The Power of Oracle SQL

Version 1.1

Contents

Foreword

Главная цель книги описать возможности языка SQL для выборки данных и его диалекта для Oracle в частности. Предполагается, что читатель уже имеет некоторый опыт работы хотя бы с одной реляционной СУБД и общее представление про язык SQL. То есть понимание, что SQL - это декларативный язык, то есть используется для описания того, что требуется получить но без указания как – это определяется планом выполнения запроса. Также желательно понимание, что такое план запроса и как его читать.

В первой части дается описание возможностей Oracle SQL, при необходимости поясняя некоторые базовые вещи языка SQL, и выполняется краткий анализ внутренней механики работы SQL движка. Весь упор делается на конструкцию select, то есть, способы модифицировать данные не рассматриваются. Работа стоимостного оптимизатора преднамеренно была затронута по минимуму, чтоб не увязнуть в деталях и по максимуму фокусироваться на логике запросов, а не особенностях CBO. Некоторые моменты все же невозможно было хотя бы кратко не затронуть, в частности трансформации. Хоть они далеко не всегда имеют «стоимостную» природу, но активируются только в случае стоимостного оптимизатора, в отличие оптимизатора по правилам.

Возможности Oracle растут от версии к версии, так же как и сложность СУБД. Иногда для подчеркивания этой эволюции, в книге упоминаются различные версии – 10g (10.2.0.5), 11g (11.2.0.4) и 12c (12.1.0.2). Вместе с добавлением новых возможностей исправляются и ошибки предыдущих версий, поэтому я старался не упоминать про баги, которые исправлены в последней версии или не важны при описании возможностей языка. Очень важно помнить про развитие от версии к версии, соответственно best practices, которые были актуальны 5, 10 или 15 лет назад могут быть далеко не лучшим подходом на последних версиях.

Задача была с одной стороны сделать всесторонний охват описываемого функционала, а с другой минимизировать объем текста. В результате чего, практически в каждой главе достаточно быстро происходит углубление от базовых возможностей к деталям. Из-за подобной специфики у читателя могут возникать вопросы, которые будут прояснены далее по тексту, так что если что-то остается непонятным или требует уточнений – стоит просто продолжить чтение.

Во второй части рассматриваются конкретные задачи и способы их решения с помощью возможностей Oracle SQL. В некоторых случаях используются элементы языка PL/SQL с целью показать текущие ограничения SQL или продемонстрировать, что PL/SQL может быть предпочтительнее SQL, даже если задача, казалось бы, красиво решается на SQL. Составлять подборку задач специально для PL/SQL я не вижу смысла, ибо для этой цели можно взять практически любую книгу по алгоритмам и адаптировать их для PL/SQL, который является полноценным процедурным языком с элементами ООП. Для понимания преимуществ PL/SQL по сравнению с обычным процедурным языком можно начать с презентации [8]. Отдельная глава посвящена описанию случаев, когда имеет смысл использовать язык PL/SQL или вынести логику за пределы языка SQL, даже если она реализуема с помощью его базовых конструкций – то есть без использования специфики Oracle.

Part I

1. Joins

Как правило, лишь самые примитивные запросы содержат обращение только к одной таблице, в реальных запросах фигурируют несколько таблиц. Под соединением подразумевается сопоставление строк двух наборов данных согласно предикату соединения. Для соединения двух таблиц не обязательно должно быть ключевое слово join или перечисление этих таблиц во фразе from. Например, условия in и exists или даже скалярные подзапросы порождают соединения, если содержат корреляцию с таблицами основного запроса. Более того, подобные запросы могут быть переписаны в семантически эквивалентные использующие соединения. Как будет показано далее, иногда это делается автоматически во время построения плана. Помимо соединений, наборы данных можно объединить в единый результат также с помощью операторов над множествами union/union all/intersect/minus. Далее будет рассмотрен ANSI синтаксис для соединений, так называемый Oracle native syntax (или traditional Oracle outer joins syntax) и различные типы соединений. Немного забегая вперед, отмечу, что существуют достаточно замысловатые запросы, использующие одну единственную таблицу, но это достаточно редкий случай.

ANSI joins

Для демонстрации будут использованы следующие таблицы

```
create table t1(id, name) as
select 1, 'A' from dual union all select 0, 'X' from dual;

create table t2(id, name) as
select 2, 'B' from dual union all select 0, 'X' from dual;
```

- Внутреннее соединение (inner join) – соединение, при котором, возвращаются строки из обеих таблиц, которые удовлетворяют условию соединения.

```
select *
  from t1
  join t2
    on t1.id = t2.id;

        ID N         ID N
---------- - ---------- -
         0 X          0 X
```

Для каждой из строк первой таблицы, условию соединения могут удовлетворять более одной строки второй таблицы, в таком случае будут возвращены они все.

Например, если условие соединения «`t1.id <= t2.id`», строке с id = 0 из первой таблицы соответствуют все строки из второй таблицы.

```
select *
  from t1
  join t2
    on t1.id <= t2.id;

  ID N   ID N
---- - ---- -
   0 X    0 X
   0 X    2 B
   1 A    2 B
```

- Внешнее соединение (outer join) – соединение, при котором возвращаются все строки аналогично внутреннему соединению, а также определенные строки, не удовлетворяющие условию соединения. Для left outer join – все строки из левой таблицы от оператора соединения, для right outer join – из правой таблицы, для full outer join – из обеих таблиц. При этом значения всех столбцов «несоединенной» таблицы для строк, не удовлетворяющих условию, не определены, то есть null.

```
select *
  from t1
  left join t2
    on t1.id = t2.id;

         ID N          ID N
---------- - ---------- -
          0 X           0 X
          1 A
```

```
select *
  from t1
  right join t2
    on t1.id = t2.id;

         ID N          ID N
---------- - ---------- -
          0 X           0 X
                        2 B
```

```
select *
  from t1
  full join t2
    on t1.id = t2.id;

         ID N          ID N
---------- - ---------- -
          0 X           0 X
                        2 B
          1 A
```

Во всех примерах выше не было указано избыточное слово outer, поскольку указание left/right/full автоматически говорит о том, что соединение внешнее. Аналогично, при отсутствии указанных ключевых слов соединение автоматически становится внутренним (inner), так что указывать это явно тоже нет смысла.

Right join практически не используется в реальных запросах, потому как всегда его можно переписать через left join что дает запрос более понятным и улучшает читаемость кода.

Для того чтобы соединить более двух таблиц (наборов данных), Oracle соединит сначала первые две, а потом полученный результат с третьей. Для внутренних соединений при ANSI синтаксисе стоимостной оптимизатор – CBO (cost based optimizer) может перебирать любой порядок соединений, если имеются внешние соединения то порядок в некоторых случаях может определяться тем как перечислены таблицы во фразе from (детальнее про это в разделе «Наглядность и читаемость»).

- Существует также cross join. В этом случае соединяются все комбинации строк обеих таблиц. Такой тип соединения еще называется декартово произведение (cartesian product).

```
select *
  from t1
  cross join t2;
```

```
        ID N          ID N
---------- - ---------- -
         0 X          0 X
         0 X          2 B
         1 A          0 X
         1 A          2 B
```

Other types of joins

- Соединения по равенству (equi joins) – предикаты соединения могут содержать только равенства. Примером не equi join (еще называют theta join) может быть следующий запрос

```
select * from t1 join t2 on t1.id <= t2.id;
```

```
        ID N          ID N
---------- - ---------- -
         0 X          0 X
         0 X          2 B
         1 A          2 B
```

Частным случаем equi join является natural join. В таком случае выполняется внутреннее соединение с предикатами равенства по всем одноименным столбцам в двух таблицах.

```
select * from t1 natural join t2;
```

```
        ID N
---------- -
         0 X
```

Следует отметить, что natural join может быть и внутренним и внешним и полное совпадение столбцов в соединяемых таблицах не требуется.

```
create table t(id, name, dummy) as select 1, 'A', 'dummy' from dual;

select * from t1 natural left join t;

        ID N DUMMY
---------- - -----
         1 A dummy
         0 X
```

В случае equi join содержащих одинаковые имена столбцов с обеих сторон от оператора равенства может быть использовано ключевое слово using для указания условия соединения вместо конструкции on. В этом случае в результирующем наборе будет фигурировать столбец соединения только одной из таблиц.

```
select * from t1 join t2 using (id);

        ID N N
---------- - -
         0  X X
```

- Semi joins. Применяются для запросов с фильтрами вида in или exists и коррелированным подзапросом. Возвращают только одну строку для таблицы даже если более одной строки подзапроса удовлетворяет условию.

```
create table t0(id, name) as
select 0, 'X' from dual union all select 0, 'X' from dual;

select t1.* from t1 where t1.id in (select id from t0);

        ID N
---------- -
         0 X

select t1.* from t1 where exists (select id from t0 where t1.id = t0.id);

        ID N
---------- -
         0 X

select t1.* from t1 join t0 on t1.id = t0.id;

        ID N
---------- -
         0 X
         0 X
```

- Anti joins. Работают по аналогии с semi joins, но возвращают те строки, которые не удовлетворяют условию в where clause. Используются предикаты not in или not exists. Not in, в отличие от not exists, ничего не возвращает, если в подзапросе имеются NULL. Это отличие в логике можно видеть и в названии операции в плане – HASH JOIN ANTI **NA**.

Следует обратить внимание, что в случае EQUI и ANTI соединений в результирующий набор данных все попадают данные только из одной таблицы, вторая таблица используется только для проверки некоторого условия.

Существуют реализации SQL движков, где используются ключевые слова semi/anti. Например, в SQL движке Impala для экосистемы Hadoop могут использоваться конструкции left/right semi join, left/right anti join.

Oracle-specific syntax

До версии 9i в Oracle не было поддержки ANSI соединений, поэтому для соединения таблиц, их имена необходимо было перечислить во from и указать условия соединения в where. Oracle-specific syntax для внешних соединений был доступен с ранних версий, включая Oracle 5.

Запись первого примера inner join выглядит следующим образом.

```
select * from t1, t2 where t1.id = t2.id;
```

Если необходимо указать, что соединение внешнее, то используется конструкция (+), которая применяется к именам колонок. Левое и правое внешние соединения из этой главы могут быть записаны следующим образом. Такая запись называется oracle native (specific) syntax.

```
select * from t1, t2 where t1.id = t2.id(+);
select * from t1, t2 where t1.id(+) = t2.id;
```

Полное внешнее соединение не может быть переписано с помощью родного синтаксиса так, чтобы каждая из таблиц сканировалась лишь раз. Так или иначе, необходимо написать два запроса, объединенных union all.

Говоря про внешние соединения очень важно отличать pre-join и post-join предикаты (Metalink Doc ID 14736.1). Pre-join предикаты вычисляются до того, как столбцу присоединяемой таблицы присвоен null, если не найдена присоединяемая запись, тогда как post-join предикаты вычисляются после этого. Простыми словами – pre-join предикаты соединения, post-join – предикаты фильтрации.

Важно отметить, что предикат фильтра по внутренней таблице может быть (будет) применен до соединения, но это не меняет определения выше. Дополнительную информацию по этому вопросу можно найти в конце главы про трансформации, где упоминается операция «selection».

Проанализируйте следующие запросы на предмет, какие из предикатов являются pre-join, а какие post-join (ответ будет дан непосредственно после запросов).

```
create table t3 as select rownum - 1 id from dual connect by level <= 3;

select *
  from t3
  left join t1
    on t1.id = t3.id
 order by t3.id;

        ID         ID N
---------- ---------- -
         0          0 X
         1          1 A
         2
```

```
select *
  from t3
  left join t1
    on t1.id = t3.id
   and t1.id = 1
 order by t3.id;

        ID         ID N
---------- ---------- -
         0
         1          1 A
         2

select *
  from t3
  left join t1
    on t1.id = t3.id
 where t1.id = 1
 order by t3.id;

        ID         ID N
---------- ---------- -
         1          1 A

select *
  from t3, t1
 where t1.id(+) = t3.id
   and t1.id(+) = 1
 order by t3.id;

        ID         ID N
---------- ---------- -
         0
         1          1 A
         2

select *
  from t3, t1
 where t1.id(+) = t3.id
   and t1.id = 1
 order by t3.id;

        ID         ID N
---------- ---------- -
         1          1 A
```

Первый запрос просто демонстрирует левое внешнее соединение по id. Во втором и четвертом запросах проверка t1.id на равенство единице это pre-join предикат, тогда как в третьем и пятом – post-join.

Концепция pre/post join предикатов применима только тогда имеется внешнее соединение. Для того, чтоб соединения считалось внешним, должен быть предикат, в котором участвуют колонки из обеих таблиц, причем для одной из таблиц используется (+).

Рассмотрим два следующих запроса

```
select *
  from t3, t1
 where 0 = 0
   and t1.id(+) > 1
 order by t3.id;

no rows selected

select *
  from t3, t1
 where nvl2(t3.id, 0, 0) = nvl2(t1.id(+), 0, 0)
   and t1.id(+) > 1
 order by t3.id;

        ID          ID N
---------- ---------- -
         0
         1
         2
```

Первый ничего не вернул, поскольку не было явно сказано, что t1 внешне соединена с t3. Тогда как во втором запросе это указано, пусть и через заведомо истинный предикат.

Если соединение по одним столбцам указано как внешнее, а по другим как внутреннее, то Oracle выполняет соединение как внутреннее. В этом случае применяется трансформация outer to inner join conversion. Про трансформации запросов будет сказано далее.

Примеры запросов, когда фактически будет выполняться внутреннее соединение:

```
select * from t1 left join t2 on t1.id = t2.id where t1.name = t2.name;
select * from t1, t2 where t1.id = t2.id(+) and t1.name = t2.name;
```

Иногда может быть не очевидно как с помощью native syntax указать, что предикат является pre-join. Например, при переписывании следующего запроса можно использовать трюк с rowid (при использовании просто «`and t3.id = 0`» в результате была бы только первая строка).

```
select * from t3 left join t1 on t3.id = t1.id and t3.id = 0;

select *
  from t3, t1
 where t1.id = t3.id(+)
   and nvl2(t1.rowid(+), t3.id, null) = 0;

        ID          ID N
---------- ---------- -
         0          0 X
         1
         2
```

Другой типичный способ указать, что предикат является частью условия соединения – построить на его основе case (детальнее этот прием описан в следующем разделе).

```
select *
  from t1, t2
 where case when t1.id = 0 then t1.id end = t2.id(+)
```

Левая часть равенства будет принимать значение t1.id, только если выполняется соответствующее условие, а в правой части указано, что таблицы соединены внешне.

