARIZE SP PER ( S { SNO } ) ADD ( SUM ( QTY ) AS TOTQ )
```

На наших тестовых данных результат будет такой:

SNO	TOTQ
S1	1300
S2	700
S3	200
S4	900
S5	0

Но SQL-выражение

```
SELECT S.SNO , ( SELECT SUM ( ALL QTY )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS TOTQ
FROM   S
```

дает результат, в котором значение TOTQ для поставщика S5 равно null, а не 0. Так получается потому, что в SQL применение любой функции множества, кроме COUNT(*) и COUNT, к пустому аргументу, по определению (неправильному) дает null. Чтобы получить правильный результат, мы должны воспользоваться функцией COALESCE:

```
SELECT S.SNO , ( SELECT COALESCE ( SUM ( ALL QTY ) , 0 )
                 FROM   SP
                 WHERE  SP.SNO = S.SNO ) AS TOTQ
FROM   S
```

Предположим теперь, что в примере SX1 требуется найти суммарное количество для каждого поставщика, но только если оно больше 250:

```
( SUMMARIZE SP PER ( S { SNO } ) ADD ( SUM ( QTY ) AS TOTQ ) )
                                                    WHERE TOTQ > 250
```

Результат:

SNO	TOTQ
S1	1300
S2	700
S4	900

«Естественная» формулировка этого запроса на SQL выглядела бы так:

```
SELECT SNO , SUM ( ALL QTY ) AS TOTQ
FROM   SP
GROUP  BY SNO
HAVING SUM ( ALL QTY ) > 250  /* не TOTQ > 250 !!! */
```

Но можно переписать его и в таком виде:

```
SELECT DISTINCT SPX.SNO , ( SELECT SUM ( ALL SPY.QTY )
                            FROM   SP AS SPY
                            WHERE  SPY.SNO = SPX.SNO ) AS TOTQ
FROM   SP AS SPX
WHERE  ( SELECT SUM ( ALL SPY.QTY )
         FROM   SP AS SPY
         WHERE  SPY.SNO = SPX.SNO ) > 250
```

Как видно из этого примера, фраза HAVING, как и GROUP BY, тоже логически избыточна – любое реляционное выражение, которое можно представить с ее помощью, можно записать и без нее. Да, версии с GROUP BY и HAVING часто более компактны; с другой стороны, верно и то, что иногда они дают неправильные ответы (подумайте, например, что произойдет, если в предыдущем примере требуется, чтобы суммарное количество было не больше, а меньше 250).

Рекомендации: Применяя фразы GROUP BY и HAVING, убедитесь, что вы обобщаете именно ту таблицу, которую хотите обобщить (в примерах из этого раздела – таблицу поставщиков, а не поставок). Кроме того, не забывайте о возможности обобщения по пустому множеству и включайте COALESCE там, где необходимо.

Есть и еще одна вещь, о которой я хочу сказать в связи с GROUP BY и HAVING. Рассмотрим следующее SQL-выражение:

```
SELECT SNO , CITY , SUM ( ALL QTY ) AS TOTQ
FROM   S NATURAL JOIN SP
GROUP  BY SNO
```

Заметим, что CITY встречается с списке SELECT, но не входит в состав столбцов группировки. Но такая ситуация допустима, потому что

таблица S удовлетворяет некоторой *функциональной зависимости* (см. главу 8 или приложение B), согласно которой каждому значению SNO в этой таблице соответствует ровно одно значение CITY (в ней же). Более того, в стандарте SQL есть правила, согласно которым система знает о наличии такой функциональной зависимости. Поэтому, несмотря на отсутствие CITY в составе столбцов группировки, известно, что в каждой группе значение CITY одно и то же, так что CITY может встречаться во фразе SELECT, как показано выше (и во фразе HAVING, если бы таковая присутствовала).

Разумеется, логически допустимо (хотя это может негативно отразиться на производительности) включить этот столбец в состав столбцов группировки:

