Т. Кайт - Аналитические функции (1)

SQL — очень мощный язык и лишь очень немногие запросы в нем нельзя создать.

Ряд запросов, которые сложно сформулировать на обычном языке SQL, весьма типичны:

• **Подсчет промежуточной суммы.** Показать суммарную зарплату сотрудников отдела построчно, чтобы в каждой строке выдавалась сумма зарплат всех сотрудников вплоть до указанного.

• **Подсчет процентов в группе.** Показать, какой процент от общей зарплаты по отделу составляет зарплата каждого сотрудника. Берем его зарплату и делим на сумму зарплат по отделу.

• **Запросы первых N.** Найти N сотрудников с наибольшими зарплатами или N наиболее продаваемых товаров по регионам.

• **Подсчет скользящего среднего.** Получить среднее значение по текущей и предыдущим N строкам.

• **Выполнение ранжирующих запросов.** Показать относительный ранг зарплаты сотрудника среди других сотрудников того же отдела.

Аналитические функции, появившиеся в версии Oracle 8.1.6, создавались для решения именно этих задач. Они расширяют язык SQL так, что подобные операции не только проще записываются, но и быстрее выполняются по сравнению с использованием чистого языка SQL. Эти расширения сейчас изучаются комитетом ANSI SQL с целью включения в спецификацию языка SQL.

Начнем эту главу с примера, демонстрирующего возможности аналитических функций. После этого будет представлен полный синтаксис и описание всех функций, а так же ряд практических примеров выполнения перечисленных выше операций. Как обычно, в конце будут рассмотрены потенциальные проблемы использования аналитических функций.

Пример

Простой пример подсчета промежуточной суммы зарплат по отделам с описанием того, что же в действительности происходит, позволят получить начальное представление о принципах использования аналитических функций:

```
SELECT ename, deptno, sal,
       SUM(sal) OVER (ORDER BY deptno, ename) running_total,
       SUM(sal) OVER (PARTITION BY deptno ORDER BY ename) department_total,
       ROW_NUMBER() OVER (PARTITION BY deptno ORDER BY ename) seq
FROM scott.emp
ORDER BY deptno, ename
```

ENAME	DEPTNO	SAL	RUNNING_TOTAL	DEPARTMENT_TOTAL	SEQ
CLARK	10	2450	2450	2450	1
KING	10	5000	7450	7450	2
MILLER	10	1300	8750	8750	3
ADAMS	20	1100	9850	1100	1
FORD	20	3000	12850	4100	2
JONES	20	2975	15825	7075	3
SCOTT	20	3000	18825	10075	4
SMITH	20	800	19625	10875	5
ALLEN	30	1600	21225	1600	1
BLAKE	30	2850	24075	4450	2
JAMES	30	950	25025	5400	3
MARTIN	30	1250	26275	6650	4
TURNER	30	1500	27775	8150	5
WARD	30	1250	29025	9400	6

В представленном выше коде удалось получить значение running_total для запроса в целом. Это было сделано по всему упорядоченному результирующему множеству с помощью конструкции **SUM**(sal) **OVER** (**ORDER BY** deptno, ename). Также удалось подсчитать промежуточные суммы по отделам, сбрасывая их в ноль при переходе к следующему отделу. Этого удалось добиться благодаря конструкции **PARTITION BY** deptno в **SUM**(sal) — в запросе была указана конструкция, задающая условие разбиения данных на группы. Для последовательной нумерации строк в каждой группе, в соответствии с критериями упорядочения, использовалась функция **ROW_NUMBER**() (для выдачи этого номера строки был добавлен столбец seq). В результате видно, что SCOTT — четвертый по списку сотрудник в отделе 20 при упорядочении по фамилии (ename). Функция **ROW_NUMBER**() используется и во многих других ситуациях, например для транспонирования или преобразования результирующих множеств (как будет описано далее).

Этот новый набор функциональных возможностей содержит много замечательного.

Он открывает абсолютно новые перспективы работы с данными. Можно избавиться от большого объема процедурного кода и сложных (или неэффективных) запросов, требующих много времени на разработку, и получить при этом желаемый результат.

Т. Кайт - Аналитические функции (2)

Синтаксис

Синтаксис вызова аналитической функции на вид весьма прост, но эта простота может быть обманчивой. Все начинается с такой конструкции:

ИМЯ_ФУНКЦИИ(<аргумент>,< аргумент >, . . .)

OVER (<конструкция_фрагментации><конструкция_упорядочения><конструкция_окна>)

Вызов аналитической функции может содержать до четырех частей: аргументы, конструкция фрагментации, конструкция упорядочения и конструкция, задающая окно. В представленном выше примере:

SUM(sal) **OVER** (**PARTITION BY** deptno **ORDER BY** ename) department_total

• **SUM** — имя функции.

• (sal) — аргумент аналитической функции. Аналитические функции принимают от нуля до трех аргументов. В качестве аргументов передаются выражения, т.е. вполне можно было бы использовать **SUM**(sal + comm).

• **OVER** — ключевое слово, идентифицирующее эту функцию как аналитическую.

В противном случае синтаксический анализатор не мог бы отличить функцию агрегирования **SUM()** от аналитической функции **SUM()**. Конструкция после ключевого слова **OVER** описывает срез данных, "по которому" будет вычисляться аналитическая функция.

• **PARTITION BY** deptno — необязательная конструкция фрагментации. Если конструкция фрагментации не задана, все результирующее множество считается одним большим фрагментом. Это используется для разбиения результирующего множества на группы, так что аналитическая функция применяется к группам, а не ко всему результирующему множеству. В первом примере главы, когда конструкция фрагментации не указывалась, функция **SUM** по столбцу sal вычислялась для всего результирующего множества. Фрагментируя результирующее множество по столбцу deptno, мы вычисляли **SUM** по столбцу sal для каждого отдела (deptno), сбрасывая промежуточную сумму для каждой группы.

• **ORDER BY** ename — необязательная конструкция **ORDER BY**; для некоторых функций она обязательна, для других — нет. Функции, зависящие от упорядочения данных, например **LAG** и **LEAD**, которые позволяют обратиться к предыдущим и следующим строкам в результирующем множестве, требуют обязательного указания конструкции **ORDER BY**. Другие функции, например **AVG**, не требуют. Эта конструкция обязательна, если используется любая функция работы с окном (подробнее см. далее в разделе "Конструкция окна"). Конструкция **ORDER BY** определяет, как упорядочиваются данные в группах при вычислении аналитической функции. В нашем случае упорядочивать по deptno и ename не нужно, потому что по столбцу deptno выполнялась фрагментация, т.е. неявно предполагается, что столбцы, по которым выполняется фрагментация, по определению входят в ключ сортировки (конструкция **ORDER BY** применяется к каждому фрагменту поочередно).

• **Конструкция окна в данном примере отсутствует.** Именно ее синтаксис иногда кажется сложным. Подробно возможные способы задания конструкции окна будут рассмотрены ниже.

Теперь более детально рассмотрим каждую из четырех частей вызова аналитической функции, чтобы понять, как их можно задавать.

Функции

Сервер Oracle предлагает 26 аналитических функций. Они разбиваются на **четыре основных класса** по возможностям.

Первый класс образуют различные функции *ранжирования*, позволяющие строить запросы типа "первых N". Мы уже использовали одну функцию этого класса, **ROW_NUMBER**, при генерации столбца seq в предыдущем примере. Она ранжировала сотрудников в отделах по фамилии (ename). Точно так же их можно было бы ранжировать по зарплате (sal) или любому другому атрибуту.

Второй класс образуют *оконные* функции, позволяющие вычислять разнообразные агрегаты. В первом примере этой главы была показана такая функция — мы вычисляли **SUM**(sal) по разным группам. Вместо функции **SUM** можно было использовать и другие функции агрегирования, например **COUNT**, **AVG**, **MIN**, **MAX** и т.д.

К **третьему классу** относятся различные *итоговые* функции. Они очень похожи на оконные, поэтому имеют те же имена: **SUM**, **MIN**, **MAX** и т.д. Тогда как оконные функции используются для работы с окнами данных, как промежуточная сумма в предыдущем примере, итоговые функции работают со всеми строками фрагмента или группы.

Например, если бы в первоначальном запросе использовались обращения:

```
SUM(sal) OVER () total_salary,
SUM(sal) OVER (PARTITION BY deptno) total_salary_for_department
```

мы бы получили общие суммы по группам, а не промежуточные. Ключевое отличие итоговой функции от оконной — отсутствие конструкции **ORDER BY** в операторе **OVER**. При отсутствии конструкции **ORDER BY** функция применяется к каждой строке группы. При наличии конструкции **ORDER BY** функция применяется к окну (подробнее об этом в разделе, описывающем конструкцию окна).

Есть также функции **LAG** и **LEAD**, позволяющие получать значения из предыдущих или следующих строк результирующего множества. Это помогает избежать само соединения данных. Например, если в таблице записаны даты визитов пациентов к врачу и необходимо вычислить время между визитами для каждого их них, очень пригодится функция **LAG**. Можно просто фрагментировать данные по пациентам и отсортировать их по дате. После этого функция **LAG** легко сможет вернуть данные предыдущей записи для пациента. Останется вычесть из одной даты другую. До появления аналитических функций для получения этих данных приходилось организовывать сложное соединение таблицы с ней же самой.

Четвертый класс. Это большой класс *статистических* функций, таких как **VAR_POP**, **VAR_SAMP**, **STDEV_POP**, набор функций линейной регрессии и т.п. Эти функции позволяют вычислять значения статистических показателей для любого неупорядоченного фрагмента.

В конце раздела, посвященного синтаксису, представлена таблица с кратким объяснением назначения всех аналитических функций.

Т. Кайт - Аналитические функции (3)

Конструкция фрагментации

Конструкция **PARTITION BY** логически разбивает результирующее множество на N групп по критериям, задаваемым выражениями фрагментации. Слова "фрагмент" и "группа" в этой главе и в документации Oracle используются как синонимы. Аналитические функции применяются к каждой группе независимо, — для каждой новой группы они сбрасываются. Например, ранее при демонстрации функции, вычисляющей промежуточную сумму зарплат, фрагментация выполнялась по столбцу deptno. Когда значение в столбце deptno в результирующем множестве изменялось, происходил сброс промежуточной суммы в ноль, и суммирование начиналось заново.