Последний пример: имеется два предиката по внутренней таблице, один из которых pre-, а другой – post-join.

```
select *
  from t3 left join t2 on t3.id = 1
 where t3.id < 2
order by t2.id;
```

```
        ID          ID N
---------- ---------- -
         1          0 X
         1          2 B
         0
```

Эквивалентный запрос с использование native syntax может выглядеть так (необходимо указать, что t2 внешне соединена с t3 и, кроме того, условие соединение истинно, если t3.id равно 1). Условие можно было бы сократить как в примере выше: «`nvl2(t2.rowid(+), t3.id, null) = 1`», главное чтоб какой-то столбец внешней таблицы фигурировал с (+).

```
select *
  from t3, t2
 where t3.id < 2 and decode(t3.id, 1, 0) = nvl2(t2.id(+), 0, 0)
order by t2.id;
```

```
        ID          ID N
---------- ---------- -
         1          0 X
         1          2 B
         0
```

ANSI vs Oracle native syntax

Каждый вариант записи соединений имеет свои плюсы и минусы. ANSI синтаксис можно рассматривать как синтаксический сахар, потому как он повышает читаемость запросов, но в итоге все запросы в ANSI синтаксисом преобразуются в Oracle native syntax. Тут, однако, есть два исключения – full join и left/right join partition by. То есть в этих случаях без ANSI синтаксиса невозможно добиться соответствующего плана выполнения (примеры таких запросов будут приведены далее). Во всех остальных случаях запрос имеет свой эквивалент с Native syntax, но до Oracle 12c такой эквивалент не всегда мог быть использован, потому что некоторые возможности не были недоступны для разработчиков – в частности lateral views. Итак, начнем с ограничений Oracle native syntax. Весь этот подраздел посвящен детальному сравнению двух подходов и, если такое углубление не интересует, можно сразу перейти к выводам в конце главы.

Ограничения Oracle native syntax

1) Невозможно использовать условия in или or в pre-join предикатах.

Для запроса ниже ограничение на 10-й версии обходится с помощью трюка с case, начиная с 11-й версии подобного ограничения нет.

```
select *
  from t1, t2
 where t1.id = t2.id(+)
   and t2.id(+) in (1, 2, 3);
   and t2.id(+) in (1, 2, 3)
                   *
ERROR at line 4:
ORA-01719: outer join operator (+) not allowed in operand of OR or IN

select *
  from t1, t2
 where t1.id = t2.id(+)
   and case when t2.id(+) in (1, 2, 3) then 1 end = 1;

        ID N         ID N
---------- - ---------- -
         1 A
         0 X
```

В то же время запрос ниже (эквивалентен исходному только если t2.id - целочисленные) выполняется без трюка с case

```
select *
  from t1, t2
 where t1.id = t2.id(+)
   and t2.id(+) between 1 and 3
```

Однако следующий запрос выполняется с ORA-01719 и на 11-й и 12-й версии. Для того, чтобы избежать ошибки опять же можно применить трюк с case.

```
select * from t1 left join t2 on t2.id in (t1.id - 1, t1.id + 1);

        ID N         ID N
---------- - ---------- -
         0 X
         1 A          0 X
         1 A          2 B

select * from t1, t2 where t2.id(+) in (t1.id - 1, t1.id + 1);
select * from t1, t2 where t2.id(+) in (t1.id - 1, t1.id + 1)
                                       *
ERROR at line 1:
ORA-01719: outer join operator (+) not allowed in operand of OR or IN

select *
  from t1, t2
 where case when t2.id(+) in (t1.id - 1, t1.id + 1) then 1 end = 1;
```

Для версии 12.1 запрос можно переписать с помощью коррелированного inline view. Для этого используется ключевое слово lateral.

```
select *
  from t1,
       lateral (select *
                  from t2
                 where t2.id = t1.id - 1
                    or t2.id = t1.id + 1)(+) v
```

Oracle Optimizer team дает такое определение lateral view: **A lateral view is an inline view that contains correlation referring to other tables that precede it in the FROM clause.**

Для коррелированного inline view был добавлен и ANSI синтаксис. Ниже семантически эквивалентный запрос для варианта с lateral (при использовании cross apply соединение было бы внутренним).

```
select *
  from t1
 outer apply (select *
                from t2
               where t2.id = t1.id - 1
                  or t2.id = t1.id + 1) v
```

Следующий запрос тоже выдаст ошибку при использовании в native syntax на любой версии до 12-й версии включительно.

```
select *
  from t1
  left join t2
    on t1.id = t2.id
    or t1.id = 1;

        ID N          ID N
---------- - ---------- -
         0 X           0 X
         1 A           0 X
         1 A           2 B

select * from t1, t2 where t1.id = t2.id(+) or t1.id = 1;
select * from t1, t2 where t1.id = t2.id(+) or t1.id = 1
                                               *
ERROR at line 1:
ORA-01719: outer join operator (+) not allowed in operand of OR or IN
```

Методы обхода те же – case или lateral/outer apply.

Сущность приема с case легче понять, если предикаты связаны конъюнкцией (AND), а не дизъюнкцией (OR).

```
select *
  from t1, t2
 where t1.id = t2.id(+) and t1.id = 1;

        ID N          ID N
---------- - ---------- -
         1 A
```

```
select *
  from t1, t2
 where t1.id = t2.id(+) and t1.id = nvl2(t2.id(+), 1, 1);

        ID N          ID N
---------- - ---------- -
         0 X
         1 A

select *
  from t1, t2
 where case when t1.id = t2.id(+) and t1.id = 1 then 1 end = 1;

        ID N          ID N
---------- - ---------- -
         0 X
         1 A
```

В первом случае предикат «`t1.id = 1`» был post-join, во втором была задействована колонка из второй таблицы, чтоб указать, что это pre-join, а в третьем, при использовании case, необходимость в таком приеме отпала. Поскольку case определяет «неразрывность предикатов», часть case не может быть pre-join, а часть – post-join.

Перейдем к дизъюнкции (или OR-предикатам).

```
select *
  from t1
  left join t2
    on t1.id = t2.id
    or t1.id = 1;

        ID N          ID N
---------- - ---------- -
         0 X          0 X
         1 A          0 X
         1 A          2 B
```

Суть запроса в следующем: если t1.id = 1 то соединить строку со всеми строками из t2, иначе соединить по равенству id.

При переписывании в лоб, native-syntax запрос мог бы выглядеть так

```
select *
  from t1, t2
 where t1.id = t2.id(+)
    or t1.id = 1;
 where t1.id = t2.id(+)
            *
ERROR at line 3:
ORA-01719: outer join operator (+) not allowed in operand of OR or IN
```

Oracle не позволяет выполнить этот запрос, хотя можно обратить внимание, что запрос не имел бы смысла если бы «`t1.id = 1`» был post-join, но SQL движок не должен учитывать смысловую нагрузку.

При попытке указания pre-join природы этого предиката, получим ту же ошибку ORA-1719

```
select *
  from t1, t2
 where t1.id = t2.id(+)
    or t1.id = nvl2(t2.id(+), 1, 1);
```

А вот case позволяет четко указать на «неразрывность условий» и получить желаемый результат.

```
select *
  from t1, t2
 where case when t1.id = 1 or t1.id = t2.id(+) then 1 end = 1;

        ID N          ID N
---------- - ---------- -
         0 X           0 X
         1 A           0 X
         1 A           2 B
```

2) Pre-join предикат не может содержать скалярных подзапросов (запрос ниже работает начиная с версии 12c).

```
select *
  from t3
  left join t1
    on t1.id = t3.id
   and t1.id = (select count(*) from dual)
 order by t3.id;

        ID         ID N
---------- ---------- -
         0
         1          1 A
         2
```

```
select *
  from t3, t1
 where t1.id(+) = t3.id
   and t1.id(+) = (select count(*) from dual)
 order by t3.id;
 order by t3.id
 *
ERROR at line 5:
ORA-01799: a column may not be outer-joined to a subquery
```

Для получения желаемого результата необходимо использовать Inline view в запросе

```
select t3.id, v.id, v.name
  from t3,
       (select id, name from t1 where t1.id = (select count(*) from dual)) v
 where t3.id = v.id(+)
 order by t3.id;

select t3.id, t1.id, t1.name
  from (select t3.*, (select count(*) from dual) cnt from t3) t3, t1
 where t3.id = t1.id(+)
   and t3.cnt = t1.id(+)
 order by t3.id;
```

Либо

```
select t3.id, t1.id, t1.name
  from t3, t1, (select count(*) cnt from dual) v
 where t3.id = t1.id(+)
   and v.cnt = t1.id(+)
 order by t3.id;
```

3) Таблица может быть внешне соединена лишь с одной другой таблицей. Ограничение снято в Oracle 12c.

```
select *
  from t1, t2, t t3
 where t1.id = t2.id
   and t1.id = t3.id(+)
   and t2.name = t3.name(+);
   and t1.id = t3.id(+)
             *
ERROR at line 4:
ORA-01417: a table may be outer joined to at most one other table
```

Для обхода ошибки – таблицы t1 и t2 можно соединить в inline view и потом с t3 (как пример - трансформированный запрос для 11g далее). В ANSI синтаксисе запрос выглядит так

```
select *
  from t1
  join t2
    on t1.id = t2.id
  left join t t3
    on t1.id = t3.id
   and t2.name = t3.name;

        ID N          ID N          ID N DUMMY
---------- - ---------- - ---------- - -----
         0 X          0 X
```

Если посмотреть его после преобразований, то в 11-й версии будет создана вспомогательная inline view (не lateral), тогда как в 12-й версии он будет выглядеть как запрос выше в native syntax. Преобразованные запросы приведены ниже, как смотреть итоговый запрос после преобразований объясняется в следующей главе «Query Transformations».

11g

```
select "from$_subquery$_003"."ID"   "ID",
       "from$_subquery$_003"."NAME" "NAME",
       "from$_subquery$_003"."ID"   "ID",
       "from$_subquery$_003"."NAME" "NAME",
       "T3"."ID"                    "ID",
       "T3"."NAME"                  "NAME",
       "T3"."DUMMY"                 "DUMMY"
  from (select "T1"."ID"   "ID",
               "T1"."NAME" "NAME",
               "T2"."ID"   "ID",
               "T2"."NAME" "NAME"
          from "T1" "T1", "T2" "T2"
         where "T1"."ID" = "T2"."ID") "from$_subquery$_003",
       "T" "T3"
 where "from$_subquery$_003"."NAME" = "T3"."NAME"(+)
   and "from$_subquery$_003"."ID" = "T3"."ID"(+)
```

12c

```
select "T1"."ID"    "ID",
       "T1"."NAME"  "NAME",
       "T2"."ID"    "ID",
       "T2"."NAME"  "NAME",
       "T3"."ID"    "ID",
       "T3"."NAME"  "NAME",
       "T3"."DUMMY" "DUMMY"
  from "T1" "T1", "T2" "T2", "T" "T3"
 where "T1"."ID" = "T3"."ID"(+)
   and "T2"."NAME" = "T3"."NAME"(+)
   and "T1"."ID" = "T2"."ID"
```

Последнее, что хотелось бы отметить в этом разделе – это скорее особенность, а не ограничение. Если таблица в запросе соединена с одной таблицей как внутренняя, а с другой как внешняя, при этом в одном из предикатов фигурирует только эта таблица, то весьма не очевидно как указывать подобный предикат. В примере ниже t2 соединена с t1 как внешняя таблица, а с t3 как внутренняя, соответственно возникает вопрос: как указывать предикат «`tt2.name is not null`» в native syntax.

Для демонстрации будут использованы следующие таблицы

```
create table tt1 as select 'name' || rownum name from dual connect by level <= 3;
create table tt2 as select 'x_name' || rownum name from dual connect by level <= 2;
create table tt3 as select 'y_x_name' || rownum name from dual;

select tt1.name, tt2.name, tt3.name
  from tt1
  left join tt2
    on tt2.name like '%' || tt1.name || '%'
  left join tt3
    on tt3.name like '%' || tt2.name || '%'
   and tt2.name is not null;

NAME       NAME       NAME
---------- ---------- ----------
name1      x_name1    y_x_name1
name2      x_name2
name3
```

При переписывании в лоб в native syntax – результат получается неверный независимо от того, указан ли предикат «`tt2.name is not null`» как внешний или нет.

```
select tt1.name, tt2.name, tt3.name
  from tt1, tt2, tt3
 where tt2.name(+) like '%' || tt1.name || '%'
   and tt3.name(+) like '%' || tt2.name || '%'
   and tt2.name is not null;

NAME       NAME       NAME
---------- ---------- ----------
name1      x_name1    y_x_name1
name2      x_name2
```

```
select tt1.name, tt2.name, tt3.name
  from tt1, tt2, tt3
 where tt2.name(+) like '%' || tt1.name || '%'
   and tt3.name(+) like '%' || tt2.name || '%'
   and tt2.name(+) is not null;

NAME       NAME       NAME
---------- ---------- ----------
name1      x_name1    y_x_name1
name2      x_name2
name3                 y_x_name1
```

Для того, чтобы указать, что предикат имеет отношение к соединению таблиц t2 и t3 можно применить прием, который был упомянут в конце раздела в «Oracle-specific синтаксис».

```
nvl2(tt2.name, 0, null) = nvl2(tt3.rowid(+), 0, 0)
```

В этом случае указано, что t3 внешне соединена с t2 и предикат истинный когда «`tt2.name`» не null.

Учитывая особенности запроса, предикаты

```
and tt3.name(+) like '%' || tt2.name || '%'
and nvl2(tt2.name, 0, null) = nvl2(tt3.rowid(+), 0, 0)
```

Можно объединить в один

```
and tt3.name(+) like nvl2(tt2.name, '%' || tt2.name || '%', null)
```

Для 12-й версии, как и в предыдущих примерах, могут быть использованы lateral или outer apply или же inline view. Oracle при трансформации этого запроса из ANSI создает lateral view.

Unnesting collections

Предположим, есть таблица, у которой один столбец nested table. Ниже определения типа коллекции и скрипт создания таблицы.

```
create or replace type numbers as table of number
/
create table tc (id int, nums numbers) nested table nums store as nums_t
/
insert into tc
select -1 id, numbers(null) nums from dual
union all select 0 id, numbers() nums from dual
union all select 1 id, numbers(1) nums from dual
union all select 2 id, numbers(1,2) nums from dual;
```

Если стоит задача: для каждой строки, где есть непустая коллекция вывести ее элементы, то она может быть решена одним из следующих способов.

```
select tc.id, x.column_value
from tc, table(tc.nums) x -- 1
--from tc, lateral(select * from table(tc.nums)) x -- 2
--from tc cross apply (select * from table(tc.nums)) x -- 3
--from tc cross join table(tc.nums) x -- 4
;

        ID COLUMN_VALUE
---------- ------------
        -1
         1            1
         2            1
         2            2
```

Второй и третий способы работают, начиная с 12c.

Теперь усложним задачу: в результат должны попасть, в том числе строки, для которых коллекция пустая.

```
select tc.id, x.column_value
from tc, table(tc.nums)(+) x -- 1
--from tc, lateral(select * from table(tc.nums))(+) x -- 2
--from tc cross apply (select * from table(tc.nums))(+) x -- 3
--from tc outer apply (select * from table(tc.nums)) x -- 4
--from tc, table(tc.nums) x where nvl2(x.column_value(+), 0, 0) = nvl2(tc.id,
0, 0) -- 5
--from tc left join table(tc.nums) x on nvl2(x.column_value, 0, 0) =
nvl2(tc.id, 0, 0) -- 6
;

        ID COLUMN_VALUE
---------- ------------
        -1
         0
         1            1
         2            1
         2            2
```

Здесь стоит отметить несколько моментов.

- вариант со смесью cross apply и native syntax возвращает корректный результат, но подобное использовать не стоит.
- в вариантах 5 и 6 использовались искусственные заведомо истинные предикаты, в которых участвуют обе таблицы. Вариант 5 возвращает некорректный результат (отсутствует строка с id = 0) из-за бага.