```
SELECT SNO , CITY , SUM ( QTY ) AS TOTQ
FROM   S NATURAL JOIN SP
GROUP  BY SNO , CITY
```

Еще об обобщении

Оператор SUMMARIZE входил в состав языка **Tutorial D** с самого начала. Но с появлением отношений-образов он стал логически избыточным, и хотя, возможно, есть причины сохранить его (например, в педагогических целях), факт остается фактом: как правило, обобщение более компактно записываются с помощью оператора EXTEND.[1] Вспомните пример SX1 из предыдущего раздела («Для каждого поставщика получить номер поставщика и количество поставленных им деталей»). Формулировка с применением SUMMARIZE выглядит так:

```
SUMMARIZE SP PER ( S { SNO } ) ADD ( COUNT ( PNO ) AS PCT )
```

А вот как выглядит эквивалентное выражение с применением оператора EXTEND:

```
EXTEND S { SNO } ADD ( COUNT ( !!SP ) AS PCT )
```

(Поскольку комбинация {SNO,PNO} является ключом переменной-отношения SP, необязательно проецировать отношение-образ на {PNO} перед вычислением счетчика.) Как видно из этого примера, оператор EXTEND дает еще один контекст, в котором отношения-образы имеют смысл; пожалуй, в этом контексте они даже полезнее, чем во фразах WHERE.

Далее в этом разделе приводятся дополнительные примеры. Я продолжаю нумерацию, начатую в разделе «Еще об отношениях-образах».

---

[1] Не говоря уже о том, что, как отмечалось выше в разделе об обобщении, в операторе SUMMARIZE по необходимости используется синтаксическая конструкция, которая, к несчастью, похожа на вызов агрегатного оператора, хотя таковым не является. Поэтому, как уже было сказано, было бы неплохо расстаться с оператором SUMMARIZE навсегда.

Пример 6. Для каждого поставщика получить подробную информацию о нем и общее количество поставленных им деталей, просуммированное по всем его поставкам.

```
EXTEND S ADD ( SUM ( !!SP , QTY ) AS TOTQ )
```

Пример 7. Для каждого поставщика получить подробную информацию о нем и общее, максимальное и минимальное количество деталей по всем его поставкам.

```
EXTEND S ADD ( SUM ( !!SP , QTY ) AS TOTQ ,
               MAX ( !!SP , QTY ) AS MAXQ ,
               MIN ( !!SP , QTY ) AS MINQ )
```

Обратите внимание на множественную форму EXTEND в этом примере.

Пример 8. Для каждого поставщика получить подробную информацию о нем, общее количество деталей, просуммированное по всем его поставкам, и общее количество деталей, просуммированное по всем поставкам всех поставщиков.

```
EXTEND S ADD ( SUM ( !!SP , QTY ) AS  TOTQ ,
               SUM (   SP , QTY ) AS GTOTQ )
```

Результат:

SNO	TOTQ	GTOTQ
S1	1300	3100
S2	700	3100
S3	200	3100
S4	900	3100
S5	0	3100

Пример 9. Для каждого города *c* получить *c* и общее и среднее количество деталей в поставке для всех поставок, для которых и поставщик, и деталь находятся в городе *c*.

```
WITH ( S JOIN SP JOIN P ) AS TEMP :
EXTEND TEMP { CITY } ADD ( SUM ( !!TEMP , QTY ) AS TOTQ ,
                           AVG ( !!TEMP , QTY ) AS AVGQ )
```

Смысл этого, довольно искусственного, примера – проиллюстрировать полезность WITH в сочетании с обобщением в случае, когда нужно несколько раз вывести некоторое, возможно, длинное подвыражение.

Группирование и разгруппирование

Напомню, что в главе 2 говорилось о допустимости атрибутов, значениями которых являются отношения (RVA-атрибутов). На рис. 7.1 показаны отношения R1 и R4 с рисунков 2.1 и 2.2; R4 имеет RVA-атрибуты, а R1 – нет, но очевидно, что оба отношения несут в себе одну и ту же информацию.