Если не указать конструкцию фрагментации, все результирующее множество считается одной группой. Во первом примере мы использовали функцию **SUM(sal)** без конструкции фрагментации, чтобы получить промежуточные суммы для всего результирующего множества.

Интересно отметить, что каждая аналитическая функция в запросе может иметь уникальную конструкцию фрагментации; фактически уже в простейшем примере в начале главы это и было сделано. Для столбца running_total конструкция фрагментации не была задана, поэтому целевой группой было все результирующее множество. Для столбца departmental_total результирующее множество фрагментируется по отделам, что позволило вычислять промежуточные суммы для каждого из них.

Синтаксис конструкции фрагментации прост и очень похож на синтаксис конструкции **GROUP BY** в обычных SQL-запросах:

```
PARTITION BY выражение [, выражение] [, выражение]
```

Конструкция упорядочения

Конструкция **ORDER BY** задает критерий сортировки данных в каждой группе (в каждом фрагменте). Это, несомненно, влияет на результат выполнения любой аналитической функции. При наличии (или отсутствии) конструкции **ORDER BY** аналитические функции вычисляются по-другому. В качестве примера рассмотрим, что происходит при использовании функции **AVG()** с конструкцией **ORDER BY** и без оной:

```
SELECT ename, sal, AVG(sal) OVER ()
FROM scott.emp
```

ENAME	SAL	AVG(SAL)OVER()
KING	5000,00	2073,214286
BLAKE	2850,00	2073,214286
CLARK	2450,00	2073,214286
JONES	2975,00	2073,214286
MARTIN	1250,00	2073,214286
ALLEN	1600,00	2073,214286
TURNER	1500,00	2073,214286
JAMES	950,00	2073,214286

WARD	1250,00	2073,214286
FORD	3000,00	2073,214286
SMITH	800,00	2073,214286
SCOTT	3000,00	2073,214286
ADAMS	1100,00	2073,214286
MILLER	1300,00	2073,214286

```
SELECT ename, sal, AVG(sal) OVER (ORDER BY ename)
FROM scott.emp
ORDER BY ename
```

ENAME	SAL	AVG(SAL)OVER(ORDERBYENAME)
ADAMS	1100,00	1100
ALLEN	1600,00	1350
BLAKE	2850,00	1850
CLARK	2450,00	2000
FORD	3000,00	2200
JAMES	950,00	1991,666667
JONES	2975,00	2132,142857
KING	5000,00	2490,625
MARTIN	1250,00	2352,777778
MILLER	1300,00	2247,5
SCOTT	3000,00	2315,909091
SMITH	800,00	2189,583333
TURNER	1500,00	2136,538462
WARD	1250,00	2073,214286

В отсутствие конструкции **ORDER BY** среднее значение вычисляется по всей группе, и одно и то же значение выдается для каждой строки (функция используется как итоговая). Когда функция **AVG**(sal) используется с конструкцией **ORDER BY**, среднее значение в каждой строке является средним по текущей и всем предыдущим строкам (функция используется как оконная). Например, средняя зарплата для пользователя ALLEN в результатах выполнения запроса с конструкцией **ORDER BY** — 1350 (среднее для значений 1100 и 1600).

*Немного забегая вперед, в следующий раздел, посвященный конструкции окна, можно сказать, что наличие конструкции **ORDER BY** в вызове аналитической функции добавляет стандартную конструкцию окна — **RANGE UNBOUNDED PRECEDING**.*

*Это означает, что для вычисления используется набор из всех предыдущих и текущей строки в текущем фрагменте. При отсутствии **ORDER BY** стандартным окном является весь фрагмент.*

Чтобы реально почувствовать, как все это работает, рекомендую применить одну и ту же аналитическую функцию при двух различных конструкциях **ORDER BY**. В первом примере текущая сумма вычисляется для всей таблицы emp с использованием конструкции **ORDER BY** deptno, ename. При этом текущая сумма вычисляется для всех строк, причем, порядок их просмотра определяется конструкцией **ORDER BY**. Если изменить порядок указания столбцов в этой конструкции на противоположный или вообще сортировать по другим столбцам, получаемые текущие суммы будут существенно отличаться; общая сумма в последней строке совпадет, но все промежуточные значения будут другими. Например:

```
SELECT ename, deptno,
SUM(sal) OVER (ORDER BY ename, deptno) sum_ename_deptno,
SUM(sal) OVER (ORDER BY deptno, ename) sum_deptno_ename
FROM scott.emp
ORDER BY ename, deptno
```

ENAME	DEPTNO	SUM_ENAME_DEPTNO	SUM_DEPTNO_ENAME
ADAMS	20	1100	9850
ALLEN	30	2700	21225
BLAKE	30	5550	24075
CLARK	10	8000	2450
FORD	20	11000	12850
JAMES	30	11950	25025

JONES	20	14925	15825
KING	10	19925	7450
MARTIN	30	21175	26275
MILLER	10	22475	8750
SCOTT	20	25475	18825
SMITH	20	26275	19625
TURNER	30	27775	27775
WARD	30	29025	29025

Оба столбца **SUM**(sal) одинаково корректны; один из них содержит **SUM**(sal) при упорядочении по столбцу deptno, а потом — по ename, а другой — при упорядочении по столбцу ename, а потом — по deptno. Поскольку результирующее множество упорядочено по (ename, deptno), значения **SUM**(sal), вычислявшиеся именно в этом порядке, кажутся более корректными, но общая сумма совпадает: 29025.

Конструкция **ORDER BY** в аналитических функциях имеет следующий синтаксис:

ORDER BY выражение [**ASC** | **DESC**] [**NULLS FIRST** | **NULLS LAST**]

Она совпадает с конструкцией **ORDER BY** для запроса, но будет упорядочивать строки только в пределах фрагментов и может не совпадать с конструкцией **ORDER BY** для запроса в целом (или любого другого фрагмента). Конструкции **NULLS FIRST** и **NULLS LAST** впервые появились в версии Oracle 8.1.6. Они позволяют указать, где при упорядочении должны быть значения **NULL** — в начале или в конце. В случае сортировки по убыванию (**DESC**), особенно в аналитических функциях, эта новая возможность принципиально важна.

Т. Кайт - Аналитические функции (4)

Конструкция окна

Синтаксис этой конструкции на первый взгляд кажется сложным из-за используемых ключевых слов. Конструкция вида **RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW**, задающая стандартное окно при использовании конструкции **ORDER BY**, не похожа на те, что постоянно используются разработчиками. Синтаксис конструкции окна достаточно сложен для описания. Вместо попыток перерисовать синтаксические схемы, представленные в руководстве *Oracle8i SQL Reference Manual*, я перечислю все варианты конструкции окна и опишу, какой набор данных в пределах группы задает соответствующий вариант. Сначала, однако, давайте разберемся, что вообще позволяет сделать конструкция окна.

Конструкция окна позволяет задать перемещающееся или жестко привязанное окно (набор) данных в пределах группы, с которым будет работать аналитическая функция.

Например, конструкция диапазона **RANGE UNBOUNDED PRECEDING** означает: "применять аналитическую функцию к каждой строке данной группы, с первой по текущую".

Стандартным является жестко привязанное окно, начинающееся с первой строки группы и продолжающееся до текущей. Если используется следующая аналитическая функция:

```
SUM(sal) OVER
(PARTITION BY deptno
ORDER BY ename
ROWS 2 PRECEDING) department_total2,
```

то будет создано перемещающееся окно в группе, и сумма зарплат будет вычисляться по столбцу sal текущей и двух предыдущих строк в этой группе. Если необходимо создать отчет, показывающий сумму зарплат текущего и двух предыдущих сотрудников отдела, соответствующий сценарий может выглядеть так:

```
SELECT deptno, ename, sal,
SUM(sal) OVER (PARTITION BY deptno ORDER BY ename ROWS 2 PRECEDING) sliding_total
FROM scott.emp
ORDER BY deptno, ename
```

DEPTNO	ENAME	SAL	SLIDING_TOTAL
10	CLARK	2450,00	2450
10	KING	5000,00	7450
10	MILLER	1300,00	8750
20	ADAMS	1100,00	1100
20	FORD	3000,00	4100
20	JONES	2975,00	7075
20	SCOTT	3000,00	8975

20	SMITH	800,00	6775
30	ALLEN	1600,00	1600
30	BLAKE	2850,00	4450
30	JAMES	950,00	5400
30	MARTIN	1250,00	5050
30	TURNER	1500,00	3700
30	WARD	1250,00	4000

Нас интересует эта часть запроса:

```
SUM(sal) OVER (PARTITION BY deptno ORDER BY ename ROWS 2 PRECEDING) sliding_total
```

Конструкция, определяющая фрагментацию, приводит к вычислению **SUM**(sal) по отделам, независимо от других групп(значение **SUM**(sal) сбрасывается при изменении номера отдела). Конструкция **ORDER BY** ename приводит к сортировке данных в каждом отделе по столбцу ename; это позволяет с помощью конструкции окна, **ROWS** 2 **PRECEDING**, при суммировании зарплат обращаться к двум предыдущим строкам всоответствии с заданным порядком сортировки. Например, значение в столбце sliding_total для сотрудника SMITH — 6775, что равно сумме значений 800, 3000 и 2975. Это сумма зарплат в строке для SMITH и двух предыдущих строках окна.

Можно создавать окна по двум критериям: по диапазону (**RANGE**) значений данных или по смещению (**ROWS**) относительно текущей строки. Конструкция **RANGE** уже встречалась ранее, **RANGE UNBOUNDED PRECEDING** например. Она требует брать все строки вплоть до текущей, в соответствии с порядком, задаваемым конструкцией **ORDER BY**. Следует помнить, что для использования окон необходимо задавать конструкцию **ORDER BY**. Сейчас мы рассмотрим задание окон с помощью конструкций **ROWS** и **RANGE**, а затем другие способы задания окон.

Т. Кайт - Аналитические функции (5)

Окна диапазона

Окна диапазона объединяют строки в соответствии с заданным порядком. Если в запросе сказано, например, "**RANGE** 5 **PRECEDING**", то будет сгенерировано перемещающееся окно, включающее предыдущие строки группы, отстоящие от текущей строки не более чем на 5 строк. Диапазон можно задавать в виде числового выражения или выражения, значением которого является дата. Применять конструкцию **RANGE** с другими типами данных нельзя.