Однако если использовать вариант, когда nested table не хранится, то получается, что наоборот вариант 5 работает корректно, а 6 нет. То есть:

```
with tc as
(select -1 id, numbers(null) nums from dual
union all select 0 id, numbers() nums from dual
union all select 1 id, numbers(1) nums from dual
union all select 2 id, numbers(1,2) nums from dual)
select tc.id, x.column_value
from tc left join table(tc.nums) x on nvl2(x.column_value, 0, 0) =
nvl2(tc.id, 0, 0);

        ID COLUMN_VALUE
---------- ------------
        -1
         1            1
         2            1
         2            2
```

Подобное поведение воспроизводимо на всех версиях до 12.1.0.2 включительно, одно из описаний: Bug 20363558 : WRONG RESULTS FOR ANSI JOIN ON NESTED TABLE.

Обойти баг используя ANSI синтаксис можно следующим образом:

```
select tc.id, x.column_value
from tc cross join table(case when cardinality(tc.nums) = 0 then
numbers(null) else tc.nums end) x
```

При использовании varray nested вместо table, то есть

```
create or replace type num_array as varray(32767) of number
/
create table tc (id int, nums num_array)
/
```

Будет всегда некорректный результат в случае 6, независимо от того хранится varray в таблице или конструируется тем или иным образом на лету.

Как бы там ни было, запрос всегда корректно работает, если указывать (+) возле выражения table(). При попытке реализовать внешнее соединение с указанием (+) в предикате, могут быть некорректные результаты в зависимости от nested table/varray и способа хранения.

Коррелированные inline view и подзапросы

Ранее уже несколько раз демонстрировались коррелированные inline view с использованием lateral/apply. До 12c подобной функциональности можно было достичь путем использования cast + multiset/collect и ключевого слова table. Минус подходов в том, что надо иметь необходимый тип коллекции для результата (при использовании PL/SQL типа как результата – SQL тип создастся Oracle автоматически).

Например, если стоит задача для каждой строки таблицы сгенерировать число строк равное значению id и присоединить к исходной таблице, то в 12c это можно реализовать так.

```
select t3.id, v.idx
  from t3,
       lateral (select rownum idx
                  from dual
                 where rownum <= t3.id
                connect by rownum <= t3.id)(+) v;

select t3.id, v.idx
  from t3
 outer apply (select rownum idx
                from dual
               where rownum <= t3.id
              connect by rownum <= t3.id) v;
```

На старых версиях это могло бы выглядеть так

```
select t3.id, v.column_value idx
  from t3,
       table(cast(multiset (select rownum
                      from dual
                     where rownum <= t3.id
                    connect by rownum <= t3.id) as sys.odcinumberlist))(+) v;

select t3.id, v.column_value idx
  from t3,
       table (select cast(collect(rownum) as sys.odcinumberlist)
                from dual
               where rownum <= t3.id
              connect by rownum <= t3.id)(+) v;
```

Вариант с cast + multiset предпочтительнее с точки зрения производительности.

Результат во всех случаях:

```
        ID        IDX
---------- ----------
         0
         1          1
         2          1
         2          2
```

Еще одним ограничением для коррелированных подзапросов и cast + multiset/collect до 12c было видимость внешних таблиц только с одним уровнем вложенности. В примере ниже m2, m4, m5 могут быть подсчитаны только в 12c (все выражения реализуют одну и ту же логику).

```
select id,
       greatest((select min(id) mid from t3 where t3.id > t.id), 1) m1,
       (select max(mid)
          from (select min(id) mid
                  from t3
                 where t3.id > t.id
                union
                select 1 from dual) z) m2,
       (select max(value(v))
          from table(cast(multiset (select min(id) mid
                                      from t3
                                     where t3.id > t.id
                                    union
                                    select 1 from dual) as sys.odcinumberlist)) v) m3,
       (select max(value(v))
          from table (select cast(collect(mid) as sys.odcinumberlist) col
                        from (select min(id) mid
                                from t3
                               where t3.id > t.id
                              union
                              select 1 from dual) z) v) m4,
       (select value(v)
          from table(cast(multiset (select max(mid)
                                     from (select min(id) mid
                                             from t3
                                            where t3.id > t.id
                                           union
                                           select 1 from dual) z) as
                                sys.odcinumberlist)) v) m5
  from t3 t
 where t.id = 1;

        ID         M1         M2         M3         M4         M5
---------- ---------- ---------- ---------- ---------- ----------
         1          2          2          2          2          2
```

Не всегда скалярный подзапрос можно упростить до одного уровня вложенности, соответственно, если необходимо реализовать сложную логику с несколькими уровнями вложенности, то до 12c это можно было сделать только с помощью UDF либо же переписыванием запроса через явные соединения. В 12c тоже не рекомендуется употреблять подобными скалярными подзапросами, ибо они, во-первых, могут некорректно трансформироваться, а во-вторых, могут приводить к неэффективному плану.

Последнее, что стоит отметить, конструкции lateral/apply не позволяют делать настолько гибкую корреляцию как collect/multiset. Например, невозможно указать поле из главной таблицы в start with. Последний пример выдаст ошибку, если раскомментировать «*t1.id*»

```
select t1.*,
       l.*
  from t1,
       table(cast(multiset (select id
                             from t3
                            start with t3.id = t1.id
                           connect by prior t3.id + 1 = t3.id) as numbers)) l;
```

```
select t1.*, l.*
  from t1,
       table (select cast(collect(id) as numbers)
                from t3
               start with t3.id = t1.id
              connect by prior t3.id + 1 = t3.id) l;

select t1.*, l.*
  from t1,
       lateral (select id
                  from t3
                 start with t3.id = 0 -- t1.id
                connect by prior t3.id + 1 = t3.id) l;
```

Трансформация из ANSI в Native

Для демонстрации будут использованы следующие таблицы

```
create table fact as (select 1 value, 1 dim_1_id, 1 dim_2_id, 'A' type from
dual);
create table dim_1 as (select 1 id, 1 dim_n_id from dual);
create table dim_n as (select 1 id, 1 value from dual);
create table map as (select 1 value, 'DETAILED VALUE' category from dual);
```

И запрос

```
select fact.*, map.*
  from fact
  join dim_1
    on dim_1.id = fact.dim_1_id
  join dim_n
    on dim_1.dim_n_id = dim_n.id
  left join map
    on fact.type in ('A', 'B', 'C')
   and ((map.category = 'FACT VALUE' and map.value = fact.value) or
       (map.category = 'DETAILED VALUE' and map.value = dim_n.value));
```

После преобразований запрос выглядит следующим образом

```
select "FACT"."VALUE"                "VALUE",
       "FACT"."DIM_1_ID"             "DIM_1_ID",
       "FACT"."DIM_2_ID"             "DIM_2_ID",
       "FACT"."TYPE"                 "TYPE",
       "VW_LAT_3C55142F"."ITEM_1_0" "VALUE",
       "VW_LAT_3C55142F"."ITEM_2_1" "CATEGORY"
  from "FACT" "FACT",
       "DIM_1" "DIM_1",
       "DIM_N" "DIM_N",
       lateral((select "MAP"."VALUE" "ITEM_1_0", "MAP"."CATEGORY" "ITEM_2_1"
                  from "MAP" "MAP"
                 where ("FACT"."TYPE" = 'A' or "FACT"."TYPE" = 'B' or
                       "FACT"."TYPE" = 'C')
                   and ("MAP"."CATEGORY" = 'FACT VALUE' and
                       "MAP"."VALUE" = "FACT"."VALUE" or
                       "MAP"."CATEGORY" = 'DETAILED VALUE' and
                       "MAP"."VALUE" = "DIM_N"."VALUE")))(+) "VW_LAT_3C55142F"
 where "DIM_1"."DIM_N_ID" = "DIM_N"."ID"
   and "DIM_1"."ID" = "FACT"."DIM_1_ID"
```

Как уже было замечено то же самое можно написать с ANSI синтаксисом и по-другому – с использованием outer apply. Но ключевой момент, что Oracle создает дополнительную non-mergeable lateral view. Запись семантически эквивалентного запроса с помощью Oracle native syntax выглядит следующим образом (без дополнительной inline view переписать на 11g невозможно из-за того, что map не может быть одновременно внешне соединена и с fact и с dim_n)

```
select *
  from (select fact.*, dim_n.value as value_1
          from fact, dim_1, dim_n
         where dim_1.id = fact.dim_1_id
           and dim_1.dim_n_id = dim_n.id) sub,
       map
 where case when decode(map.rowid(+), map.rowid(+), sub.type) in ('A', 'B',
'C') then 1 end = 1
   and decode(map.category(+), 'FACT VALUE', sub.value, 'DETAILED VALUE',
sub.value_1) = map.value(+);
```

Запрос не так легко читается как записанный в ANSI синтаксисе, но при такой записи нет non-mergeable correlated inline view и можно добиться лучшей производительности, поскольку становится возможным соединение HASH JOIN. Этот метод соединения с lateral view исключен.

Очень важный момент, что если теперь взять предикаты в том виде, как они указаны для native syntax и использовать в запросе ANSI, то создания lateral view можно избежать.

```
select fact.*, map.*
  from fact
  join dim_1
    on dim_1.id = fact.dim_1_id
  join dim_n
    on dim_1.dim_n_id = dim_n.id
  left join map
    on case when decode(map.rowid, map.rowid, fact.type) in ('A', 'B', 'C')
then 1 end = 1
   and decode(map.category, 'FACT VALUE', fact.value, 'DETAILED VALUE',
dim_n.value) = map.value
```

Желающие могут убедиться, что для этого запроса отсутствует операция VIEW в плане, а итоговый запрос выглядит примерно так:

```
select "FACT"."VALUE"    "VALUE",
       "FACT"."DIM_1_ID" "DIM_1_ID",
       "FACT"."DIM_2_ID" "DIM_2_ID",
       "FACT"."TYPE"     "TYPE",
       "MAP"."VALUE"     "VALUE",
       "MAP"."CATEGORY"  "CATEGORY"
  from "FACT"  "FACT",
       "DIM_1" "DIM_1",
       "DIM_N" "DIM_N",
       "MAP"   "MAP"
 where case when decode("MAP".ROWID(+), "MAP".ROWID(+), "FACT"."TYPE") in
('A', 'B', 'C') then 1 end = 1
   and "MAP"."VALUE"(+) = decode("MAP"."CATEGORY"(+), 'FACT VALUE',
"FACT"."VALUE", 'DETAILED VALUE', "DIM_N"."VALUE")
   and "DIM_1"."DIM_N_ID" = "DIM_N"."ID"
   and "DIM_1"."ID" = "FACT"."DIM_1_ID"
```

Преобразование ANSI -> native усовершенствуется от версии к версии и те запросы, которые ранее приводили к созданию lateral views, на более новых версиях могут быть преобразованы без них.

Как было замечено ранее, full join не преобразуется в native syntax а также left/right join partition by.

Для демонстрации последнего решим следующую задачу. Пусть есть таблица с днями недели и таблица с презентациями для каждого автора, которая содержит имя, день недели и время. Вывеси для каждого автора все дни недели и количество презентаций в каждом из них.

```
create table week(id, day) as
select rownum, to_char(trunc(sysdate, 'd') + level - 1, 'fmday')
  from dual
connect by rownum <= 7;

create table presentation(name, day, time) as
select 'John', 'monday', '14' from dual
union all
select 'John', 'monday', '9' from dual
union all
select 'John', 'friday', '9' from dual
union all
select 'Rex', 'wednesday', '11' from dual
union all
select 'Rex', 'friday', '11' from dual;
```

Результат можно получить следующим запросом

```
select p.name, w.day, count(p.time) cnt
  from week w
  left join presentation p partition by(p.name)
    on w.day = p.day
 group by p.name, w.day, w.id
 order by p.name, w.id;
```

```
NAME DAY              CNT
---- --------- ----------
John monday             2
John tuesday            0
John wednesday          0
John thursday           0
John friday             1
John saturday           0
John sunday             0
Rex  monday             0
Rex  tuesday            0
Rex  wednesday          1
Rex  thursday           0
Rex  friday             1
Rex  saturday           0
Rex  sunday             0

14 rows selected.
```

Итоговый запрос выглядит так

```
select "from$_subquery$_003"."NAME_0" "NAME",
       "from$_subquery$_003"."QCSJ_C000000000300000_2" "DAY",
       count("from$_subquery$_003"."TIME_4") "CNT"
  from (select "P"."NAME" "NAME_0",
               "W"."ID"   "ID_1",
               "W"."DAY"  "QCSJ_C000000000300000_2",
               "P"."DAY"  "QCSJ_C000000000300001",
               "P"."TIME" "TIME_4"
          from "PRESENTATION" "P" partition by("P"."NAME")
         right outer join "WEEK" "W"
            on "W"."DAY" = "P"."DAY") "from$_subquery$_003"
 group by "from$_subquery$_003"."NAME_0",
          "from$_subquery$_003"."QCSJ_C000000000300000_2",
          "from$_subquery$_003"."ID_1"
 order by "from$_subquery$_003"."NAME_0", "from$_subquery$_003"."ID_1"
```

partition by (p.name) означает, что для каждого имени из p будут присоединены все строки из w. Аналогичный результат можно получить без outer join partition by, но понадобится дополнительное соединение.

```
select w.name, w.day, count(p.time) cnt
  from (select p0.name, w0.*
          from (select distinct name from presentation) p0, week w0) w,
       presentation p
 where w.day = p.day(+)
   and w.name = p.name(+)
 group by w.name, w.day, w.id
 order by w.name, w.id;
```

Рассмотрим теперь запрос после преобразований для случая SEMI join рассмотренного ранее. Запрос «`select t1.* from t1 where t1.id in (select id from t0)`» преобразовывается в следующий

```
select "T1"."ID" "ID", "T1"."NAME" "NAME"
  from "M12"."T0" "T0", "M12"."T1" "T1"
 where "T1"."ID" = "T0"."ID"
```

Поскольку нет специального обозначения для SEMI join предикатов, то в условии стоит просто равенство, соответственно если в плане изначального запроса соединение HASH JOIN SEMI, то при попытке построить план для преобразованного будет просто HASH JOIN. Поэтому надо быть внимательным при работе с итоговым запросом после преобразований, аналогичная ситуация для ANTI предикатов. Для получения верного результата с помощью преобразованного запроса надо добавить в список выборки t1.rowid и ключевое слово distinct.

Для рассматриваемого запроса можно как изменить метод соединения, так и вовсе отключить преобразование в соединение. Это делается с помощью хинтов, как показано ниже.

```
select t1.* from t1 where t1.id in (select /*+ use_nl(t0) */ id from t0);
select /*+ no_query_transformation */ t1.* from t1 where t1.id in (select id from t0);
```

Во втором случае для каждой строки из t1 будет выполняться полное сканирование t0 до нахождения первого соответствия (при наличие индекса это мог бы быть индексный доступ) в поисках соответствия.

```
-----------------------------------
| Id  | Operation           | Name |
-----------------------------------
|   0 | SELECT STATEMENT    |      |
|   1 |  FILTER             |      |
|   2 |   TABLE ACCESS FULL| T1   |
|   3 |   TABLE ACCESS FULL| T0   |
-----------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter( EXISTS (SELECT 0 FROM "T0" "T0" WHERE "ID"=:B1))
   3 - access("ID"=:B1)
```

В общем случае план исходного запроса и план преобразованного могут отличаться по ряду причин, но преобразованный запрос – это всего лишь приблизительная запись на языке SQL того, что по факту будет выполняться.