Если имеется таблица emp со столбцом hiredate типа даты и задана аналитическая функция **COUNT(*) OVER (ORDER BY** hiredate **ASC RANGE** 100 **PRECEDING)** она найдет все предыдущие строки фрагмента, значение которых в столбце hiredate лежит в пределах 100 дней от значения hiredate текущей строки. В этом случае, поскольку данные сортируются по возрастанию **(ASC)**, значения в окне будут включать все строки текущей группы, у которых значение в столбце HIREDATE меньше значения hiredate текущей строки, но не более чем на 100 дней. Если использовать функцию **COUNT(*) OVER (ORDER BY** hiredate **DESC RANGE** 100 **PRECEDING)** и сортировать фрагмент по убыванию **(DESC)**, базовая логика работы останется той же, но, поскольку группа отсортирована иначе, в окно попадет другой набор строк. В рассматриваемом случае функция найдет все строки, предшествующие текущей, где значение в поле hiredate больше значения hiredate в текущей строке, но не более чем на 100 дней. Пример поможет это прояснить. Я буду использовать запрос с аналитической функцией **FIRST_VALUE**. Эта функция возвращает вычисленное значение для первой строки окна. Так мы легко сможем понять, где начинается окно:

```
SELECT ename, sal, hiredate, hiredate-100 windowtop,
FIRST_VALUE(ename) OVER (ORDER BY hiredate ASC RANGE 100 PRECEDING) ename_prec,
FIRST_VALUE(hiredate) OVER (ORDER by hiredate ASC RANGE 100 PRECEDING) hiredate_prec
FROM scott.emp
ORDER BY hiredate ASC
```

ENAME	SAL	HIREDATE	WINDOWTOP	ENAME_PREC	HIREDATE_PREC
SMITH	800,00	17.12.1980	08.09.1980	SMITH	17.12.1980
ALLEN	1600,00	20.02.1981	12.11.1980	SMITH	17.12.1980
WARD	1250,00	22.02.1981	14.11.1980	SMITH	17.12.1980
JONES	2975,00	02.04.1981	23.12.1980	ALLEN	20.02.1981
BLAKE	2850,00	01.05.1981	21.01.1981	ALLEN	20.02.1981
CLARK	*2450,00*	*09.06.1981*	*01.03.1981*	*JONES*	*02.04.1981*

TURNER	1500,00	08.09.1981	31.05.1981	CLARK	09.06.1981
MARTIN	1250,00	28.09.1981	20.06.1981	TURNER	08.09.1981
KING	5000,00	17.11.1981	09.08.1981	TURNER	08.09.1981
FORD	3000,00	03.12.1981	25.08.1981	TURNER	08.09.1981
JAMES	950,00	03.12.1981	25.08.1981	TURNER	08.09.1981
MILLER	1300,00	23.01.1982	15.10.1981	KING	17.11.1981
SCOTT	3000,00	09.12.1982	31.08.1982	SCOTT	09.12.1982
ADAMS	1100,00	12.01.1983	04.10.1982	SCOTT	09.12.1982

Мы упорядочили один фрагмент по критерию hiredate **ASC**. При этом использовалась аналитическая функция **FIRST_VALUE** для поиска первого значения ename и первого значения hiredate в соответствующем окне. Посмотрев на строку данных для сотрудника CLARK, можно обнаружить, что для него значение в столбце hiredate — 09.06.1981, а дата за 100 дней до этой соответствует 01.03.1981. Для удобства эта дата помещена в столбец windowtop. Аналитическая функция затем вычисляется для всех строк отсортированного фрагмента, предшествующих строке для сотрудника CLARK и имеющих значение в столбце hiredate в диапазоне с 01.03.1981 по 09.06.1981. Первое значение ename для этого окна — JONES.

Это имя и выдает аналитическая функция в столбце ename_prec.

Упорядочив данные по критерию hiredate **DESC**, мы получим:

```
SELECT ename, sal, hiredate, hiredate+100 windowtop,
FIRST_VALUE(ename) OVER (ORDER BY hiredate DESC RANGE 100 PRECEDING) ename_prec,
FIRST_VALUE(hiredate) OVER (ORDER BY hiredate DESC RANGE 100 PRECEDING) hiredate_prec
FROM scott.emp
ORDER BY hiredate DESC
```

ENAME	SAL	HIREDATE	WINDOWTOP	ENAME_PREC	HIREDATE_PREC
ADAMS	1100,00	12.01.1983	22.04.1983	ADAMS	12.01.1983
SCOTT	3000,00	09.12.1982	19.03.1983	ADAMS	12.01.1983
MILLER	1300,00	23.01.1982	03.05.1982	MILLER	23.01.1982
FORD	3000,00	03.12.1981	13.03.1982	MILLER	23.01.1982
JAMES	950,00	03.12.1981	13.03.1982	MILLER	23.01.1982
KING	5000,00	17.11.1981	25.02.1982	MILLER	23.01.1982
MARTIN	1250,00	28.09.1981	06.01.1982	FORD	03.12.1981
TURNER	*1500,00*	*08.09.1981*	*17.12.1981*	*FORD*	*03.12.1981*
CLARK	*2450,00*	*09.06.1981*	*17.09.1981*	*TURNER*	*08.09.1981*
BLAKE	2850,00	01.05.1981	09.08.1981	CLARK	09.06.1981
JONES	2975,00	02.04.1981	11.07.1981	CLARK	09.06.1981
WARD	1250,00	22.02.1981	02.06.1981	BLAKE	01.05.1981
ALLEN	1600,00	20.02.1981	31.05.1981	BLAKE	01.05.1981
SMITH	800,00	17.12.1980	27.03.1981	WARD	22.02.1981

Если снова обратиться к строке сотрудника CLARK, окажется, что выбрано другое окно, поскольку данные фрагмента отсортированы по-иному. Окно для строки CLARK по условию **RANGE** 100 **PRECEDING** теперь доходит до строки TURNER, поскольку значение hiredate для TURNER — последняя дата среди значений hiredate в строках, предшествующих строке CLARK, отличающаяся не более чем на 100 дней.

Иногда достаточно сложно понять, какие значения будут входить в диапазон. Я считаю использование функции **FIRST_VALUE** удобным методом, помогающим увидеть диапазоны окна и проверить, корректно ли установлены параметры. Теперь, представив диапазоны окон, мы используем их для вычисления чего-то более существенного. Пусть необходимо выбрать зарплату каждого сотрудника и среднюю зарплату всех принятых на работу в течение 100 предыдущих дней, а также среднюю зарплату всех принятых на работу в течение 100 следующих дней. Соответствующий запрос будет выглядеть так:

```
SELECT ename, hiredate, sal,
AVG(sal) OVER (ORDER BY hiredate ASC RANGE 100 PRECEDING) avg_sal_100_days_before,
AVG(sal) OVER (ORDER BY hiredate DESC RANGE 100 PRECEDING) avg_sal_100_days_after
```

```
FROM scott.emp
ORDER BY hiredate DESC
```

ENAME	HIREDATE	SAL	AVG_SAL_100_DAYS_BEFORE	AVG_SAL_100_DAYS_AFTER
ADAMS	12.01.1983	1100,00	2050	1100
SCOTT	09.12.1982	3000,00	3000	2050
MILLER	23.01.1982	1300,00	2562,5	1300
FORD	03.12.1981	3000,00	2340	1750
JAMES	03.12.1981	950,00	2340	1750
KING	17.11.1981	5000,00	2583,333333	2562,5
MARTIN	28.09.1981	1250,00	1375	2550
TURNER	08.09.1981	1500,00	1975	2340
CLARK	*09.06.1981*	*2450,00*	*2758,333333*	*1975*
BLAKE	01.05.1981	2850,00	2168,75	2650
JONES	02.04.1981	2975,00	1941,666667	2758,333333
WARD	22.02.1981	1250,00	1216,666667	2358,333333
ALLEN	20.02.1981	1600,00	1200	2168,75
SMITH	17.12.1980	800,00	800	1216,666667

Если теперь снова обратиться к строке для сотрудника CLARK, то, поскольку мы уже понимаем, какое окно в группе будет с ней связано, легко убедиться, что средняя зарплата (2758,333333) равна (2450+2850+2975)/3. Это средняя зарплата для строки CLARK и строк, предшествующих CLARK (это строки для сотрудников JONES и BLAKE) при упорядочении данных по возрастанию. С другой стороны, средняя зарплата 1975 равна (2450+1500)/2. Это средняя зарплата для строки CLARK и строк, предшествующих CLARK при упорядочении данных по убыванию. С помощью этого запроса можно одновременно вычислить среднюю зарплату для сотрудников, принятых на работу за 100 дней до и за 100 дней после сотрудника CLARK.

Окна диапазона можно задавать только по данным типа **NUMBER** или **DATE**, поскольку нельзя добавить или вычесть N единиц из значения типа **VARCHAR2**. Еще одно ограничение для таких окон состоит в том, что в конструкции **ORDER BY** может быть только один столбец — диапазоны по природе своей одномерны. Нельзя задать диапазон в N-мерном пространстве.

Т. Кайт - Аналитические функции (6)

Окна строк

Окна срок задаются в физических единицах, строках. Перепишем вступительный пример из предыдущего раздела, задав окно строк:

```
COUNT (*) OVER (ORDER BY x ROWS 5 PRECEDING)
```

Это окно будет включать до 6 строк: текущую и пять предыдущих (порядок определяется конструкцией **ORDER BY**). Для окон по строкам нет ограничений, присущих окнам по диапазону; данные могут быть любого типа и упорядочивать можно по любому количеству столбцов. Вот пример, сходный с рассмотренным ранее:

```
SELECT ename, sal, hiredate,
FIRST_VALUE(ename) OVER (ORDER BY hiredate ASC ROWS 5 PRECEDING) ename_prec,
FIRST_VALUE(hiredate) OVER (ORDER BY hiredate ASC ROWS 5 PRECEDING) hiredate_prec
FROM scott.emp
ORDER BY hiredate ASC
```

ENAME	SAL	HIREDATE	ENAME_PREC	HIREDATE_PREC
SMITH	800,00	17.12.1980	SMITH	17.12.1980
ALLEN	1600,00	20.02.1981	SMITH	17.12.1980
WARD	1250,00	22.02.1981	SMITH	17.12.1980
JONES	2975,00	02.04.1981	SMITH	17.12.1980
BLAKE	2850,00	01.05.1981	SMITH	17.12.1980
CLARK	*2450,00*	*09.06.1981*	*SMITH*	*17.12.1980*
TURNER	1500,00	08.09.1981	ALLEN	20.02.1981
MARTIN	1250,00	28.09.1981	WARD	22.02.1981
KING	5000,00	17.11.1981	JONES	02.04.1981

JAMES	950,00	03.12.1981	BLAKE	01.05.1981
FORD	3000,00	03.12.1981	CLARK	09.06.1981
MILLER	1300,00	23.01.1982	TURNER	08.09.1981
SCOTT	3000,00	09.12.1982	MARTIN	28.09.1981
ADAMS	1100,00	12.01.1983	KING	17.11.1981

Взглянув на строку для сотрудника CLARK, можно увидеть, что первой в окне **ROWS** 5 **PRECEDING** следует строка для сотрудника SMITH — она просто пятая перед строкой для CLARK в соответствии с заданным порядком. Строка для сотрудника SMITH будет первой в окне и для всех предыдущих строк (для BLAKE, JONES и т.д.). Дело в том, что строка для SMITH — первая в данной группе (она будет первой и для самой себя). При сортировке группы по возрастанию окна изменяются:

```
SELECT ename, sal, hiredate,
FIRST_VALUE(ename) OVER (ORDER BY hiredate DESC ROWS 5 PRECEDING) ename_prec,
FIRST_VALUE(hiredate) OVER (ORDER BY hiredate DESC ROWS 5 PRECEDING) hiredate_prec
FROM scott.emp
ORDER BY hiredate DESC
```

ENAME	SAL	HIREDATE	ENAME_PREC	HIREDATE_PREC
ADAMS	1100,00	12.01.1983	ADAMS	12.01.1983
SCOTT	3000,00	09.12.1982	ADAMS	12.01.1983
MILLER	1300,00	23.01.1982	ADAMS	12.01.1983
FORD	3000,00	03.12.1981	ADAMS	12.01.1983
JAMES	950,00	03.12.1981	ADAMS	12.01.1983
KING	5000,00	17.11.1981	ADAMS	12.01.1983
MARTIN	1250,00	28.09.1981	SCOTT	09.12.1982
TURNER	1500,00	08.09.1981	MILLER	23.01.1982
CLARK	*2450,00*	*09.06.1981*	*FORD*	*03.12.1981*
BLAKE	2850,00	01.05.1981	JAMES	03.12.1981
JONES	2975,00	02.04.1981	KING	17.11.1981
WARD	1250,00	22.02.1981	MARTIN	28.09.1981
ALLEN	1600,00	20.02.1981	TURNER	08.09.1981
SMITH	800,00	17.12.1980	CLARK	09.06.1981

Теперь первое значение для набора из 5 строк, предшествующих в группе строке для сотрудника CLARK, — строка для сотрудника FORD. Теперь можно вычислить среднюю зарплату для указанного сотрудника и пяти принятых на работу до него и после него:

```
SELECT ename, hiredate, sal,
AVG(sal) OVER (ORDER BY hiredate ASC ROWS 5 PRECEDING) avg_5_before,
COUNT(*) OVER (ORDER by hiredate ASC ROWS 5 PRECEDING) obs_before,
AVG(sal) OVER (ORDER BY hiredate DESC ROWS 5 preceding) avg_5_after,
COUNT(*) OVER (ORDER BY hiredate DESC ROWS 5 PRECEDING) obs_after
FROM scott.emp
ORDER BY hiredate
```

ENAME	HIREDATE	SAL	AVG_5_BEFORE	OBS_BEFORE	AVG_5_AFTER	OBS_AFTER
SMITH	17.12.1980	800,00	800	1	1987,5	6
ALLEN	*20.02.1981*	*1600,00*	*1200*	*2*	*2104,166667*	*6*
WARD	22.02.1981	1250,00	1216,666667	3	2045,833333	6
JONES	02.04.1981	2975,00	1656,25	4	2670,833333	6
BLAKE	01.05.1981	2850,00	1895	5	2333,333333	6
CLARK	09.06.1981	2450,00	1987,5	6	2358,333333	6
TURNER	08.09.1981	1500,00	2104,166667	6	2166,666667	6
MARTIN	28.09.1981	1250,00	2045,833333	6	2416,666667	6
KING	17.11.1981	5000,00	2670,833333	6	2391,666667	6

JAMES	03.12.1981	950,00	2333,333333	6	1870	5
FORD	03.12.1981	3000,00	2358,333333	6	2100	4
MILLER	23.01.1982	1300,00	2166,666667	6	1800	3
SCOTT	09.12.1982	3000,00	2416,666667	6	2050	2
ADAMS	12.01.1983	1100,00	2391,666667	6	1100	1

Обратите внимание, что я выбирал также значение **COUNT(*)**. Это позволяет понять, по какому количеству строк было вычислено среднее значение. Можно явно увидеть, что для вычисления средней зарплаты сотрудников, принятых до сотрудника ALLEN, использовалось только 2 записи, а для вычисления средней зарплаты сотрудников, нанятых после него, — 6. В том месте группы, где находится строка для сотрудника ALLEN, есть только 1 предыдущая запись, а при вычислении аналитической функции используются все имеющиеся строки.

Т. Кайт - Аналитические функции (7)

Задание окон

Теперь, понимая различие между окнами диапазонов и окнами строк, можно изучать способы задания окон.

В простейшем случае, окно задается с помощью одной из трех следующих взаимоисключающих конструкций.

• **UNBOUNDED PRECEDING**. Окно начинается с первой строки текущей группы и заканчивается текущей обрабатываемой строкой.

```
SELECT ename, hiredate, sal,
AVG(sal) OVER (ORDER by hiredate ASC ROWS 5 PRECEDING) avg_5_before,
COUNT(*) OVER (ORDER by hiredate ASC ROWS 5 PRECEDING) obs_before,
AVG(sal) OVER (ORDER by hiredate DESC ROWS 5 PRECEDING) avg_5_after,
COUNT(*) OVER (ORDER by hiredate DESC ROWS 5 PRECEDING) obs_after
FROM scott.emp
ORDER BY hiredate
```

ENAME	HIREDATE	SAL	AVG_5_BEFORE	OBS_BEFORE	AVG_5_AFTER	OBS_AFTER
SMITH	17.12.1980	800,00	800	1	1987,5	6
ALLEN	20.02.1981	1600,00	1200	2	2104,166667	6
WARD	22.02.1981	1250,00	1216,666667	3	2045,833333	6
JONES	02.04.1981	2975,00	1656,25	4	2670,833333	6
BLAKE	01.05.1981	2850,00	1895	5	2333,333333	6
CLARK	09.06.1981	2450,00	1987,5	6	2358,333333	6
TURNER	08.09.1981	1500,00	2104,166667	6	2166,666667	6
MARTIN	28.09.1981	1250,00	2045,833333	6	2416,666667	6
KING	17.11.1981	5000,00	2670,833333	6	2391,666667	6
JAMES	03.12.1981	950,00	2333,333333	6	1870	5
FORD	03.12.1981	3000,00	2358,333333	6	2100	4
MILLER	23.01.1982	1300,00	2166,666667	6	1800	3
SCOTT	09.12.1982	3000,00	2416,666667	6	2050	2
ADAMS	12.01.1983	1100,00	2391,666667	6	1100	1

• **CURRENT ROW**. Окно начинается (и заканчивается) текущей строкой.

• **Числовое_выражение PRECEDING**. Окно начинается со строки за **числовое_выражение** строк до текущей, если оно задается по строкам, или со строки, меньшей по значению столбца, упомянутого в конструкции **ORDER BY**, не более чем на **числовое выражение**, если оно задается по диапазону.

Окно **CURRENT ROW** в простейшем виде, вероятно, никогда не используется, поскольку ограничивает применение аналитической функции одной текущей строкой, а для этого аналитические функции не нужны. В более сложном случае для окна задается также конструкция **BETWEEN**. В ней **CURRENT ROW** можно указывать в качестве начальной или конечной строки окна. Начальную и конечную строку окна в конструкции **BETWEEN** можно задавать с использованием любой из перечисленных выше конструкций и еще одной, дополнительной:

• **Числовое_выражение FOLLOWING**. Окно заканчивается (или начинается) со строки, через **числовое_выражение** строк после текущей, если оно задается по строкам, или со строки, большей по значению столбца, упомянутого в конструкции **ORDER BY**, не более чем на **числовое_выражение**, если оно задается по диапазону.

Рассмотрим ряд примеров такого задания окон:

```
SELECT deptno, ename, hiredate,
COUNT(*) OVER (PARTITION BY deptno ORDER BY hiredate NULLS FIRST RANGE 100 PRECEDING)
cnt_range,
COUNT(*) OVER (PARTITION BY deptno ORDER BY hiredate NULLS FIRST ROWS 2 PRECEDING)
cnt_rows
FROM scott.emp
WHERE deptno IN (10, 20)
ORDER BY deptno, hiredate
```

DEPTNO	ENAME	HIREDATE	CNT_RANGE	CNT_ROWS
10	CLARK	09.06.1981	1	1
10	KING	17.11.1981	1	2
10	MILLER	23.01.1982	2	3
20	SMITH	17.12.1980	1	1
20	JONES	02.04.1981	1	2
20	FORD	03.12.1981	1	3
20	SCOTT	09.12.1982	1	3
20	ADAMS	12.01.1983	2	3

Как видите, окно **RANGE** 100 **PRECEDING** содержит только строки текущего фрагмента, предшествующие текущей строке, и те, значение которых hiredate находится в диапазоне hiredate - 100 и hiredate относительно текущей. В данном случае таких строк всегда 1 или 2, т.е. интервал между приемом людей на работу обычно превышает 100 дней (за исключением двух случаев). Окно **ROWS** 2 **PRECEDING**, однако, содержит от 1 до 3 строк (это определяется тем, как далеко текущая строка находится от начала группы). Для первой строки группы имеем значение 1 (предыдущих строк нет).