В случае использования стоимостного оптимизатора (Cost Based Optimizer - CBO), исходные запросы, выглядящие совершенно по-разному, могут быть преобразованы в одинаковый итоговый запрос с одинаковым планом выполнения, благодаря преобразованиям запросов.

Если же используется Rule Based Optimizer, то итоговый запрос и план во многом определяется именно тем как написан исходный запрос. В частности, метод соединения SEMI JOIN не реализован для RBO.

```
select /*+ rule */ t1.* from t1 where t1.id in (select id from t0);

----------------------------------------
| Id  | Operation             | Name     |
----------------------------------------
|   0 | SELECT STATEMENT      |          |
|   1 |  MERGE JOIN           |          |
|   2 |   SORT JOIN           |          |
|   3 |    TABLE ACCESS FULL  | T1       |
|*  4 |   SORT JOIN           |          |
|   5 |    VIEW               | VW_NSO_1 |
|   6 |     SORT UNIQUE       |          |
|   7 |      TABLE ACCESS FULL| T0       |
----------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("T1"."ID"="ID")
       filter("T1"."ID"="ID")
```

Стоимостный оптимизатор был введен в версии 7.3 и усовершенствуется от версии к версии, так что Oracle еще несколько лет назад рекомендовал полностью отказаться от

использования оптимизатора по правилам. Пример был приведен, чтоб подчеркнуть, отсутствие некоторых методов соединения и полное отсутствие трансформаций для RBO.

В Oracle есть три основных метода соединения двух наборов данных – merge join, hash join и nested loops. Плюс, как было показано выше, это может быть FILTER (механика работы которого очень похожа на nested loops), если итоговые данные берутся только из одного набора, но фильтруются по второму.

Наглядность и читаемость

Для демонстрации рассмотрим простую схему из таблицы фактов и одного измерения. Для каждой записи в таблице фактов используются две координаты этого измерения.

```
create table fact_ as (select 1 value, 1 dim_1_id, 2 dim_2_id from dual);
create table dim_ as (select rownum id, 'name'||rownum name from dual connect
by rownum <= 2);
```

Предположим необходимо соединить факт с измерением по обеим координатам

```
select *
from fact_ f
join dim_ d1 on f.dim_1_id = d1.id
join dim_ d2 on f.dim_2_id = d2.id
```

В случае ANSI преимущества в наглядности следующие

1) Условия соединения для каждого из измерений вынесены в соответствующий раздел «on».

   В случае внутренних соединений может быть некоторая неоднозначность: где указывать фильтры по таблицам – в where или on. Можно взять за правило: указывать в where только фильтры по таблице фактов. При внешнем соединении такой неоднозначности очевидно нет.
2) Нет необходимости создавать inline/lateral views в случае внешних соединений. Однако в упрощенной наглядности есть и минус – худший контроль плана.
3) Есть дополнительный контроль на предикаты: можно указывать столбцы только тех таблиц, которые перечислены до текущей. То есть, в примере выше нельзя указать

   ```
   select *
   from fact_ f
   join dim_ d1 on f.dim_1_id = d2.id
   join dim_ d2 on f.dim_2_id = d2.id
   ```

   При использовании native syntax все предикаты перечислены в where. Дополнительный контроль может быть опять же за счет создания inline views.

При использовании native syntax можно следовать разнообразным правилам перечисления и оформления предикатов в where, но если, скажем, внутренне соединяется порядка 20 таблиц, то это предикаты равно будет выглядеть «как куча».

Нелепость такого подхода подтверждается тем, что никто при использовании ANSI синтаксиса и внутренних соединений не станет перечислять все таблицы через cross join и все предикаты в where.

4) Факт принадлежности предиката к конкретному соединению определятся тем, где он находится в запросе. В случае native syntax иногда приходится прибегать к различным трюкам, чтоб указать это, как было показано в конце раздела «Ограничения Oracle native syntax».

В заключение стоит добавить, что гибкость ANSI порой позволяет писать весьма не очевидные запросы. Я предпочел бы не использовать подобные возможности, но тем не менее, считаю нужным о них рассказать. Итак, в случае ANSI может играть роль, в каком порядке перечислены таблицы и соединения во фразу from.

Для запросов ниже отличаются результаты, поскольку в первом случае соединяются t1, t2, а потом результат с t3; тогда как, во втором случае соединяется t2, t3 а потом результат с t1.

```
select t1.*, t2.*, t3.*
  from t1
  full join t2
    on t1.id = t2.id
  join t3
    on t2.id = t3.id
 order by t1.id;
```

```
        ID N         ID N         ID
---------- - ---------- - ----------
         0 X          0 X          0
                      2 B          2
```

```
select t1.*, t2.*, t3.*
  from t2
  join t3
    on t2.id = t3.id
  full join t1
    on t1.id = t2.id
 order by t1.id;
```

```
        ID N         ID N         ID
---------- - ---------- - ----------
         0 X          0 X          0
         1 A
                      2 B          2
```

Однако в первом запросе можно изменить порядок соединения, не меняя порядка перечисления таблиц.

```
select t1.*, t2.*, t3.*
  from t1
  full join (t2 join t3 on t2.id = t3.id) on t1.id = t2.id
 order by t1.id;
```

```
        ID N         ID N         ID
---------- - ---------- - ----------
         0 X          0 X          0
         1 A
                      2 B          2
```

Этот запрос будет выглядеть еще более неоднозначно, если убрать скобки, без которых он тоже выполнится. Тем не менее, это работает согласно документации. В запросе на месте table_reference может быть join_clause. В примитивном случае это будет выглядеть так:

```
-- table_reference in ()
select * from (dual);
-- join_clause in ()
select * from (dual cross join dual);
```

Mixing syntax

Каждый решает для себя, какой из синтаксисов для соединений использовать: ANSI или native. В редких случаях, я считаю допустимо в одном и том же запросе, но на разных уровнях (или в разных подзапросах) использовать разный синтаксис: например, если принято за стандарт разработки использовать ANSI, но в части запроса не удается зафиксировать план из-за преобразования ANSI -> native и неявного создания inline views. Однако Oracle позволяет смешивать синтаксисы даже в одном и том же разделе from. Следующие примеры будут рассмотрены с целью описать «как делать НЕ надо».

Итак, если указать ANSI inner join, но в условие соединения указать (+), то по факту будет выполняться внешнее соединение.

```
select * from t1 join t2 on t1.id = t2.id(+)
```

Преобразованный запрос выглядит так

```
select "T1"."ID"   "ID",
       "T1"."NAME" "NAME",
       "T2"."ID"   "ID",
       "T2"."NAME" "NAME"
  from "T1" "T1", "T2" "T2"
 where "T1"."ID" = "T2"."ID"(+)
```

Идем дальше, попробуем выполнить такой запрос

```
select *
  from t1, t2
  left join t3
    on t3.id = t2.id + 1;

        ID N         ID N         ID
---------- - ---------- - ----------
         0 X          0 X          1
         1 A          0 X          1
         0 X          2 B
         1 A          2 B
```

Обратите внимание, во фразе from сначала перечисляются две таблицы, а потом указывается внешнее соединение с третьей.

Теперь попробуем написать смешанный аналог такого запроса

```
select *
  from t1
 cross join t2
  left join t3
    on t3.id = t2.id + 1
   and t3.id = t1.id;

        ID N         ID N         ID
---------- - ---------- - ----------
         1 A          0 X          1
```

```
         0 X          0 X
         1 A          2 B
         0 X          2 B
```

При попытке использовать t1 в соединении с t3 получаем ошибку, то есть в конструкции on видна только таблица t2.

```
select *
  from t1, t2
  left join t3
    on t3.id = t2.id + 1
   and t3.id = t1.id;
   and t3.id = t1.id
               *
ERROR at line 5:
ORA-00904: "T1"."ID": invalid identifier
```

При попытке указать внешнее соединение t1 с t3 через (+) в where получаем ORA-25156.

```
select *
  from t1, t2
  left join t3
    on t3.id = t2.id + 1
 where t3.id(+) = t1.id;
 where t3.id(+) = t1.id
       *
ERROR at line 5:
ORA-25156: old style outer join (+) cannot be used with ANSI joins
```

Логично было бы ошибку «ORA-25156» выдавать всегда, когда есть ключевое слово join и (+) в одной и той же фразе from, но первый запрос из этого раздела успешно отрабатывает на любых версиях 10g, 11g, 12c.

Контроль плана выполнения

Следует быть внимательным при использовании ANSI синтаксиса с внешними соединениями и хинтов. Иногда в результате создания новых lateral/inline views после трансформации хинты могут стать недействительными. Более того, некоторые хинты вообще не применимы при использовании ANSI, например, qb_name – бывает полезен при использовании в хинтах основного запроса имен таблиц из inline view.

Для запросов ниже посмотрим информацию про алиасы с помощью «`select * from table(dbms_xplan.display_cursor(format => 'BASIC ALIAS'));`».

```
select --+ qb_name(q)
 *
  from t1
  join t2
    on t1.id = t2.id;

select --+ qb_name(q)
 *
  from t1, t2
 where t1.id = t2.id;
```

Результат следующий (при использовании ANSI хинт стал недействительным)

```
Query Block Name / Object Alias (identified by operation id):
------------------------------------------------------------

   1 - SEL$695B99D2
   2 - SEL$695B99D2 / T1@SEL$1
   3 - SEL$695B99D2 / T2@SEL$1

Query Block Name / Object Alias (identified by operation id):
------------------------------------------------------------

   1 - Q
   2 - Q / T1@Q
   3 - Q / T2@Q
```

Ограничения ANSI

Помимо того, что ANSI предоставляет худший контроль над планом выполнения, до 12c было еще одно ограничение – использование ANSI в коррелированных подзапросах. Точнее я бы назвал это не ограничением, а багом.

Рассмотрим следующий синтетический пример

```
select t3.id,
       (select count(t2.rowid) + count(t1.rowid)
          from t2
          join t1
            on t2.id = t1.id
           and t2.id = t3.id) x
  from t3
 order by t3.id;
           and t2.id = t3.id) x
                       *
ERROR at line 6:
ORA-00904: "T3"."ID": invalid identifier
```

Если в условии соединения используется колонка таблицы из основного запроса, то выдается ошибка, что она не видна. Как и во всех прочих случаях, от коррелированного подзапроса (в данном случае еще и скалярного) можно уйти, переписав через явные соединения (в группировку добавлен столбец t3.rowid, потому что нигде не сказано, что значение t3.id - уникально).

```
select t3.id, count(t2.rowid) + count(t1.rowid) x
  from t3
  left join(t2 join t1 on t2.id = t1.id) on t3.id = t2.id
 group by t3.rowid, t3.id
 order by t3.id;

        ID          X
---------- ----------
         0          2
         1          0
         2          0
```

Но коррелированные скалярные подзапросы могут быть предпочтительнее в случае малого числа различных значений t3.id и применения scalar subquery caching.

Итак, если перенести предикат, содержащий t3, в конструкцию where, то запрос будет работать.

```
select t3.id,
       (select count(t2.rowid) + count(t1.rowid)
          from t2
          join t1
            on t2.id = t1.id
         where t2.id = t3.id) x
  from t3
 order by t3.id;
```

Если же в подзапросе внешнее соединение, содержащее столбец таблицы основного запроса, то такой предикат невозможно указать в версии 11g. Ниже результат выполнения на 12c.

```
select t3.id,
       (select count(t2.rowid) + count(t1.rowid)
          from t2
          left join t1
            on t2.id = t1.id
           and t3.id > 0
         where t2.id = t3.id) x
  from t3
 order by t3.id;

        ID          X
---------- ----------
         0          1
         1          0
         2          1
```

В 11g результат можно получить следующим образом (перенеся логику в select list), но это нельзя считать полноценной альтернативой.

```
select t3.id,
       (select count(t2.rowid) + decode(sign(t3.id), 1, count(t1.rowid), 0)
          from t2
          left join t1
            on t2.id = t1.id
         where t2.id = t3.id) x
  from t3
 order by t3.id;

        ID          X
---------- ----------
         0          1
         1          0
         2          1
```

Однако в скалярный подзапрос можно добавить конструкцию table с корреляцией:

```
select t3.id,
       (select count(t2.rowid) + count(tt.column_value)
          from t2
          left join table(cast(multiset (select nvl2(t2.rowid, 1, null)
                                           from t1
                                          where t2.id = t1.id
                                            and t3.id > 0) as
sys.odcinumberlist)) tt
            on 1 = 1
         where t2.id = t3.id) x
  from t3
 order by t3.id;
```

Ну и напоследок, через native syntax это переписывается, например, так:

```
select t3.id,
       (select count(t2.rowid) + count(t1.rowid)
          from t2, t1
         where t2.id = t3.id
           and t2.id = t1.id(+)
           and decode(sign(t3.id), 1, 0) = nvl2(t1.id(+), 0, 0)) x
  from t3
 order by t3.id;
```

Те же самые проблемы возникают в случае с ANSI и коррелированными подзапросами в конструкции where, а не только скалярными подзапросами в select list.

Резюме

Как правило, в запросах фигурирует больше одной таблицы (или одна и та же таблица встречается более одного раза) и наборы данных из таблиц должны быть так или иначе соединены (за исключением случаев union/union all/intersect/minus, когда выполняются операторы над множествами или некоррелированных подзапросов, когда не требуется устанавливать соответствие строк из разных наборов) для получения единого результирующего набора. Соединения могут быть указаны как явно с помощью ключевого слова join так и в традиционном Oracle синтаксисе путем перечисления таблиц в where и указания внешних соединений при помощи (+). (ANTI) SEMI соединения могут быть указаны с помощью условий (not) in/exists.

Одну и ту же логику можно записать совершенно по-разному, но не всегда для разных записей можно добиться одного и того же плана. Перед выполнением запроса и построением его плана выполняется преобразование ANSI синтаксиса в native syntax и применяются разнообразные трансформации запросов – об этом пойдет речь в следующей главе.

Сравнивая ANSI и native syntax, стоит отметь, что ANSI предоставляет большую наглядность, но дает меньше контроля над планом выполнения – могут создаваться дополнительные inline/lateral views, которые могут ограничивать возможные варианты соединений (например, невозможность HASH JOIN) или делать невалидными хинты. Два вида ANSI соединений – full и outer join partition by не могут быть выражены в родном синтаксисе с сохранением плана выполнения.

Исторически сложилось, что ANSI синтаксис был реализован значительно позже native syntax и содержал огромное число багов, в результате чего многие остерегались его использовать. Проблемы с багами есть и в версии 12c, но как было описано в разделе «Unnesting collections» неожиданные результаты могут быть для ANSI так и для native syntax.

2. Query transformations

В некоторых случаях запросы, которые выглядят совершенно по-разному, имеют одинаковый план выполнения и производительность. Это достигается за счет трансформаций, в результате которых исходные запросы приводятся к единому итоговому запросу.