Для следующей строки в группе таких строк 2. Наконец, для третьей и далее строк значение **COUNT(*)** остается постоянным, поскольку мы считаем только текущую строку и две предыдущие.

Теперь рассмотрим использование конструкции **BETWEEN**. Все заданные до сих пор окна заканчивались текущей строкой и возвращались по результирующему множеству в поисках дополнительной информации. Можно задать окно так, что обрабатываемая строка не будет последней, а окажется где-то в середине окна. Например:

```
SELECT ename, hiredate,
FIRST_VALUE(ename) OVER (ORDER BY hiredate ASC RANGE BETWEEN 100 PRECEDING AND 100
FOLLOWING),
LAST_VALUE(ename) OVER (ORDER BY hiredate ASC RANGE BETWEEN 100 PRECEDING AND 100
FOLLOWING)
FROM scott.emp
ORDER BY hiredate ASC
```

ENAME	HIREDATE	FIRST_VALUE(ENAME)OVER(ORDERBY	LAST_VALUE(ENAME)OVER(ORDERBYH
SMITH	17.12.1980	SMITH	WARD
ALLEN	20.02.1981	SMITH	BLAKE
WARD	22.02.1981	SMITH	BLAKE
JONES	02.04.1981	ALLEN	CLARK
BLAKE	01.05.1981	ALLEN	CLARK
CLARK	*09.06.1981*	*JONES*	*TURNER*
TURNER	08.09.1981	CLARK	JAMES
MARTIN	28.09.1981	TURNER	JAMES
KING	17.11.1981	TURNER	MILLER
FORD	03.12.1981	TURNER	MILLER
JAMES	03.12.1981	TURNER	MILLER
MILLER	23.01.1982	KING	MILLER
SCOTT	09.12.1982	SCOTT	ADAMS
ADAMS	12.01.1983	SCOTT	ADAMS

Обратившись снова к строке для сотрудника CLARK, можно убедиться, что окно начинается со строки для JONES и продолжается до строки для сотрудника TURNER. Теперь в окно входят строки для тех, кто принят на работу за 100 дней до и (а не или, как прежде) после текущего сотрудника.

Итак, теперь мы хорошо знаем синтаксис четырех компонентов вызова аналитической функции. Это:

- имя функции;
- конструкция фрагментации, используемая для разбиения результирующего множества на независимые группы;
- **конструкция ORDER BY**, сортирующая данные в группе **для оконных** функций;
- конструкция окна, задающая набор строк, к которым применяется аналитическая функция.

ИМЯ_ФУНКЦИИ(<аргумент>, <аргумент>, . . .)
OVER (конструкция фрагментации> <конструкция упорядочений <конструкция окна>)

Рассмотрим кратко предлагаемые функции.

Т. Кайт - Аналитические функции (8)

Функции

Сервер предлагает более 26 аналитических функций. Имена некоторых из них совпадают с именами функций агрегирования, например **AVG** и **SUM**. Другие, с новыми именами, обеспечивают новые возможности. В этом разделе будут перечислены имеющиеся функции и кратко описано их назначение.

Аналитическая функция	***Назначение***
AVG([**DISTINCT** | **ALL**] выражение)	Используется для вычисления среднего значения выражения в пределах группы и окна. Для поиска среднего после удаления дублирующихся значений можно указывать ключевое слово **DISTINCT**.
CORR(выражение, выражение)	Выдает коэффициент корреляции для пары выражений, возвращающих числовые значения. Это сокращение для выражения: **COVAR_POP**<выражение1, выражение2) / **STDDEV_POP**(выражение!.)* **STDDEV_POP**(выражение2)). В статистическом смысле, корреляция — это степень связи между переменными. Связь между переменными означает, что значение одной переменной можно в определенной степени предсказать по значению другой. Коэффициент корреляции представляет степень корреляции в виде числа в диапазоне от -1 (высокая обратная корреляция) до 1 (высокая корреляция). Значение 0 соответствует отсутствию корреляции.
COUNT([**DISTINCT**][*] [выражение])	Эта функция считает строки в группах. Если указать * или любую константу, кроме **NULL**, функция **COUNT** будет считать все строки. Если указать выражение, функция **COUNT** будет считать строки, для которых выражение имеет значение не NULL. Можно задавать модификатор **DISTINCT**, чтобы считать строки в группах после удаления дублирующихся строк.
COVAR_POP(выражение, выражение)	Возвращает *ковариацию генеральной совокупности* (population covariance) пары выражений с числовыми значениями.
COVAR_SAMP(выражение, выражение)	Возвращает *выборочную ковариацию* (sample covariance) пары выражений с числовыми значениями.
CUME_DIST	Вычисляет относительную позицию строки в группе. Функция **CUME_DIST** всегда возвращает число большее 0 и меньше или равное 1. Это число представляет "позицию" строки в группе из N арок. В группе из трех строк, например, возвращаются следующие значения кумулятивного распределения: 1/3, 2/3 и 3/3.
DENSE_RANK	Эта функция вычисляет относительный ранг каждой возвращаемой запросом строки по отношению к другим строкам, основываясь на значениях выражений в конструкции **ORDER BY**. Данные в группе сортируются в

	соответствии с конструкцией **ORDER BY**, а затем каждой строке поочередно присваивается числовой ранг, начиная с 1. Ранг увеличивается при каждом изменении значений выражений, входящих в конструкцию **ORDER BY**. Строки с одинаковыми значениями получают один и тот же ранг (при этом сравнении значения **NULL** считаются одинаковыми). Возвращаемый этой функцией "плотный" ранг дает ранговые значения без промежутков. Сравните с представленной далее функцией **RANK**.
FIRST_VALUE	Возвращает первое значение в группе.
LAG(выражение, <смещение>,<стандартное значение>)	Функция **LAG** дает доступ к другим строкам результирующего множества, избавляя от необходимости выполнять самосоединения. Она позволяет работать с курсором как с массивом. Можно ссылаться на строки, предшествующие текущей строке в группе. О том, как обращаться к следующим строкам в группе, см. в описании функции **LEAD**. Смещение - это положительное целое число со стандартным значением 1 (предыдущая строка). Стандартное значение возвращается, если индекс выходит за пределы окна (для первой строки группы будет возвращено стандартное значение).
LAST_VALUE	Возвращает последнее значение в группе.
LEAD(выражение, <смещение>,<стандартное значение>)	Функция **LEAD** противоположна функции **LAG**. Если функция **LAG** дает доступ к предшествующим строкам группы, то функция **LEAD** позволяет обращаться к строкам, следующим за текущей. Смещение — это положительное целое число со стандартным значением 1 (следующая строка). Стандартное значение возвращается, если индекс выходит за пределы окна (для последней строки группы будет возвращено стандартное значение).
MAX(выражение)	Находит максимальное значение выражения в пределах окна в группе.
MIN(выражение)	Находит минимальное значение выражения в пределах окна в группе.
NTILE(выражение)	Делит группу на фрагменты по значению выражения. Например, если выражение = 4, то каждой строке в группе присваивается число от 1 до 4 в соответствии с фрагментом, в которую она попадает. Если в группе 20 строк, первые 5 получат значение 1, следующие 5 — значение 2 и т.д. Если количество строк в группе не делится на значение выражения без остатка, строки распределяются так, что ни в одном фрагменте количество строк не превосходит минимальное количество в других фрагментах более чем на 1, причем дополнительные строки будут в группах с меньшими номера фрагмента. Например, если снова выражение = 4, а количество строк = 21, в первом фрагменте будет 6 строк, во втором и последующих - 5.
PERCENT RANK	Аналогична функции **CUME_DIST** (кумулятивное распределение). Вычисляет ранг строки в группе минус 1, деленный на количество обрабатываемых строк минус 1. Эта функция всегда возвращает значения в диапазоне от 0 до 1 включительно.
RANK	Эта функция вычисляет относительный ранг каждой строки, возвращаемой запросом, на основе значений выражений, входящих в конструкцию **ORDER** BY. Данные в группе сортируются в соответствии с конструкцией **ORDER BY**, а затем каждой строке поочередно присваивается числовой ранг, начиная с 1. Строки с

	одинаковыми значениями выражений, входящих в конструкцию **ORDER BY**, получают одинаковый ранг, но если две строки получат одинаковый ранг, следующее значение ранга пропускается. Если две строки получили ранг 1, строки с рангом 2 не будет; следующая строка в группе получит ранг 3. В этом отличие от функции **DENSE_RANK**, которая не пропускает значений.
RATIO_TO_REPORT(выражение)	Эта функция вычисляет значение выражение **/** (**SUM**(выражение)) по строкам группы. Это дает процент, который составляет значение текущей строки по отношению к **SUM**(выражение).
REGR_xxxxxxx(выражение, выражение)	Эти функции линейной регрессии применяют стандартную линейную регрессию по методу наименьших квадратов к паре выражений. Предлагается 9 различных функций регрессии.
ROW_NUMBER	Возвращает смещение строки по отношению к началу упорядоченной группы. Может использоваться для последовательной нумерации строк, упорядоченных по определенным критериям.
STDDEV(выражение)	Вычисляет *стандартное (среднеквадратичное) отклонение* (standard deviation) текущей строки по отношению к группе.
STDDEV_POP(выражение)	Эта функция вычисляет *стандартное отклонение генеральной совокупности* (population standard deviation) и возвращает квадратный корень из *дисперсии генеральной совокупности* (population variance). Она возвращает значение, совпадающее с квадратным корнем из результата функции **VAR_POP**.
STDDEV_SAMP(выражение)	Эта функция вычисляет *накопленное стандартное отклонение выборки* (cumulative sample standard deviation) и возвращает квадратный корень *выборочной дисперсии* (sample variance). Она возвращает значение, совпадающее с квадратным корнем из результата функции **VAR_SAMP**.
SUM(выражение)	Вычисляет общую сумму значений выражения для группы.
VAR_POP(выражение)	Эта функция возвращает дисперсию генеральной совокупности для набора числовых значений (значения **NULL** игнорируются). Функция **VAR_POP** вычисляет значение: (**SUM**(выражение ***** выражение) – **SUM**(выражение) ***** **SUM**(выражение) **/** **COUNT**(выражение)) **/** **COUNT**(выражение)
VAR_SAMP(выражение)	Эта функция возвращает выборочную дисперсию для набора числовых значений (значения **NULL** игнорируются). Она вычисляет значение: (**SUM**(выражение ***** выражение) – **SUM**(выражение) ***** **SUM**(выражение) **/** **COUNT**(выражение)) **/** (**COUNT**(выражение) **-**
VARIANCE(выражение)	Возвращает дисперсию для выражения. Сервер Oracle вычисляет дисперсию как: • 0, если количество строк в группе **=** 1; • **VAR_SAMP**, если количество строк в группе **>** 1.