Например, для таблиц из предыдущей главы запросы ниже

```
select t1.* from t1 left join t2 on t1.id = t2.id where t2.id is null;

select t1.* from t1 where not exists (select t2.id from t2 where t1.id =
t2.id);

select t1.* from t1, t2 where t1.id = t2.id(+) and t2.id is null;


select t1.* from t1 where t1.id not in (select t2.id from t2);
```

Имеют одинаковую производительность и план. Во всех случаях способ соединения HASH JOIN ANTI (за исключением последнего запроса, в котором соединение HASH JOIN ANTI NA и результат будет пустым, если в таблице t2 хотя бы один id is null).

Смотреть какие трансформации были применены к запросу и итоговый вид запроса можно с помощью трассировки, выполнив одну из следующих команд (детальное описание команд, а также анализ файлов трассировки выходит за рамки данной книги, дальнейшее изучение можно начать с [1]) перед выполнением запроса.

```
alter session set events 'trace[rdbms.SQL_Optimizer.*]';
alter session set events '10053 trace name context forever, level 1';
```

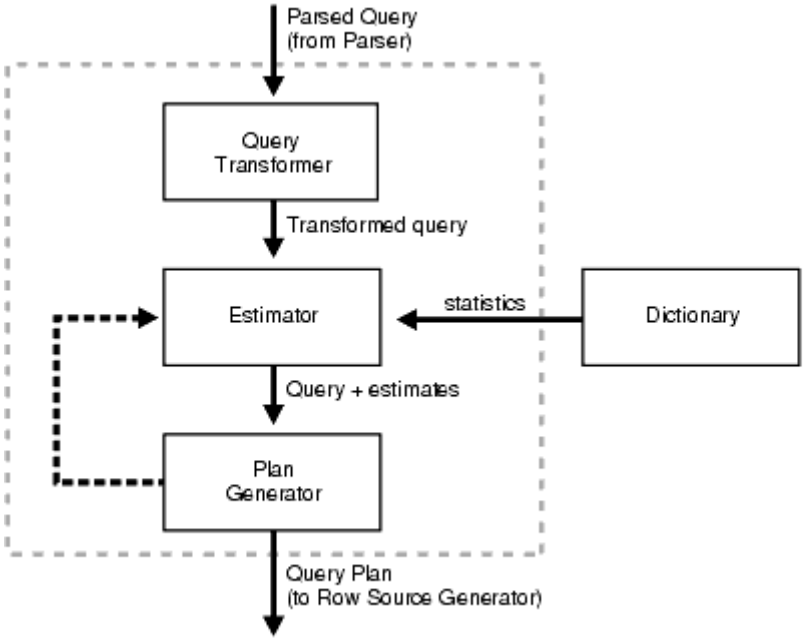
Итоговый запрос будет в файле трассировки в блоке «Final query after transformations:******* UNPARSED QUERY IS *******». Для всех четырех запросов выше он выглядит следующим образом (имя схемы с таблицей было вручную убрано из запроса):

```
SELECT "T1"."ID" "ID","T1"."NAME" "NAME" FROM "T2", "T1" WHERE
"T1"."ID"="T2"."ID"
```

Как видно запрос возвращает совершенно иной результат, нежели требуется. Это потому что в файле трассировки нет специального обозначения для предиката ANTI JOIN (тем не менее, такое обозначение существует в реляционной алгебре), так что следует быть очень внимательным при выполнении итоговых запросов из файла трассировки. Аналогичное замечание применимо и к semi join. Даже без anti/semi join если построить план для итогового запроса, он может отличаться от плана для исходного запроса (один из примеров будет далее при описании работы connect by + where).

В Oracle 12c была добавлена процедура dbms_utility.expand_sql_text (начиная с версии 11.2.0.3 она называлась dbms_sql2.expand_sql_text и была не документирована), которая позволяет смотреть итоговый текст запроса, но он может отличаться от того, что показан в трассировке оптимизатора, так что я бы рекомендовал пользоваться трассировкой.

Трансформации являются частью оптимизатора. Общая схема оптимизации запроса выглядит следующим образом



Трансформации еще называют логической оптимизацией, а генерацию планов (перебор различных методов соединения, методов доступа к таблицам, порядка таблиц) – физической оптимизацией. Дополнительные детали могут быть найдены в [2], [3].

Трансформации в свою очередь делятся на (дополнительные детали в документации и [4])

- Cost-based transformations (основанные на стоимости), например, or-expansion
- Heuristic-based transformations (основанные на эвристиках), например, simple/complex view merging

Первые выполняются только если стоимость трансформированного запроса меньше стоимости исходного, тогда как вторые применяются всегда. Для того чтоб применялись трансформации запросов должен использоваться стоимостной оптимизатор – это поведение по умолчанию.

Для демонстрации or-expansion можно привести следующий пример

```
create table tr(id primary key, name) as
select rownum, lpad('#',rownum,'#') from dual connect by level <= 1e5;

Table created.

explain plan for select * from tr where id = nvl(:p, id);

Explained.

select * from table(dbms_xplan.display(format => 'basic predicate'));
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 2631158932

-----------------------------------------------------
| Id  | Operation                     | Name         |
-----------------------------------------------------
|   0 | SELECT STATEMENT              |              |
|   1 |  CONCATENATION                |              |
|*  2 |   FILTER                      |              |
|*  3 |    TABLE ACCESS FULL          | TR           |
|*  4 |   FILTER                      |              |
|   5 |    TABLE ACCESS BY INDEX ROWID| TR           |
|*  6 |     INDEX UNIQUE SCAN         | SYS_C0011581 |
-----------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(:P IS NULL)
   3 - filter("ID" IS NOT NULL)
   4 - filter(:P IS NOT NULL)
   6 - access("ID"=:P)
```

Если значение связываемой переменной null, то будет выполнено полное сканирование таблицы иначе будет доступ по индексу.

Семантически запрос эквивалентен следующему

```
select *
  from tr
 where id is not null
   and :p is null
union all
select *
  from tr
 where id = :p
   and :p is not null
```

Итоговый запрос после трансформации выглядит следующим образом

```
SELECT "TR"."ID" "ID","TR"."NAME" "NAME" FROM "TR" WHERE
"TR"."ID"=NVL(:B1,"TR"."ID")
```

То есть по его виду невозможно сказать, была ли применена трансформация или нет, но это видно в плане. Как уже было сказано, при других данных такая трансформация могла бы не быть применена – например, если индекс неуникальный и не селективный. Если есть необходимость скорректировать решение оптимизатора касательно этой трансформации, то это достигается хинтами use_concat/no_expand.

Пример ниже демонстрирует view merging (слияния представлений).

В Oracle 12c для воспроизведения надо отключить адаптивные планы с помощью команды «`alter session set optimizer_adaptive_reporting_only = true;`».

```
explain plan for
select name, cnt
  from t3
  join (select id, max(name) name, count(*) cnt from tr group by id) sub
    on sub.id = t3.id;

Explained.
```

```
select * from table(dbms_xplan.display(format => 'basic predicate'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 1900897066

-----------------------------------------------------
| Id  | Operation                     | Name         |
-----------------------------------------------------
|   0 | SELECT STATEMENT              |              |
|   1 |  HASH GROUP BY                |              |
|   2 |   NESTED LOOPS                |              |
|   3 |    NESTED LOOPS               |              |
|   4 |     TABLE ACCESS FULL         | T3           |
|*  5 |     INDEX UNIQUE SCAN         | SYS_C0011582 |
|   6 |    TABLE ACCESS BY INDEX ROWID| TR           |
-----------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("ID"="T3"."ID")

18 rows selected.
```

Как видно из плана, сначала полностью сканируется таблица t3, потом по индексу выбираются нужные строки из tr и последним шагом выполняется группировка. Итоговый запрос выглядит следующим образом

```
SELECT MAX("TR"."NAME") "NAME",COUNT(*) "CNT" FROM "T3", "TR" WHERE
"TR"."ID"="T3"."ID" GROUP BY "TR"."ID","T3".ROWID
```

View merging регулируется хинтами merge/no_merge однако если эту трансформацию отключить для запроса выше, то будет применена другая – filter push down. Для полного отключения всех трансформаций надо использовать хинт no_query_transformation.

Запрос

```
select --+ no_query_transformation
 name, cnt
  from t3
  join (select id, max(name) name, count(*) cnt from tr group by id) sub
    on sub.id = t3.id;
```

После преобразований выглядит как

```
select "from$_subquery$_004"."NAME_0" "NAME",
       "from$_subquery$_004"."CNT_1"  "CNT"
  from (select "SUB"."NAME" "NAME_0", "SUB"."CNT" "CNT_1"
          from "T3",
               (select "TR"."ID" "ID",
                       max("TR"."NAME") "NAME",
                       count(*) "CNT"
                  from "TR"
                 group by "TR"."ID") "SUB"
         where "SUB"."ID" = "T3"."ID") "from$_subquery$_004"
```

То есть, несмотря на то, что никакие трансформации не были применены, ANSI синтаксис в любом случае преобразуется в Oracle native syntax. Такой запрос выполняется более чем в 100 раз дольше чем трансформированный.

Как уже было замечено, в редких случаях трансформации (как стоимостные, так и эвристические) могут приводить к ухудшению производительности, но их можно отключать с помощью хинтов.

Трансформации запросов это не панацея и в некоторых случаях приходится прибегать к ручному переписыванию запроса.

Предположим, что есть таблицы fact_ и dim_, между которыми не установлены ссылочные ограничения целостности. Стоит задача определить факты, для которых отсутствуют необходимые строки в измерении.

```
create table fact_ as
select rownum value, rownum - 1 dim_1_id, rownum dim_2_id from dual connect
by rownum <= 1e6;
create table dim_ as
select rownum id, 'name'||rownum name from dual connect by rownum <= 1e6;
```

Если это делать для каждой из координат отдельно, то запрос преобразовывается к HASH JOIN ANTI и выполняется достаточно быстро.

```
select * from fact_ f where dim_1_id not in (select id from dim_);
select * from fact_ f where dim_2_id not in (select id from dim_);
```

Однако если попытаться получить результат для обеих координат следующим запросом

```
select *
  from fact_ f
 where dim_1_id not in (select id from dim_)
    or dim_2_id not in (select id from dim_);
```

То преобразование не сможет произойти и для каждого факта будет полное сканирование обоих измерений для проверки двух фильтров. Выполнение такого запроса будет длиться практически бесконечно.

```
explain plan for
select *
  from fact_ f
 where dim_1_id not in (select id from dim_)
    or dim_2_id not in (select id from dim_);

Explained.

select * from table(dbms_xplan.display(format => 'basic predicate'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------------
Plan hash value: 481481104

-----------------------------------
| Id  | Operation          | Name  |
-----------------------------------
|   0 | SELECT STATEMENT   |       |
|*  1 |  FILTER            |       |
|   2 |   TABLE ACCESS FULL| FACT_ |
|*  3 |   TABLE ACCESS FULL| DIM_  |
|*  4 |   TABLE ACCESS FULL| DIM_  |
-----------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter( NOT EXISTS (SELECT 0 FROM "DIM_" "DIM_" WHERE
             LNNVL("ID"<>:B1)) OR  NOT EXISTS (SELECT 0 FROM "DIM_" "DIM_" WHERE
             LNNVL("ID"<>:B2)))
   3 - filter(LNNVL("ID"<>:B1))
   4 - filter(LNNVL("ID"<>:B1))

20 rows selected.
```

Разнообразие и возможности трансформаций постоянно расширяются, так в последних релизах 11g было добавлено преобразование scalar subquery unnesting, что может в корне изменить производительность некоторых запросов. Однако реализовать все возможные переписывания в трансформациях практически невозможно.

Помимо трансформаций стоимостного оптимизатора, выполняются и другие важные преобразования. Начиная от ANSI -> native conversion и заканчивая column projection. Важные моменты, которые хотелось бы отметить

- Трансформации влияют на план запроса и соответственно скорость выполнения, но могут быть не отражены в итоговом запросе. Для детального анализа, какие трансформации были применены, необходимо анализировать раздел «Query transformations (QT)» трассировки оптимизатора.
- Следует различать трансформации оптимизатора и иные преобразования запросов, например ANSI syntax -> Oracle native syntax. Последнее выполняется всегда независимо от того использован оптимизатор по стоимости или оптимизатор по правилам (cost based optimizer, rule based optimizer) и от того отключены ли все трансформации хинтом или нет.

  ANSI синтаксис может оставаться в итоговом запросе, если в изначальном используется
  - full join
  - left/right join partition by
- Отдельно стоит упомянуть автоматическую генерацию предикатов при использовании CBO. Часто для обозначения этого явления используют словосочетания transitive closure (Metalink Doc ID 68979.1). На практике это означает, что если в условии вида «`where t1.id = t2.id and t1.id = 1 and t2.id = 1`» исключить второе или третье равенство, то оно будет сгенерировано автоматически.
- Другим важным преобразованием является column projection. Projection – одна из пяти базовых операций реляционной алгебры: selection, projection, union, difference, join. Неплохие вводные статьи по реляционной алгебре можно найти у Iggy Fernandez: SQL Sucks [5], Explaining the EXPLAIN PLAN [6].

  Используя таблицу из первой главы выполним такой запрос

  ```
  with t_ as (select id, id, name from t)
  select name from t_;
  ```

  Он корректно отработает, поскольку по факту будет преобразован в (после projection остался только столбец name)

  ```
  SELECT "T"."NAME" "NAME" FROM "T" "T"
  ```

В Oracle, так же как и в большинстве популярных СУБД, реализованы эвристики такие как:

- выполнять projection «исключение ненужных колонок» как можно раньше
- выполнять selection «исключение ненужных строк» как можно раньше

С другой стороны Oracle не даст создать представление с запросом из subquery factoring выше по очевидным причинам.

- Итоговый запрос в трассировке оптимизатора это всего лишь приблизительная текстовая запись того, что в итоге выполняется. В некоторых случаях, если построить план для итогового запроса и изначального они могут отличаться. Более того, иногда итоговый запрос может возвращать иной результат, потому, что в нем нет специальных обозначений, в частности, для anti/semi соединений.
- Трансформации запросов следует учитывать при использовании хинтов, поскольку после применения трансформаций некоторые планы могут стать невозможными, то есть хинт не будет использован.

Резюме

Трансформации запросов дают Oracle огромную гибкость при построении плана выполнения, они позволяют двум совершенно разным с виду запросам, но семантически эквивалентным, иметь один и тот же план. Также они дают возможность не беспокоится о порядке выполнения операций или даже избегать дублирования кода. Так, например, при соединении двух таблиц с группировкой, Oracle сам решит, что делать сначала - соединение или группировку, когда это возможно; трансформация concatenation позволяет избегать дублирования кода через union all, transitive closure позволяет избегать «лишних» предикатов. Тем не менее, не всегда различные по внешнему виду, но эквивалентные по получаемому результату запросы могут быть одинаково эффективны. Так что разработчикам всегда стоит следовать best practices при написании запросов и учитывать особенности данных. Я не думаю, что трансформации запросов когда-либо достигнут такой функциональности, что полностью отпадет необходимость в ручном переписывании неэффективных запросов.

Кроме трансформация запросов, которые являются частью стоимостного оптимизатора, к любому запросу применяются и другие преобразования, начиная от ANSI -> native conversion и заканчивая column projection.

3. Analytic Functions

Базовые возможности SQL позволяют оперировать над набором данных на уровне строк. При добавлении агрегатных (групповых функций) появляется возможность анализа на уровне групп. При этом каждая строка попадает строго в одну конкретную группу согласно выражению группировки.

Аналитические функции позволяют производить анализ для каждой строки в рамках окна над набором данных. Окно определяет диапазон строк, на основании которого производить вычисления для текущей строки. Эти функции выполняются после всех других операций таких как (joins, where, group by, having) но перед order by и поэтому могут фигурировать только в списке выборки либо в конструкции order by.

Принцип работы легче продемонстрировать на следующем примере

```
with t as
 (select rownum id, trunc(rownum / 4) part from dual connect by rownum <= 6)
select t.*,
       sum(id) over(partition by part order by id) sum1,
       sum(id) over(partition by part) sum2
  from t
 order by id;
```

```
        ID       PART       SUM1       SUM2
---------- ---------- ---------- ----------
         1          0          1          6
         2          0          3          6
         3          0          6          6
         4          1          4         15
         5          1          9         15
         6          1         15         15

6 rows selected.
```

Конструкция «partition by» определяет секции, в раках которых применяется аналитическое окно. При отсутствии конструкции «order by» окно для каждой строки секции начинается с первой строки и до последней, соответственно и значение результата для всех строк одно и то же. Если «order by» указано, то окно определяется от первой строки секции и до текущей, поэтому в примере выше таким образом получена нарастающая сумма.

Некоторые функции требуют обязательного наличия «order by», например, row_number или rank.

То, что делается с аналитическими функциями за один проход таблицы, иначе бы потребовало дополнительных коррелированных подзапросов или соединений. Пример выше можно переписать таким образом.

```
with t as
 (select rownum id, trunc(rownum / 4) part from dual connect by rownum <= 6)
select t.*,
       (select sum(id) from t t0 where t0.part = t.part and t0.id <= t.id)
sum1,
       (select sum(id) from t t0 where t0.part = t.part) sum2
  from t
 order by id;
```

```
        ID       PART       SUM1       SUM2
---------- ---------- ---------- ----------
         1          0          1          6
         2          0          3          6
         3          0          6          6
         4          1          4         15
         5          1          9         15
         6          1         15         15

      6 rows selected.
```

Возвращаясь к трансформациям запросов, Oracle не может преобразовать запрос выше со скалярными подзапросами в запрос с использованием аналитических функций для того чтоб уйти от дополнительных сканирований таблицы и дополнительных соединений. Я думаю, если это когда-то и станет возможным, то очень нескоро.

Аналитические функции позволяют избежать дополнительных соединений/коррелированных подзапросов, даже если в условиях соединения фигурируют разные атрибуты. В примере ниже одна и та же логика реализована в коррелированном подзапросе и с помощью аналитических функций.

```
exec dbms_random.seed(99);

create table ta as
select rownum id,
       trunc(dbms_random.value(1, 5 + 1)) x1,
       trunc(dbms_random.value(1, 5 + 1)) x2,
       trunc(dbms_random.value(1, 5 + 1)) x3
  from dual
connect by level <= 10;

select (select sum(x3) from ta t0 where t0.x2 = ta.x1) s,
       case
         when x1 > x2 then
          sum(x3) over(order by x2 range between greatest(x1 - x2, 0)
                       following and greatest(x1 - x2, 0) following)
         else
          sum(x3) over(order by x2 range between greatest(x2 - x1, 0)
                       preceding and greatest(x2 - x1, 0) preceding)
       end sa,
       ta.*
  from ta
 order by id;
```

```
         S          SA         ID         X1         X2         X3
---------- ---------- ---------- ---------- ---------- ----------
         4          4          1          3          1          2
        10         10          2          1          5          4
         1          1          3          2          5          1
         9          9          4          5          3          4
         4          4          5          3          1          1
                               6          4          5          1
                               7          4          5          3
         4          4          8          3          1          5
         9          9          9          5          2          1
         9          9         10          5          1          2

10 rows selected.
```

Для того, чтобы посчитать необходимую сумму по x2, которые равны x1, используется окно со смещением по диапазону (range) равным разности этих двух атрибутов для каждой строки. В зависимости от того меньше x2 чем x1 или больше рассматриваются строки в окне со сдвигом вперед или назад соответственно. Для каждой строки нас интересует только одна из сумм, посчитанных в case, но Oracle должен подсчитать обе суммы для каждой строки в любом случае. Для того, чтобы определение окна в любом случае было корректным применена функция greatest.

Кроме смещения по диапазону можно также указывать смещение по строкам. Для того чтоб лучше понять разницу в смещении по строкам и по диапазону рассмотрим следующую задачу. Есть таблица с информацией о снятии наличных в банкомате (сумма может быть кратная 5 в диапазоне от 5 до 100 включительно, для простоты снятия происходят каждые две минуты).

```
exec dbms_random.seed(11);

create table atm as
select trunc(sysdate) + (2 * rownum - 1) / (24 * 60) ts,
       trunc(dbms_random.value(1, 20 + 1)) * 5 amount
  from dual
connect by level <= 15;
```

Необходимо на момент каждого снятия определить

- Сколько раз было получено 50 и более денежных единиц за 5 предыдущих снятий и текущее – 6 снятий всего (cnt1)
- Сколько раз было получено 50 и более денежных единиц в отрезок времени от пяти минут до снятия (cnt2)

```
select to_char(ts, 'mi') minute,
       amount,
       count(nullif(sign(amount - 50), -1))
        over(order by ts rows 5 preceding) cnt1,
       count(nullif(sign(amount - 50), -1))
        over(order by ts range interval '5' minute preceding) cnt2
  from atm;
```

MI	AMOUNT	CNT1	CNT2
01	85	1	1
03	15	1	1
05	100	2	2
07	40	2	1
09	30	2	1
11	50	3	1
13	85	3	2
15	60	4	3
17	5	3	2
19	100	4	2
21	25	4	1
23	30	3	1
25	80	3	1
27	5	2	1
29	35	2	1

```
15 rows selected.
```

Более простой пример на тему смещений ниже.

```
with t as
(select rownum id, column_value value from
table(sys.odcinumberlist(1,2,3,4.5,4.6,7,10)))
select t.*,
       last_value(value) over(order by value range between unbounded
preceding and 1 preceding) l1,
       last_value(value) over(order by value rows between unbounded preceding
and 1 preceding) l2
  from t;

        ID      VALUE         L1         L2
---------- ---------- ---------- ----------
         1          1
         2          2          1          1
         3          3          2          2
         4        4,5          3          3
         5        4,6          3        4,5
         6          7        4,6        4,6
         7         10          7          7

7 rows selected.
```

L1 и L2 отличаются для id равного 5, потому что верхняя граница для поиска L1 – 3.6 на единицу меньше текущего, а для L2 – 4.5, то есть предыдущее по порядку значение.

Детальное указание окна имеет смысл не для всех функций. Соответственно, для некоторых это недоступно (поскольку не имеет смысла), например lag/lead.

Из ограничений аналитических функций стоит выделить следующие:

1) При сортировке по нескольким значениям, указание диапазона возможно только с границами в начале окна, конце окна и текущей строке. Например, если есть таблица с координатами (x, y), то невозможно посчитать с помощью аналитической функции за один проход таблицы, сколько точек находятся пределах заданного расстояния от текущей. Тогда как, если координата одна, то это делается элементарно.
2) В самой функции нельзя ссылаться на значения атрибутов в текущей строке. То есть, если, например, необходимо подсчитать для каждой точки сумму расстояний от нее до всех остальных точек, то это опять же сделать невозможно. Более того, это невозможно сделать даже если координата одна. С другой стороны, элементарно можно подсчитать расстояние от всех дочек до некой константной точки независимо от числа координат.

Эти особенности с краткими комментариями продемонстрированы в примере ниже

```
with points as
 (select rownum id, rownum * rownum x, mod(rownum, 3) y
    from dual
  connect by rownum <= 6)
, t as
(select p.*,
       -- число точек на расстоянии 5 от текущей
       -- в случае рассмотрения двух координат не решается
       count(*) over(order by x range between 5 preceding and 5 following) cnt,
       -- сумма расстояний от всех точек _до текущей строки включительно_ до точки (3, 3)
       -- в случае расстоянии до текущей строки а не константы не решается
       round(sum(sqrt((x - 3) * (x - 3) + (y - 3) * (y - 3)))
             over(order by id),
             2) dist
  from points p)
select t.*,
(select count(*)
   from t t0 where t0.x between t.x-5 and t.x + 5) cnt1,
(select count(*)
   from t t0 where t0.x between t.x-5 and t.x + 5 and t0.y between t.y-1 and t.y + 1) cnt2,
(select round(sum(sqrt((x - 3) * (x - 3) + (y - 3) * (y - 3))), 2)
   from t t0 where t0.id <= t.id) dist1,
(select round(sum(sqrt((x - t.x) * (x - t.x) + (y - t.y) * (y - t.y))), 2)
   from t t0 where t0.id <= t.id) dist2
from t
 order by id;
```

ID	X	Y	CNT	DIST	CNT1	CNT2	DIST1	DIST2
1	1	1	2	2.83	2	2	2.83	0
2	4	2	3	4.24	3	2	4.24	3.16
3	9	0	2	10.95	2	1	10.95	13.45
4	16	1	1	24.1	1	1	24.1	34.11
5	25	2	1	46.13	1	1	46.13	70.2
6	36	0	1	79.26	1	1	79.26	125.28

```
6 rows selected.
```

Значения cnt2 и dist2, полученные с помощью подзапросов, невозможно получить с помощью аналитических функций.

Стоит также упомянуть, что если поле сортировки имеет тип, который не поддерживает арифметические операции, то range вообще не может быть использован. Для rows, очевидно, такого ограничения нет.

Большинство аналитических функций могут являться также и агрегатными функциями (при отсутствии over()), но есть и исключения, такие как row_number, rank и другие. Для таких исключений всегда в определении окна необходимо указывать сортировку.

Особняком среди всех аналитических функций стоит listagg. Во-первых, ее особенность в том, что она не коммутативная. То есть, результат конкатенации первой строки со второй отличается от конкатенации второй с первой в отличие от, например, суммы или среднего. Во-вторых, в конструкции over () нельзя указывать сортировку. В-третьих, в самой функции нельзя использовать distinct. Некоторые различия между встроено listagg и пользовательской агрегатной функцией stragg (код есть на сайте asktom) более понятны, если запустить следующий запрос.

```
with t as
 (select rownum id, column_value value
    from table(sys.odcinumberlist(2, 1, 1, 3, 1))),
t0 as
 (select t.*, row_number() over(partition by value order by id) rn from t)
select t1.*,
       (select listagg(value, ',') within group(order by value)
          from t t_in
         where t_in.id <= t1.id) cumul_ord
  from (select t0.*,
               listagg(value, ',') within group(order by value) over() list_ord,
               listagg(decode(rn, 1, value), ',') within group(order by value) over() dist_ord,
               stragg(value) over(order by id) cumul,
               stragg(distinct value) over() dist,
               stragg(decode(rn, 1, value)) over(order by id) cumul_dist
          from t0) t1
 order by id;
```

ID	VALUE	RN	LIST_ORD	DIST_ORD	CUMUL	DIST	CUMUL_DIST	CUMUL_ORD
1	2	1	1,1,1,2,3	1,2,3	2	1,2,3	2	2
2	1	1	1,1,1,2,3	1,2,3	2,1	1,2,3	2,1	1,2
3	1	2	1,1,1,2,3	1,2,3	2,1,1	1,2,3	2,1	1,1,2
4	3	1	1,1,1,2,3	1,2,3	2,1,1,3	1,2,3	2,1,3	1,1,2,3
5	1	3	1,1,1,2,3	1,2,3	2,1,1,3,1	1,2,3	2,1,3	1,1,1,2,3

Если кратко – для listagg невозможно получить кумулятивную конкатенацию с сортировкой в окне, для stragg наоборот – можно указывать сортировку в окне, но невозможно указать сортировку результата.

Если же задача получить кумулятивную конкатенацию с сортировкой в окне, то аналитической функцией этого достичь невозможно. Выше это реализовано с помощью скалярного подзапроса и агрегатной listagg, в качестве альтернативы, можно было бы использовать PL/SQL, recursive subquery factoring, model или формировать необходимый результат с помощью клиентского приложения.

Важно отметить, что аналитические функции – это не всегда панацея. Иногда дополнительное соединение или подзапрос может быть выгоднее. Например, есть таблица stream, в которую записываются порции данных, идентифицируемые по batch_id и есть индекс по этому полю.

```
create table stream as
select batch_id, value
  from (select rownum value from dual connect by rownum <= 10000) x1,
       (select rownum batch_id from dual connect by level <= 1000)
 order by 1, 2;

create index stream_batch_id_idx on stream(batch_id);

exec dbms_stats.gather_table_stats(user, 'stream');
```

Допустим необходимо посчитать сумму всех значений value для последнего batch_id. Сравним производительность для запросов с аналитическими функциями и без.

```
alter session set statistics_level = all;

select sum(s.value)
  from stream s
 where batch_id = (select max(s0.batch_id) from stream s0);
select * from table(dbms_xplan.display_cursor(null,null,'IOSTATS LAST'));
```

```
select sum(value)
  from (select s.*, dense_rank() over(order by batch_id) drnk from stream s)
 where drnk = 1;
select * from table(dbms_xplan.display_cursor(null,null,'IOSTATS LAST'));
```

Планы выполнения приведены ниже, столбцы Name и Starts вырезаны для того, чтоб информация поместилась по ширине. В случае с подзапросом время выполнения в 35 раз меньше: 0.09 секунды против 3.48. Во-первых, более чем в 400 раз отличается число логических и физических чтений. Во-вторых, в случае с аналитикой, основное время ушло на сортировку – 3.47 – 1.40 = 2.07 секунды.

```
-------------------------------------------------------------------------------------
| Id  | Operation                     | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |        |      1 |00:00:00.09 |      43 |     40 |
|   1 |  SORT AGGREGATE               |      1 |      1 |00:00:00.09 |      43 |     40 |
|   2 |   TABLE ACCESS BY INDEX ROWID |  10000 |  10000 |00:00:00.09 |      43 |     40 |
|*  3 |    INDEX RANGE SCAN           |  10000 |  10000 |00:00:00.06 |      25 |     22 |
|   4 |     SORT AGGREGATE            |      1 |      1 |00:00:00.05 |       3 |      3 |
|   5 |      INDEX FULL SCAN (MIN/MAX)|      1 |      1 |00:00:00.05 |       3 |      3 |
-------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------
| Id  | Operation                | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |        |      1 |00:00:03.48 |   17823 |  17820 |
|   1 |  SORT AGGREGATE          |      1 |      1 |00:00:03.48 |   17823 |  17820 |
|*  2 |   VIEW                   |    10M|  10000 |00:00:03.47 |   17823 |  17820 |
|*  3 |    WINDOW SORT PUSHED RANK|    10M|  10001 |00:00:03.47 |   17823 |  17820 |
|   4 |     TABLE ACCESS FULL    |    10M|    10M|00:00:01.40 |   17823 |  17820 |
-------------------------------------------------------------------------------------
```

Встречаются запросы, в которых сортировка, порождаемая аналитикой, настолько неэффективна, что даже без каких-либо индексов два прохода таблицы может быть эффективнее, но такие случае тоже довольно редки.