Т. Кайт - Аналитические функции (8)

Примеры

Теперь можно переходить к самой интересной части — возможностям, предоставляемым аналитическими функциями. Приводимые примеры не демонстрируют все возможности, а лишь дают начальное представление.

Запрос первых N

Мне часто задают вопрос: "Как получить первых N записей набора полей?". До появления аналитических функций ответить на такой вопрос было очень трудно.

С запросами первых N записей, однако, бывают трудности связанные в основном с формулировкой задачи. Это надо учитывать при проектировании отчетов. Рассмотрим следующее, вполне разумное на первый взгляд требование: получить для каждого отдела трех наиболее высокооплачиваемых специалистов по продажам.

Однако эта задача неоднозначна из-за возможного повторения значений: в отделе может быть четыре человека с одинаково огромной зарплатой, и что тогда делать?

Я могу предложить как минимум три одинаково разумных интерпретации этого требования, причем каждой интерпретации может соответствовать и не три записи! Требование можно интерпретировать так.

- Выдать список специалистов по продажам, имеющих одну из трех максимальных зарплат. Другими словами, найти все различные значения зарплаты, отсортировать, выбрать три наибольших, и вернуть всех сотрудников, зарплата которых совпадает с одним из этих трех значений.
- Выдать до трех человек с максимальными зарплатами. Если четыре человека имеют одинаковую максимальную зарплату, в ответ не должно выдаваться ни одной строки. Если два сотрудника имеют максимальную зарплату и два — следующую по значению, ответ будет предполагать две строки (два сотрудника с максимальной зарплатой).
- Отсортировать специалистов по продажам по убыванию зарплат. Вернуть первые три строки. Если в отделе менее трех специалистов по продажам, в результате будет менее трех записей.

После дополнительных вопросов и уточнений оказывается, что некоторым необходима первая интерпретация; другим — вторая или третья. Давайте рассмотрим, как с помощью аналитических функций сформулировать все три запроса, и как это делалось без них.

Для этих примеров используем таблицу scott.emp. Сначала реализуем запрос "Выдать список специалистов по продажам в каждом отделе, имеющих одну из трех максимальных зарплат":

```
SELECT * FROM
(
  SELECT deptno, ename, sal,
  DENSE_RANK() OVER (PARTITION BY deptno ORDER BY sal DESC) dr
  FROM scott.emp
)
WHERE dr <= 3
ORDER BY deptno, sal DESC
```

DEPTNO	ENAME	SAL	DR
10	KING	5000,00	1
10	CLARK	2450,00	2
10	MILLER	1300,00	3
20	FORD	3000,00	1
20	SCOTT	3000,00	1
20	JONES	2975,00	2
20	ADAMS	1100,00	3
30	BLAKE	2850,00	1
30	ALLEN	1600,00	2
30	TURNER	1500,00	3

Здесь для получения трех максимальных зарплат была использована функция **DENSE_RANK()**. Мы присвоили записям непрерывные ранговые значения по столбцу sal и отсортировали результат по убыванию. Если обратиться к описанию функций, оказывается, что при непрерывном ранжировании значения ранга не пропускаются и две строки с одинаковыми значениями получают одинаковый ранг. Поэтому после построения результирующего множества в виде подставляемого представления, можно просто выбирать все строки с "плотным" рангом не более трех. В результате для каждого отдела будут получены все сотрудники с одной из трех максимальных зарплат в отделе. Для сравнения выберем функцию RANK и сравним, что происходит при обнаружении дублирующихся значений:

```
SELECT deptno, ename, sal,
DENSE_RANK() OVER (PARTITION BY deptno ORDER BY sal DESC) dr,
RANK() OVER (PARTITION BY deptno ORDER BY sal DESC) r
FROM scott.emp
ORDER BY deptno, sal DESC
```

DEPTNO	ENAME	SAL	DR	R
10	KING	5000,00	1	1
10	CLARK	2450,00	2	2
10	MILLER	1300,00	3	3

20	SCOTT	3000,00	1	1
20	FORD	3000,00	1	1
20	JONES	2975,00	2	3
20	ADAMS	1100,00	3	4
20	SMITH	800,00	4	5
30	BLAKE	2850,00	1	1
30	ALLEN	1600,00	2	2
30	TURNER	1500,00	3	3
30	MARTIN	1250,00	4	4
30	WARD	1250,00	4	4
30	JAMES	950,00	5	6

Если бы использовалась функция **RANK**, сотрудник ADAMS (получивший ранг 4) не вошел бы в результирующее множество, но он — один из сотрудников отдела 20, получивших одну из трех максимальных зарплат, так что в результат он попадать должен.

В данном случае использование функции **RANK** вместо **DENSE_RANK** привело бы к неправильному ответу на поставленный вопрос.

Наконец, пришлось использовать подставляемое представление и задать псевдоним dr для результатов аналитической функции **DENSE_RANK()**. Дело в том, что нельзя использовать аналитические функции в конструкциях **WHERE** или **HAVING** непосредственно, так что пришлось выбрать результат в представление, а затем отфильтровать, оставив только необходимые строки. Использование подставляемого представления с условием — типичная конструкция для многих примеров в этой главе.

Теперь вернемся к запросу "Выдать не более трех человек с максимальными зарплатами по каждому отделу":

```
SELECT * FROM
(
  SELECT deptno, ename, sal,
  COUNT(*) OVER (PARTITION BY deptno ORDER BY sal DESC RANGE UNBOUNDED PRECEDING) cnt
   FROM scott.emp
)
WHERE cnt <= 3
ORDER BY deptno, sal DESC
```

DEPTNO	ENAME	SAL	CNT
10	KING	5000,00	1
10	CLARK	2450,00	2
10	MILLER	1300,00	3
20	SCOTT	3000,00	2
20	FORD	3000,00	2
20	JONES	2975,00	3
30	BLAKE	2850,00	1
30	ALLEN	1600,00	2
30	TURNER	1500,00	3

Этот запрос немного нетривиален. Мы подсчитываем все записи в окне, предшествующие текущей, при сортировке по зарплате. Диапазон **RANGE UNBOUNDED PRECEDING** задает окно, включающее все записи, зарплата в которых больше или равна зарплате в текущей записи, поскольку сортировка выполнена по убыванию **(DESC)**. Подсчитывая всех сотрудников с такой же или более высокой зарплатой, можно выбирать только строки, в которых значение этого количества (cnt), меньше или равно 3. Обратите внимание, что в отделе 20 для сотрудников SCOTT и FORD возвращается значение 2. Оба они получили наибольшую зарплату в отделе, так что попадают в окно друг для друга. Интересно отметить небольшое отличие, которое дает следующий запрос:

```
SELECT * FROM
(
  SELECT deptno, ename, sal,
  COUNT(*) OVER (PARTITION BY deptno ORDER BY sal DESC, ename RANGE UNBOUNDED PRECEDING)
cnt
```

```
   FROM scott.emp
)
WHERE cnt <= 3
ORDER BY deptno, sal DESC
```

DEPTNO	ENAME	SAL	CNT
10	KING	5000,00	1
10	CLARK	2450,00	2
10	MILLER	1300,00	3
20	FORD	3000,00	1
20	SCOTT	3000,00	2
20	JONES	2975,00	3
30	BLAKE	2850,00	1
30	ALLEN	1600,00	2
30	TURNER	1500,00	3

Обратите внимание, как добавление столбца в конструкцию **ORDER BY** повлияло на окно. Ранее сотрудники FORD и SCOTT оба имели в столбце cnt значение 2. Причина в том, что окно строилось исключительно по столбцу зарплаты. Более избирательное окно дает другие результаты функции **COUNT**. Я привел этот пример, чтобы подчеркнуть, что функция окна зависит от обеих конструкций, **ORDER BY** и **RANGE**. Если фрагмент сортировался только по зарплате, строка для сотрудника FORD предшествовала строке для SCOTT, когда строка для SCOTT была текущей, а строка для SCOTT, в свою очередь, предшествовала строке для FORD, когда та была текущей. Только при сортировке по столбцам sal и ename можно однозначно упорядочить строки для сотрудников SCOTT и FORD по отношению друг к другу.

Чтобы убедиться, что этот подход, предусматривающий использование функции **COUNT**, позволяет возвращать не более трех записей, давайте изменим данные так, чтобы максимальная зарплата была у большего числа сотрудников отдела:

```
UPDATE scott.emp SET sal = 99 WHERE deptno = 30;

SELECT * FROM
(
  SELECT deptno, ename, sal,
  COUNT(*) OVER (PARTITION BY deptno ORDER BY sal DESC RANGE UNBOUNDED PRECEDING) cnt
  FROM scott.emp
)
WHERE cnt <= 3
ORDER BY deptno, sal DESC
```

DEPTNO	ENAME	SAL	CNT
10	KING	5000,00	1
10	CLARK	2450,00	2
10	MILLER	1300,00	3
20	SCOTT	3000,00	2
20	FORD	3000,00	2
20	JONES	2975,00	3

Теперь строк для отдела 30 в отчете нет, поскольку 6 сотрудников этого отдела имеют одинаковую зарплату. В поле cnt для всех них находится значение 6, которое никак не меньше или равно 3.

Перейдем теперь к последнему запросу: "Отсортировать специалистов по продажам по убыванию зарплат и вернуть первые три строки". Это легко сделать с помощью функции **ROW_NUMBER()**:

```
UPDATE scott.emp SET sal = 99 WHERE deptno = 30;

SELECT * FROM
(
  SELECT deptno, ename, sal,
```

```
  ROW_NUMBER() OVER (PARTITION BY deptno ORDER BY sal DESC) rn
   FROM scott.emp
)
WHERE rn <= 3
```

DEPTNO	ENAME	SAL	RN
10	KING	5000,00	1
10	CLARK	2450,00	2
10	MILLER	1300,00	3
20	FORD	3000,00	1
20	SCOTT	3000,00	2
20	JONES	2975,00	3
30	BLAKE	99,00	1
30	MARTIN	99,00	2
30	ALLEN	99,00	3

При выполнении запроса каждый фрагмент сортируется по убыванию значений зарплат, после чего по мере обработки каждой строке фрагмента присваивается последовательный номер. После этого с помощью конструкции **WHERE** мы получаем только первые три строки каждого фрагмента. В примере с транспонированием результирующего множества мы используем такой же прием для преобразования строк в столбцы. Следует отметить, однако, что для отдела deptno = 30 возвращаются в определенном смысле случайные строки. Если помните, информация в отделе 30 была изменена так, что все 6 сотрудников получили значение 99 в столбце зарплаты. Можно в некоторой степени управлять тем, какие три записи будут возвращаться, с помощью конструкции **ORDER BY**. Например, можно использовать конструкцию **ORDER BY** sal **DESC**, ename для получения упорядоченной по фамилии информации о наиболее высоко оплачиваемых сотрудниках, если несколько из них имеют одинаковую зарплату.

Теперь сделаем **ROLLBACK**, т.к. изменения вносились для определенных примеров.

Интересно отметить, что с помощью функции **ROW_NUMBER** можно получать произвольную секцию данных из группы строк. Это может пригодиться в среде, не поддерживающей информацию о состоянии, когда надо выдавать данные постранично. Например, если необходимо выдавать данные из таблицы emp, отсортированные по столбцу ename, группами по пять строк, можно использовать запрос следующего вида:

```
SELECT ename, hiredate, sal FROM
(
  SELECT ename, hiredate, sal,
  ROW_NUMBER() OVER (ORDER BY ename) rn
   FROM scott.emp
)
WHERE rn BETWEEN 5 AND 10
ORDER BY rn
```

ENAME	HIREDATE	SAL
FORD	03.12.1981	3000,00
JAMES	03.12.1981	950,00
JONES	02.04.1981	2975,00
KING	17.11.1981	5000,00
MARTIN	28.09.1981	1250,00
MILLER	23.01.1982	1300,00

Т. Кайт - Аналитические функции (9)

Запрос с транспонированием

При *запросе с транспонированием* (опорный запрос — pivot query) берутся данные вида:

```
C1 C2 C3
a l b1 xl
a l b1 x2
a l b1 x3
```

и выдаются в следующем виде:

```
C1 C2 C3(1) C3(2) C3(3)
a1 b1 xl    x2    x3
```

Этот запрос преобразует строки в столбцы. Например, можно выдать должности сотрудников отдела в виде столбцов:

DEPTNO	JOB_1	JOB_2	JOB_3
10	CLERK	MANAGER	PRESIDENT
20	ANALYST	ANALYST	CLERK
30	CLERK	MANAGER	SALESMAN

а не виде строк:

DEPTNO	JOB
10	CLERK
10	MANAGER
10	PRESIDENT
20	ANALYST
20	CLERK
20	MANAGER
30	CLERK
30	MANAGER
30	SALESMAN

Я представлю два примера запросов с транспонированием. Первый — разновидность описанного выше запроса трех сотрудников с максимальными зарплатами. Второй пример показывает, как транспонировать любое результирующее множество, и дает шаблон необходимых для этого действий.

Предположим, необходимо выдать фамилии сотрудников отдела с тремя наибольшими зарплатами в виде **столбцов**. Запрос должен возвращать ровно одну строку для каждого отдела, причем в строке должно быть 4 столбца: номер отдела (deptno), фамилия сотрудника с наибольшей зарплатой в отделе, фамилия сотрудника со следующей по величине зарплатой и т.д. С помощью новых аналитических функций это сделать просто (а до их появления — практически невозможно):

```
SELECT deptno,
MAX(DECODE(seq,1,ename,NULL)) highest_paid,
MAX(DECODE(seq,2,ename,NULL)) second_highest,
MAX(DECODE(seq,3,ename,NULL)) third_highest
FROM
(
  SELECT deptno, ename,
  ROW_NUMBER() OVER (PARTITION BY deptno ORDER BY sal DESC NULLS LAST) seq
  FROM scott.emp
)
WHERE seq <= 3
GROUP BY deptno
```

DEPTNO	HIGHEST_PAID	SECOND_HIGHEST	THIRD_HIGHEST
10	KING	CLARK	MILLER
20	FORD	SCOTT	JONES

30	BLAKE	ALLEN	TURNER

Мы создали внутреннее результирующее множество, где сотрудники отделов пронумерованы по убыванию зарплат. Функция **DECODE** во внешнем запросе оставляет только строки со значениями номеров 1, 2 или 3 и присваивает взятые из них фамилии соответствующему столбцу. Конструкция **GROUP BY** позволяет избавиться от лишних строк и получить сжатый результат. Возможно, понять, что я имею в виду, проще, если сначала посмотреть на результирующее множество запроса без конструкций **GROUP BY** и **MAX**:

```
SELECT deptno,
DECODE(seq,1,ename,NULL) highest_paid,
DECODE(seq,2,ename,NULL) second_highest,
DECODE(seq,3,ename,NULL) third_highest
FROM
(
  SELECT deptno, ename,
  ROW_NUMBER() OVER (PARTITION BY deptno ORDER BY sal DESC NULLS LAST) seq
  FROM scott.emp
)
WHERE seq <= 3
```

DEPTNO	HIGHEST_PAID	SECOND_HIGHEST	THIRD_HIGHEST
10	KING		
10		CLARK	
10			MILLER
20	FORD		
20		SCOTT	
20			JONES
30	BLAKE		
30		ALLEN	
30			TURNER

Функция агрегирования **MAX** будет применяться конструкцией группировки **GROUP BY** по столбцу deptno. Значение в столбце highest_paid для отдела только в одной строке будет непустым — в остальных строках этот столбец всегда будет иметь значение **NULL**. Функция **MAX** будет выбирать только строку с непустым значением. По этому сочетание группирования и функции **MAX** позволит, убрав значения **NULL**, "свернуть" результирующее множество и получить желаемый результат.

Если есть таблица T со столбцами C1 и C2 и необходимо получить результат вида:

Cl C2(l) C2(2) .... C2(N),

где столбец C1 должен присутствовать во всех строках (значения выдаются по направ лению к концу страницы), а столбец C2 должен быть транспонирован так, чтобы он представлялся в виде строк (значения C2 выдаются по направлению к концу строки, они становятся столбцами, а не строками), надо создать такой запрос:

```
SELECT cl,
MAX(DECODE(rn,1,c2,NULL)) c2_l,
MAX(DECODE(rn,2,c2,NULL)) c2_2,
MAX(DECODE(rn,N,c2,NULL)) c2_N
FROM
(
  SELECT cl, c2,
  ROW_NUMBER() OVER (PARTITION BY cl ORDER BY <столбцы>) rn
  FROM t
  <условие>
)
GROUP BY cl
```

В представленном выше примере в качестве cC1 использовался столбец deptno, а в качестве c2 — ename. Поскольку упорядочение выполнялось по критерию sal **DESC**, первые три полученные

строки соответствовали трем наиболее высокооплачиваемым сотрудникам соответствующего отдела (напоминаю: если максимальные зарплаты получало четыре человека, одного из них мы теряем).

Второй пример: транспонировать результирующее множество. Рассмотрим более общий случай, когда опорный (отсюда и второе название запроса — опорный) столбец, c1, и транспонируемый столбец, c2, представляют собой наборы столбцов. Решение очень похоже на то, что представлено выше. Предположим, необходимо для каждого отдела и должности выдать фамилии и зарплаты сотрудников. При этом в отчете фамилии и соответствующие зарплаты должны выдаваться в строке, как столбцы. Кроме того, в строке сотрудников надо упорядочивать слева направо по возрастанию зарплат. Для решения этой проблемы необходимо выполнить следующее:

```
SELECT MAX(COUNT(*)) FROM scott.emp GROUP BY deptno, job;
```

MAX(COUNT(*))
4

В результате мы получаем количество столбцов. Теперь можно создавать запрос:

```
SELECT
  deptno, job,
  MAX(DECODE(rn, 1, ename, NULL)) ename_1,
  MAX(DECODE(rn, 1, sal, NULL)) sal_1,
  MAX(DECODE(rn, 2, ename, NULL)) ename_2,
  MAX(DECODE(rn, 2, sal, NULL)) sal_2,
  MAX(DECODE(rn, 3, ename, NULL)) ename_3,
  MAX(DECODE(rn, 3, sal, NULL)) sal_3,
  MAX(DECODE(rn, 4, ename, NULL)) ename_4,
  MAX(DECODE(rn, 4, sal, NULL)) sal_4
FROM
(
  SELECT
    deptno, job, ename, sal,
    ROW_NUMBER() OVER (PARTITION BY deptno, job ORDER BY sal, ename) rn
  FROM scott.emp
)
GROUP BY deptno, job
```

DEPTNO	JOB	ENAME_L	SAL_L	ENAME_2	SAL_2	ENAME_3	SAL_3	ENAME_4	SAL_4
10	CLERK	MILLER	1300						
10	MANAGER	CLARK	2450						
10	PRESIDENT	KING	5000						
20	ANALYST	FORD	3000	SCOTT	3000				
20	CLERK	SMITH	800	ADAMS	1100				
20	MANAGER	JONES	2975						
30	CLERK	JAMES	950						
30	MANAGER	BLAKE	2850						
30	SALESMAN	MARTIN	1250	WARD	1250	TURNER	1500	ALLEN	1600

Т. Кайт - Аналитические функции (10)

Доступ к строкам вокруг текущей строки

Часто необходимо обращаться к данным не только в текущей строке, но и в ближайших предыдущих или последующих. Предположим, необходимо создать отчет, в котором по отделам были бы представлены все сотрудники, причем, для каждого сотрудника выдана дата его приема на работу, за сколько дней до этой даты последний раз принимали сотрудника на работу, и через сколько дней после этого приняли на работу следующего сотрудника. Написание подобного запроса с помощью "чистого" языка SQL чрезвычайно сложная задача. Более того, производительность полученного запроса вызывает сомнения. В прошлом я либо пытался применять прием "select из select", либо писал PL/SQL-функцию, которая по данным из текущей строки находила предыдущую и следующую строки данных. Это работало, но очень много времени уходило

на разработку запроса (приходилось писать больше кода); кроме того, расходовалось большое количество ресурсов при его выполнении.