Differences and interchangeability of functions

В этом разделе не будет объяснения отличий, скажем, row_number от rank или rank от в dense_rank. Вместо этого речь пойдет о том, когда разные функции могут (или не могут) быть использованы для одних и тех же целей, учитывая специфику указания окна и самих функций.

Иногда можно встретить такой код

```
max(version) over (partition by dt order by version
rows between unbounded preceding and unbounded following) latest_version
```

В этом случае указание сортировки не имеет ни малейшего смысла, поскольку указан диапазон по всей секции (partition). С другой стороны, если сортировку убрать, то окно по умолчанию будет от первой до последней строки секции. То есть абсолютно идентичное с точки зрения логики выражение выглядит следующим образом

```
max(version) over (partition by dt) latest_version
```

Однако для некоторых других функций имеет смысл указывать сортировку даже если окно для всех строк одинаковое и совпадает с размером секции.

Рассмотрим следующую задачу: для последовательности значений отобразить максимальное значение, соответствующее максимальной дате.

Это реализовано в выражениях для m2 и m3 в примере ниже

```
with t(id, value, dt, part) as
(
select 1, 10, date '2015-07-01', 1 from dual
union all select 2, 3, date '2015-08-01', 1 from dual
union all select 3, 2, date '2015-09-01', 1 from dual
union all select 4, 0, date '2016-11-01', 1 from dual
union all select 5, 5, date '2016-11-01', 1 from dual
union all select 6, 9, date '2017-01-01', 1 from dual
union all select 7, 4, date '2017-01-01', 1 from dual
)
select
 t.*,
 max(value) over (partition by part) m1,
 max(value) keep (dense_rank last order by dt) over (partition by part) m2,
 last_value(value)
 over (partition by part order by dt, value
      rows between unbounded preceding and unbounded following) m3,
 max(value)
 over (partition by part order by dt, value
      rows between unbounded preceding and unbounded following) m4
 from t
order by id;
```

ID	VALUE	DT	PART	M1	M2	M3	M4
1	10	01.07.15	1	10	9	9	10
2	3	01.08.15	1	10	9	9	10
3	2	01.09.15	1	10	9	9	10
4	0	01.11.16	1	10	9	9	10
5	5	01.11.16	1	10	9	9	10
6	9	01.01.17	1	10	9	9	10
7	4	01.01.17	1	10	9	9	10

```
7 rows selected.
```

То есть, максимальная дата – 01.01.2017 и для нее есть два значения 4 и 9. Это может быть получено функцией max(value) с сортировкой по дате в разделе keep, либо функцией last_value, но в сортировке должна быть и дата и value.

Выражения m1 и m4 идентичны по смыслу, но дают неверный результат, поскольку не учитываются сортировку по дате. Для функции max играет роль только границы окна, тогда как для функции last_value и границы и сортировка в окне.

Если бы было необходимо получить минимальное значение для максимальной даты, это можно было бы сделать с помощью следующий выражений

```
 min(value) keep (dense_rank last order by dt) over (partition by part) m2,
 last_value(value)
 over (partition by part order by dt, value desc
      rows between unbounded preceding and unbounded following) m3
 from t
```

В первом случае изменилась аналитическая функция, во втором - направление сортировки по полю value.

Последний пример будет касательно last value и конструкции ignore nulls. До версии 10g нельзя было указывать ignore nulls, но это достаточно легко можно было обойти, используя count и max вместо last_value.

```
with t(id, value, part) as
(
select 1, null, 1 from dual
union all select 2, 'one', 1 from dual
union all select 3, null, 1 from dual
union all select 1, 'two', 2 from dual
union all select 2, null, 2 from dual
union all select 3, null, 2 from dual
union all select 4, 'three', 2 from dual
)
select t.*, max(value) over(partition by part, cnt) lv0
  from (select t.*,
               last_value(value ignore nulls) over(partition by part order by
id) lv,
               count(value) over(partition by part order by id) cnt
          from t
         order by part, id) t;

        ID VALUE       PART LV           CNT LV0
---------- ----- ---------- ----- ---------- -----
         1                  1                0
         2 one              1 one            1 one
         3                  1 one            1 one
         1 two              2 two            1 two
         2                  2 two            1 two
         3                  2 two            1 two
         4 three            2 three          2 three

7 rows selected.
```

Казалось бы функции с совершенно разным предназначением могут быть использованы для одних и тех же целей.

Резюме

Аналитические функции – очень мощный инструмент, позволяющий получить результат, который в противном случае потребовался бы self join или коррелированные подзапросы. Впервые они были представлены в версии 8i, но небольшие усовершенствования выпускаются в каждой следующей версии, включая 12c. Oracle предоставляет возможности изменения стандартного определения окна с помощью ключевых слов range/rows between, эти возможности тоже имеют свои ограничения, но для большинства «типичных» задач аналитики существующих средств вполне достаточно.

4. Aggregate functions

Агрегатные функции, в отличие от аналитических, возвращают одну строку для каждой группы. Группа описывается в конструкции group by, где могут быть перечислены как просто имена столбцов, так и выражения. Строки, для которых совпадают значения всех выражений перечисленных в group by, принадлежат к одной группе. Каждая строка из набора данных, для которого применяется группировка, может принадлежать к одной и только одной группе. Если выражение group by не указано, то весь набор данных представляет собой одну группу. Конструкции group by cube/rollup/grouping sets могут быть использованы для получения различных промежуточных итогов по группам.

Например, используя таблицу с презентациями для первой главы можно подсчитать для каждого из авторов общее число презентаций и число рабочих дней.

```
select p.name,
       count(*) cnt_all,
       count(distinct p.day) cnt_day,
       listagg(p.day || ' ' || p.time || ':00', '; ') within group(order by w.id) details
  from presentation p, week w
 where p.day = w.day
 group by p.name;

NAME    CNT_ALL    CNT_DAY DETAILS
---- ---------- ---------- ----------------------------------------
John          3          2 monday 14:00; monday 9:00; friday 9:00
Rex           2          2 wednesday 11:00; friday 11:00
```

Для того, чтоб на вход агрегатной функции подавались только уникальные значения из группы использовано ключевое слово distinct. Для конкатенации строк была использована функции listagg. Как и в случае с аналитическими функциями, listagg отличается от всех остальных функций, дополнительные детали по этому поводу могут быть найдены в предыдущей главе. Кроме listagg, для групповых функций есть еще две некоммутативные функции collect и xmlagg, поскольку сортировка в группе влияет на результат, то ее можно указывать при использовании функции.

Групповые функции могут быть вложенными, а также использоваться совместно с аналитическими, детально эти моменты рассматриваются в девятой главе «Logical execution order of query clauses».

Еще среди агрегатных функций стоит выделить collect. Она возвращает коллекцию, для обработки которой можно написать собственную пользовательскую функцию. Например, вместо встроенной функции склейки строк можно было бы использовать следующую UDF.

```
create or replace function collagg(p in strings) return varchar is
  result varchar2(4000);
begin
  for i in 1 .. p.count loop
    result := result || ', ' || p(i);
  end loop;
  return(substr(result, 2));
end collagg;
```

Где тип strings определен следующим образом

```
create or replace type strings as table of varchar2(4000)
/
```

Получить список всех дней и список уникальных дней для каждого автора можно так

```
select name,
       collagg(cast(collect(day) as strings)) days,
       collagg(set(cast(collect(day) as strings))) days_unique
  from presentation
 group by name;

NAME DAYS                           DAYS_UNIQUE
---- ------------------------------ --------------------
John  monday, friday, monday         monday, friday
Rex   wednesday, friday              wednesday, friday
```

Обратите внимание, что сортировка элементов списка не определена, но этом можно исправить, изменив реализацию collagg, либо указав сортировку в collect.

Последнее что стоит добавить касательно общих моментов – Oracle (с версии 9i Release 1) предоставляет интерфейс для реализации пользовательских агрегатных функций (UDAG – user defined aggregate functions), так что если среди встроенных функций нет той, которая вам необходимо, то не составит труда реализовать ее самостоятельно. Например, есть функция для битового «И» - bitand, но нет аналогичной функции для группы строк. При необходимости использовать такую функцию при группировке можно либо реализовать собственную UDAG либо collect + UDF.

Агрегатные функции, так же как и аналитические, могут в некоторых случаях позволять избавиться от лишних соединений и сканирований таблиц.

Например, есть таблица с описанием позиций и требуемых знаний. Необходимо выбрать позиции, для которых требуется Oracle и не требуется Linux.

```
create table requirements(position, skill) as
(
select 'Data Scientist', 'R' from dual
union all select 'Data Scientist', 'Python' from dual
union all select 'Data Scientist', 'Spark' from dual
union all select 'DB Developer', 'Oracle' from dual
union all select 'DB Developer', 'Linux' from dual
union all select 'BI Developer', 'Oracle' from dual
union all select 'BI Developer', 'MSSQL' from dual
union all select 'BI Developer', 'Analysis Services' from dual
union all select 'System Administrator', 'Linux' from dual
union all select 'System Administrator', 'Network Protocols' from dual
union all select 'System Administrator', 'Python' from dual
union all select 'System Administrator', 'Perl' from dual
);
```

Решение «в лоб» могло бы выглядеть следующим образом

```
select position
  from requirements r
 where skill = 'Oracle'
```

```
   and not exists (select null
         from requirements r0
        where r0.position = r.position
          and r0.skill = 'Linux');

POSITION
--------------------
BI Developer
```

Его главный минус – это коррелированный подзапрос в конструкции where, который приводит к дополнительному сканированию таблицы requirements.

Можно подойти к решению с другой стороны и, подсчитав число требований Linux и Oracle для каждой позиции, отфильтровать нужные.

```
select position
  from requirements
 group by position
having count(decode(skill, 'Oracle', 1)) = 1 and count(decode(skill, 'Linux',
1)) = 0;

POSITION
--------------------
BI Developer
```

Такое решение значительно более предпочтительное с точки зрения производительности. Можно обратить внимание, что значения агрегатов не попадают в итоговую выборку, а используются лишь для фильтрации.

Рассмотрим более общий пример, пусть имеется EAV design, то есть предусматривается хранение сущностей с произвольным числом атрибутов в одной таблице, а в другой таблице построчно хранятся значения всех атрибутов.

```
with entity(id, name) as
(select 1, 'E1' from dual
union all select 2, 'E2' from dual
union all select 3, 'E3' from dual),
property(id, entity_id, name, value) as
(select 1, 1, 'P1', 1 from dual
union all select 2, 1, 'P2', 10 from dual
union all select 3, 1, 'P3', 20 from dual
union all select 4, 1, 'P4', 50 from dual
union all select 5, 2, 'P1', 1 from dual
union all select 6, 2, 'P3', 100 from dual
union all select 7, 2, 'P4', 50 from dual
union all select 8, 3, 'P1', 1 from dual
union all select 19, 3, 'P2', 10 from dual
union all select 10, 3, 'P3', 100 from dual)
```

Пусть наша цель выбрать имена сущностей, для которых значения атрибутов P1, P2, P3 – 1, 10, 100 соответственно. Иногда для таких целей выполняют соединение с таблицей property столько раз сколько атрибутов интересует, что является крайне неэффективным.

```
select e.name
  from entity e
  join property p1 on p1.entity_id = e.id and p1.name = 'P1'
  join property p2 on p2.entity_id = e.id and p2.name = 'P2'
  join property p3 on p3.entity_id = e.id and p3.name = 'P3'
 where p1.value = 1 and p2.value = 10 and p3.value = 100;

NAME
-----
E3
```

При необходимости получить значения 20-ти атрибутов – понадобилось бы 20 соединений с property, что на больших объемах данных могло бы быть нежизнеспособно.

Если принять во внимание, что для каждого свойства может быть либо одно значение либо свойство отсутствует, то значения всех интересующих свойств можно получить в плоском виде и потом отфильтровать.

```
select name
  from (select e.name,
               max(decode(p.name, 'P1', value)) p1_value,
               max(decode(p.name, 'P2', value)) p2_value,
               max(decode(p.name, 'P3', value)) p3_value
          from entity e
          join property p
            on p.entity_id = e.id
         group by e.name)
 where (p1_value, p2_value, p3_value) in ((1, 10, 100));
```

Ну и последнее что можно упростить – если нас интересует только имя сущности без значений каких-либо свойств, то необходимый результат можно получить с помощью следующей группировки и применения having.

```
select e.name
  from property p
  join entity e on p.entity_id = e.id
 where (p.name, p.value) in (('P1', 1), ('P2', 10), ('P3', 100))
 group by e.name
having count(*) = 3
```

Резюме

Групповые функции позволяют «схлопнуть» набор строк так, что для каждой группы будет одна строка в результате, и при этом подсчитать значения необходимых агрегатов. Порядок строк для групповых функций не важен за исключением некоммутативных listagg/collect/xmlagg, для которых можно указывать сортировку в группе. Полный контроль при обработке набора строк можно достичь, если использовать групповую функцию collect с последующей обработкой полученной коллекции. Oracle предоставляет интерфейс для создания пользовательских агрегатных функций, которые могут быть и аналитическими при использовании over (). В некоторых случаях групповые функции позволяют избежать лишних соединений или подзапросов, но подобные задачи весьма специфичны. Важным элементом Oracle-реализации языка SQL является ключевое слово keep, которое позволяет обратиться к первым (или последним) значениям группы согласно сортировке, и может быть использовано как для групповых, так и для аналитических функции.

5. Hierarchical queries: connect by

Конструкция connect by применяется для построения иерархии, если она хранится в базе в виде связей parent-child (родитель-потомок). Также можно встретить название такого способа хранения как adjacency lists (списки смежности). Подразумевается, что для каждого родительского узла связь с каждым из его потомков хранится в отдельной строке.

```
with tree as
(
select 2 id, 1 id_parent from dual
union all select 3 id, 1 id_parent from dual
union all select 4 id, 3 id_parent from dual
union all select 5 id, 4 id_parent from dual
union all select 11 id, 10 id_parent from dual
union all select 12 id, 11 id_parent from dual
union all select 13 id, 11 id_parent from dual
)
select connect_by_root id_parent root,
       level lvl,
       rpad(' ', (level - 1) * 3, ' ') || t.id as id,
       prior id_parent grand_parent,
       sys_connect_by_path(id, '->') path,
       connect_by_isleaf is_leaf
  from tree t
 start with t.id_parent in (1, 10)
connect by prior t.id = t.id_parent;
```

ROOT	LVL	ID	GRAND_PARENT	PATH	IS_LEAF
1	1	2		->2	1
1	1	3		->3	0
1	2	4	1	->3->4	0
1	3	5	3	->3->4->5	1
10	1	11		->11	0
10	2	12	10	->11->12	1
10	2	13	10	->11->13	1

Для построения иерархии необходимо указать

- корень – в примере строятся два дерева с корнями 1 и 10
- связь между родителями и потомками. `prior` это унарный оператор, который возвращает значение указанного выражения (как правило, колонки) для родительской записи для каждой строки.

connect_by_root – унарный оператор, который возвращает значение колонки для корневой строки.

level и connect_by_isleaf – псевдостолбцы, которые указывают для каждой строки (узла) глубину дерева и является ли узел листовым.