С помощью аналитических функций это делается быстро и эффективно. Соответствующий запрос будет выглядеть так:

```
SELECT
  deptno, ename, hiredate,
  LAG(hiredate, 1, NULL) OVER (PARTITION BY deptno ORDER BY hiredate, ename ) last_hire,
  hiredate - LAG(hiredate, 1, NULL) OVER (PARTITION BY deptno ORDER BY hiredate, ename)
days_last,
  LEAD(hiredate, 1, NULL) OVER (PARTITION BY deptno ORDER BY hiredate, ename) next_hire,
  LEAD(hiredate, 1, NULL) OVER (PARTITION BY deptno ORDER BY hiredate, ename) - hiredate
days_next
FROM scott.emp
ORDER BY deptno, hiredate
```

DEPTNO	ENAME	HIREDATE	LAST_HIRE	DAYS_LAST	NEXT_HIRE	DAYS_NEXT
10	CLARK	**09.06.1981**			17.11.1981	161
10	KING	17.11.1981	**09.06.1981**	161	**23.01.1982**	67
10	MILLER	**23.01.1982**	17.11.1981	67		
20	SMITH	17.12.1980			02.04.1981	106
20	JONES	02.04.1981	17.12.1980	106	03.12.1981	245
20	FORD	03.12.1981	02.04.1981	245	09.12.1982	371
20	SCOTT	09.12.1982	03.12.1981	371	12.01.1983	34
20	ADAMS	12.01.1983	09.12.1982	34		
30	ALLEN	20.02.1981			22.02.1981	2
30	WARD	22.02.1981	20.02.1981	2	01.05.1981	68
30	BLAKE	01.05.1981	22.02.1981	68	08.09.1981	130
30	TURNER	08.09.1981	01.05.1981	130	28.09.1981	20
30	MARTIN	28.09.1981	08.09.1981	20	03.12.1981	66
30	JAMES	03.12.1981	28.09.1981	66		

Функции **LEAD** и **LAG** можно рассматривать как способы индексации в пределах группы. С помощью этих функций можно обратиться к любой отдельной строке. Обратите внимание: в представленных выше результатах запись для сотрудника KING включает данные (выделены полужирным) из предыдущей строки (LAST_HIRE) и последующей (NEXT_HIRE). Можно получить поля предыдущих или последующих записей в упорядоченном фрагменте.

Рассмотрим функции **LAG** и **LEAD**. Эти функции принимают три аргумента:

LAG(Arg1, Arg2, Arg3)

- **Arg1** — выражение, которое надо вернуть на основе другой строки.
- **Arg2** — смещение требуемой строки в группе относительно текущей. Смещение задается как положительное целое число. В случае функции **LAG** берется соответствующая смещению предыдущая строка, а в случае функции **LEAD** — следующая. Этот аргумент имеет стандартное значение 1.
- **Arg3** — возвращаемое значение в том случае, если смещение, заданное аргументом Arg2, выводит за границу группы. Например, первая строка в каждой группе не имеет предыдущей, так что значение функции **LAG**(..., 1) для этой строки определить нельзя. Можно возвращать стандартное значение **NULL** или указать значение явно. Следует учитывать, что окна для функций **LAG** и **LEAD** не используются — можно задавать конструкции **PARTITION BY** и **ORDER BY**, но не **ROWS** или **RANGE**.

Итак, в нашем примере:

```
hiredate - LAG(hiredate, 1, NULL)
OVER (PARTITION BY deptno
ORDER BY hiredate, ename) days_last,
```

функция **LAG** использовалась для поиска предыдущей строки, поскольку в качестве второго параметра передавалось значение 1 (если предыдущей записи нет, возвращается значение **NULL**). Мы фрагментировали данные по столбцу deptno, так что каждый отдел просматривается независимо от остальных. Полученный фрагмент мы упорядочили по значению столбца hiredate, так что вызов **LAG**(hiredate, 1, **NULL**) возвращает максимальное значение hiredate, меньшее соответствующего значения в текущей строке.

Аналитические функции в конструкции WHERE

Следует учитывать, что аналитические функции применяются по ходу выполнения запроса почти в самом конце (после них обрабатывается только окончательная конструкция **ORDER BY**). Это означает, что аналитические функции нельзя непосредствен но использовать в условиях (т.е. применять в конструкциях **WHERE** и **HAVING**). Если необходимо включать данные в результирующее множество на основе результатов аналитической функции, придется использовать подставляемое представление. Аналитические функции могут использоваться только в списке выбора или в конструкции **ORDER BY** запроса.

В этой главе приводилось много примеров использования подставляемых представлений, в частности в разделе, посвященном выбору первых N строк. Например, чтобы найти группу сотрудников каждого отдела с тремя наибольшими зарплатами, мы выполняли следующий запрос:

```
SELECT * FROM
(
   SELECT deptno, ename, sal,
   DENSE_RANK() OVER (PARTITION BY deptno ORDER BY sal DESC) dr
   FROM scott.emp
)
WHERE dr <= 3
ORDER BY deptno, sal DESC
```

DEPTNO	ENAME	SAL	DR
10	KING	5000,00	1
10	CLARK	2450,00	2
10	MILLER	1300,00	3
20	FORD	3000,00	1
20	SCOTT	3000,00	1
20	JONES	2975,00	2
20	ADAMS	1100,00	3
30	BLAKE	2850,00	1
30	ALLEN	1600,00	2
30	TURNER	1500,00	3

Поскольку функцию **DENSE_RANK** нельзя использовать в конструкции **WHERE** непосредственно, приходится скрывать ее в подставляемом представлении под псевдонимом dr, чтобы в дальнейшем можно было использовать столбец dr в условии для получения необходимых строк. Такой прием часто используется при работе с аналитическими функциями.

Значения NULL и сортировка

Значения **NULL** могут влиять на результат работы аналитических функций, особенно при использовании сортировки по убыванию. По умолчанию значения **NULL** считаются больше любых других значений. Рассмотрим следующий пример:

```
SELECT ename, comm FROM scott.emp ORDER BY comm DESC
```

ENAME	COMM
KING	
BLAKE	
CLARK	
JONES	
MILLER	
SCOTT	
SMITH	
ADAMS	
JAMES	
FORD	
MARTIN	1400,00
WARD	500,00
ALLEN	300,00

TURNER	0,00

Выбрав первые N строк, получим:

```
SELECT ename, comm, dr FROM
(
   SELECT ename, comm, DENSE_RANK() OVER (ORDER BY comm DESC) dr
   FROM scott.emp
)
WHERE dr <= 3
ORDER BY comm
```

ENAME	COMM	DR
WARD	500,00	3
MARTIN	1400,00	2
CLARK		1
JONES		1
JAMES		1
SCOTT		1
MILLER		1
ADAMS		1
SMITH		1
BLAKE		1
KING		1
FORD		1

Хотя формально это верно, но вряд ли соответствует желаемому результату. Значения **NULL** либо вообще не должны учитываться, либо интерпретироваться как "наименьшие" в данном случае. Поэтому надо либо исключить значения **NULL** из рассмотрения,
добавив условие **where comm is not null**:

```
SELECT ename, comm, dr FROM
(
   SELECT ename, comm,
   DENSE_RANK() OVER (ORDER BY comm DESC) dr
   FROM scott.emp
   WHERE comm IS NOT NULL
)
WHERE dr <= 3
ORDER BY comm DESC
```

ENAME	COMM	DR
MARTIN	1400,00	1
WARD	500,00	2
ALLEN	300,00	3

либо использовать **NULLS LAST** в конструкции **ORDER BY**:

```
SELECT ename, comm, dr FROM
(
   SELECT ename, comm,
   DENSE_RANK() OVER (ORDER BY comm DESC NULLS LAST) dr
   FROM scott.emp
   WHERE comm IS NOT NULL
)
WHERE dr <= 3
ORDER BY comm DESC
```

ENAME	COMM	DR
MARTIN	1400,00	1
WARD	500,00	2
ALLEN	300,00	3

Следует помнить, что **NULLS LAST** можно указывать и в обычных конструкциях **ORDER BY**, а не только при вызове аналитических функций.

Производительность

До сих пор все, что удалось узнать об аналитических функциях, свидетельствует о них как об универсальном средстве повышения производительности. Однако при неправильном использовании они могут отрицательно повлиять на производительность.
При использовании этих функций надо опасаться видимой легкости, с которой они позволяют сортировать и фильтровать множества немыслимыми в стандартном языке **SQL** способами. Каждый вызов аналитической функции в списке выбора оператора **SELECT** может использовать свои фрагменты, окна и порядок сортировки. Если они несовместимы (не являются подмножествами друг друга), может выполняться огромный объем сортировки и фильтрования. Например, ранее мы выполняли следующий запрос:

```
SELECT
   ename, deptno,
   SUM(sal) OVER (ORDER BY ename, deptno) sum_ename_deptno,
   SUM(sal) OVER (ORDER BY deptno, ename) sum_deptno_ename
FROM scott.emp
ORDER BY ename, deptno
```

ENAME	DEPTNO	SUM_ENAME_DEPTNO	SUM_DEPTNO_ENAME
ADAMS	20	1100	9850
ALLEN	30	2700	21225
BLAKE	30	5550	24075
CLARK	10	8000	2450
FORD	20	11000	12850
JAMES	30	11950	25025
JONES	20	14925	15825
KING	10	19925	7450
MARTIN	30	21175	26275
MILLER	10	22475	8750
SCOTT	20	25475	18825
SMITH	20	26275	19625
TURNER	30	27775	27775
WARD	30	29025	29025

В этом запросе имеются три конструкции **ORDER BY**, то есть может потребоваться три сортировки. Две сортировки можно объединить, поскольку они выполняются по одним и тем же столбцам, но третью — придется выполнять отдельно. Это — не повод для беспокойства или отказа от использования аналитических функций. Просто надо это учитывать. С помощью аналитических функций можно так же легко написать запрос, использующий все ресурсы компьютера, как и запросы, элегантно и эффективно решающие сложные задачи.