Sys_connect_by_path – функция, возвращающая путь от корня до текущего узла для заданного столбца.

Оператор prior может использоваться и в select list, но может быть применен к конкретному выражению только единожды. Если есть цель вывести значение узла два уровня назад, то это можно достигнуть, если применить оператор prior к родителю.

При использовании connect by всегда выполняется построение дерева в глубину, таким образом, для каждого узла сначала выводятся дочерние узлы, а потом узлы того же уровня. Сортировать узлы на одном уровне можно с помощью ключевых слов order siblings by. В этом случае, по-прежнему, дочерние узлы будут выведены до узлов того же уровня, но на одном уровне для конкретного родителя порядок дочерних узлов может быть изменен. Для большей ясности, следующий результат будет получен, если в запрос выше добавить «`order siblings by t.id desc`». Подобная сортировка не гарантирует сортировку корневых узлов, поскольку нельзя сказать, что они имеют общего родителя.

```
      ROOT        LVL ID          GRAND_PARENT PATH           IS_LEAF
---------- ---------- ---------- ------------ ---------- ----------
        10          1 11                      ->11                0
        10          2    13                10 ->11->13            1
        10          2    12                10 ->11->12            1
         1          1 3                       ->3                 0
         1          2    4                 1 ->3->4               0
         1          3       5              3 ->3->4->5            1
         1          1 2                       ->2                 1
```

Иерархия может строиться не только по одной таблице (или inline view), но и по соединению таблиц. В таком случае части запроса выполняются в следующем порядке: join, connect by, where.

Создадим следующие таблицы для демонстрации

```
drop table tree;
drop table nodes;
create table tree(id, id_parent) as
select rownum, rownum - 1 from dual connect by level <= 4;
create table nodes(id, name, sign) as
select rownum, 'name' || rownum, decode(rownum, 3, 0, 1)
  from dual connect by rownum <= 4;
```

В первом запросе фильтр по sign сработал после построения иерархии, а во втором и в третьем – до построения.

```
select t.*, n.name
  from tree t, nodes n
 where t.id = n.id
   and n.sign = 1
 start with t.id_parent = 0
connect by prior t.id = t.id_parent;
```

```
        ID  ID_PARENT NAME
---------- ---------- ----------------------------------------------
         1          0 name1
         2          1 name2
         4          3 name4
```

```
select *
  from (select t.*, n.name
          from tree t, nodes n
         where t.id = n.id
           and n.sign = 1) t
 start with t.id_parent = 0
connect by prior t.id = t.id_parent;

        ID  ID_PARENT NAME
---------- ---------- ----------------------------------------
         1          0 name1
         2          1 name2

select t.*, n.name
  from tree t
  join nodes n
    on t.id = n.id
   and n.sign = 1
 start with t.id_parent = 0
connect by prior t.id = t.id_parent;

        ID  ID_PARENT NAME
---------- ---------- ----------------------------------------
         1          0 name1
         2          1 name2
```

В трассировке 10053 итоговый запрос для ANSI join был преобразован в native syntax, но если посмотреть его план (cartesian join) – он будет отличаться от плана исходного запроса. Более логично было ожидать, что в итоговом запросе будет автоматически создано inline view.

```
select "T"."ID" "ID", "T"."ID_PARENT" "ID_PARENT", "N"."NAME" "NAME"
  from "TREE" "T", "NODES" "N"
 start with "T"."ID_PARENT" = 0
        and "T"."ID" = "N"."ID"
        and "N"."SIGN" = 1
connect by "T"."ID_PARENT" = prior "T"."ID"
       and "T"."ID" = "N"."ID"
       and "N"."SIGN" = 1
```

Интересно также выглядит итоговый запрос для первого варианта.

```
select "T"."ID" "ID", "T"."ID_PARENT" "ID_PARENT", "N"."NAME" "NAME"
  from "TREE" "T", "NODES" "N"
 where "N"."SIGN" = 1
 start with "T"."ID_PARENT" = 0
        and "T"."ID" = "N"."ID"
connect by "T"."ID_PARENT" = prior "T"."ID"
       and "T"."ID" = "N"."ID"
```

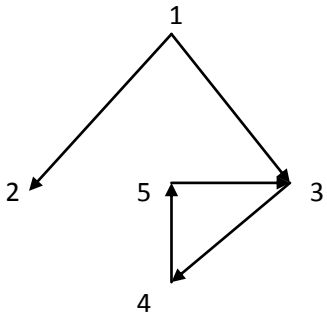
Предикат соединения из where добавлен в конструкции start with и connect by. Еще раз подчеркиваю, что эти планы для этих итоговых запросов отличаются от планов соответствующих исходных запросов и хоть они семантически эквивалентны с исходными, но производительность у них будет значительно хуже ввиду того, что построение иерархии выполняется по декартовому произведению таблиц.

При использовании внешних соединений нет логических различий между native и ANSI синтаксисом, потому что если предикат содержит (+), то он однозначно выполнится до построения иерархии.

Также стоит отметить, что если требуется найти дочерние записи на n-ом уровне, то имеет смысл добавить условие level <= n в connect by в дополнение к фильтру where level = n. В таком случае иерархия не будет строиться далее желаемого уровня. При отсутствии этого условия в connect by результат всегда будет верным, но производительность может быть хуже, поскольку where срабатывает только после построения иерархии.

Еще одним важным моментом является то, что проверка условия connect by выполняется после возврата строк на текущем уровне. То есть если в условие соединения добавить level <= 1 и level <= 0, то в любом случае будут возвращены все строки первого уровня. Строки первого уровня должны удовлетворять условиям start with и where если таковые имеются.

С помощью конструкции connect by можно обрабатывать направленные графы даже если они имеют циклы.




```
with graph (id, id_parent) as
(select 2, 1 from dual
union all select 3, 1 from dual
union all select 4, 3 from dual
union all select 5, 4 from dual
union all select 3, 5 from dual)
select level lvl, graph.*, connect_by_iscycle cycle
  from graph
 start with id_parent = 1
connect by nocycle prior id = id_parent;
```

LVL	ID	ID_PARENT	CYCLE
1	2	1	0
1	3	1	0
2	4	3	0
3	5	4	1

Потомком узла с id = 5 является узел с id = 3, который уже был обработан. Поэтому построение иерархии для замкнувшейся ветки приостанавливается и соответствующему узлу устанавливается признак цикла – его можно вывести с помощью псевдостолбца connect_by_iscycle. Он равен 1 если текущая стока имеет дочернюю, которая в то же время является ее родительской (на одном из обойденных уровней).

Стоит отметить, что конструкция start with может отсутствовать так же как и ссылка на родительскую запись в условии построения. Это часто используется для генерации списков.

Например:

```
select level id from dual connect by level <= 10;
select rownum id from dual connect by rownum <= 10;
select rownum id from (select * from dual connect by 1 = 1)
 where rownum <= 10;
```

В этих запросах нет циклов, потому что нет обращения к родительской записи. Поскольку нет отношения родитель-потомок как такового, то и цикла быть не может. Попробуйте выполнить любой из запросов, добавив в connect by условие «`prior 1 = 1`».

В документации есть фраза «it must use the PRIOR operator to refer to the parent row». То есть для ссылки на родительскую запись должен быть использован оператор prior, но это не означает, что на родительскую запись обязательно надо ссылаться.

Ключевой момент при использовании connect by – это то, что невозможно генерировать дочерние значения на основании родительских. Стоит отметить, что новые строки могут быть сгенерированы (при отсутствии prior как показано выше) и даже новые значения можно генерировать, в том числе, зависящие от текущего уровня, но значение используемое в операторе prior, должно существовать до построения иерархии.

Как будет показано в следующей главе, при использовании recursive subquery factoring подобного ограничения нет.

Чтоб продемонстрировать ограничение рассмотрим такую задачу: необходимо сгенерировать рекурсивную последовательность как в функции ниже.

```
create or replace type integers as table of int;
/
create or replace function f(n in number) return integers as
  result integers := integers();
begin
  result.extend(n + 1);
  result(1) := 1;
  for i in 2 .. n + 1 loop
    result(i) := round(100 * sin(result(i - 1) + i - 2));
  end loop;
  return result;
end;
/
```

Иными словами текущий элемент равен синусу суммы предыдущего с индексом предыдущего (при нумерации с нуля – поэтому в коде `i-2`) помноженной на 100 и округленной до целого.

Поскольку значение sin находится на отрезке [-1; 1] и элементы последовательности принимают только целые значения, то можно сгенерировать список всех возможных значений элементов последовательности – отрезок [-100; 100].

```
with t as
 (select -100 + level - 1 result from dual connect by level <= 201)
select level - 1 as id, result, connect_by_iscycle cycle
  from t
 start with result = 1
connect by nocycle round(100 * sin(prior result + level - 2)) = result
       and level <= 21;
```

ID	RESULT	CYCLE
0	1	0
1	84	0
2	-18	0
3	29	0
4	55	0
5	64	0
6	-11	0
7	96	0
8	62	0
9	77	0
10	-92	0
11	-31	0
12	-91	0
13	44	1

В примере выше сгенерировано 14 элементов вместо 21, потому что в 14-м элементе есть цикл. Чтоб сгенерировать необходимое число элементов необходимо обмануть Oracle, чтоб он не трактовал элементы с одинаковыми значениями result как один и тот же узел. Для этого можно ввести условие `prior sys_guid() is not null`. Значение sys_guid для родительской и дочерней записей всегда будет отличаться, поэтому последовательность будет считаться без циклов.

```
with t as
 (select -100 + level - 1 result from dual connect by level <= 201)
select level - 1 as id, result, connect_by_iscycle cycle
  from t
 start with result = 1
connect by nocycle round(100 * sin(prior result + level - 2)) = result
       and prior sys_guid() is not null
       and level <= 21;
```

ID	RESULT	CYCLE
0	1	0
1	84	0
2	-18	0
3	29	0
4	55	0
5	64	0
6	-11	0
7	96	0
8	62	0
9	77	0
10	-92	0
11	-31	0
12	-91	0
13	44	0
14	44	0
15	99	0
16	78	0
17	-25	0
18	-99	0
19	63	0
20	31	0

Теперь после применения трюка и убедившись, что циклов нет, можно удалить и ключевое слово nocycle.

Подытоживая, стоит отметить следующее:

- Цикл при connect by может быть, только если в условии построения иерархии есть оператор prior.
- Если использовать prior с функцией дающей заведомо разные результаты, то цикл идентифицирован не будет, ибо родительская и дочерняя записи всегда будут считаться различными.

Подобный подход перебора из сгенерированного множества можно применить и для случая, когда рекурсивная формула содержит элементы на двух предыдущих итерациях, но в этом случае необходимо сгенерировать все необходимые пары. Например, для генерации чисел Фибоначчи запрос будет выглядеть так

```
with t as
 (select rownum - 1 id from dual connect by rownum <= power(2, 15) / 15),
pairs as
 (select t1.id id1, t2.id id2
    from t t1, t t2
   where t2.id between (1 / 2) * t1.id and (3 / 4) * t1.id
  union all
  select 1, 0 from dual
  union all
  select 1, 1 from dual)
select rownum lvl, id2 fib
  from pairs
 start with (id1, id2) in ((1, 0))
connect by prior id1 = id2
       and prior (id1 + id2) = id1
       and level <= 15;
```

LVL	FIB
1	0
2	1
3	1
4	2
5	3
6	5
7	8
8	13
9	21
10	34
11	55
12	89
13	144
14	233
15	377

Можно заметить, что i-й элемент меньше числа 2^i/i и предыдущий элемент находится в диапазоне от половины следующего до трех четвертей следующего. Эти факты были использованы для ограничения множества сгенерированных пар. В отличие от предыдущего примера, в условии построения иерархии нет необходимости использовать трюк с sys_guid, потому что последовательность строго возрастающая, а значит, речи о циклах быть не может. Тем не менее, скорость падает очень значительно при увеличении уровня, и в реальных задачах нет никакого смысла использовать подобные переборы. Основная цель вышесказанного исключительно демонстрация возможностей.

Трюк с sys_guid может быть так же использован для генерации строк до нужного количества для каждой исходной строки.

```
with t as
 (select 'A' value, 2 cnt from dual
  union all
  select 'B' value, 3 cnt from dual)
select *
  from t
connect by level <= cnt
       and prior value = value
       and prior sys_guid() is not null;
```

```
V        CNT
- ----------
A          2
A          2
B          3
B          3
B          3
```

Здесь отсутствует условие старта, то есть он выполняется из всех строк. Соединение идет в рамках value – предыдущее всегда равно текущему и для того, чтоб Oracle считал, что циклов нет, использован трюк с sys_guid. Тем не менее, есть множество других способов «размножить» строк и я бы предпочел не использовать трюки без необходимости.

Интересно продемонстрировать описанный примем при обходе графов с циклами. Поскольку он позволяет дать понять Ораклу, что циклов нет, то цикл можно обойти несколько раз.

```
select level lvl, graph.*, connect_by_iscycle cycle
  from graph
 start with id_parent = 1
connect by nocycle prior id = id_parent
       and prior sys_guid() is not null
       and level <= 10;
```

```
       LVL         ID  ID_PARENT      CYCLE
---------- ---------- ---------- ----------
         1          2          1          0
         1          3          1          0
         2          4          3          0
         3          5          4          0
         4          3          5          0
         5          4          3          0
         6          5          4          0
         7          3          5          0
         8          4          3          0
         9          5          4          0
        10          3          5          0
```

Как и следовало ожидать, признак цикла везде равен нулю.

Если стоит задача обойти все ребра, в том числе замыкающее цикл, то можно добавить условие prior id_parent is not null. В этом случае цикл замкнется, если мы вновь посетим уже обойденный узел.

```
select level lvl, graph.*, connect_by_iscycle cycle
  from graph
 start with id_parent = 1
connect by nocycle prior id = id_parent
       and prior id_parent is not null;
```

LVL	ID	ID_PARENT	CYCLE
1	2	1	0
1	3	1	0
2	4	3	0
3	5	4	0
4	3	5	1

Pseudocolumn generation in detail

Ранее в этой уже был затронут вопрос очередности работы join, connect by, where. При наличии в запросе псевдостолбцов, например, level или rownum нельзя сказать, что их генерация происходит после выполнения какой-либо из конструкций. Но можно сформулировать такие правила:

- Значение level инкрементируется при переходе на новый уровень
- Значение rownum инкрементируется при добавлении новой строки в результирующий набор данных

  Продемонстрируем вышесказанное на следующем примере

```
create table t_two_branches(id, id_parent) as
(select rownum, rownum - 1 from dual connect by level <= 10
union all
select 100 + rownum, 100 + rownum - 1 from dual connect by level <= 10
union all
select 0, null from dual
union all
select 100, null from dual);
```

```
select rownum rn,
       level lvl,
       replace(sys_connect_by_path(rownum, '~'), '~') as path_rn,
       replace(sys_connect_by_path(level, '~'), '~') as path_lvl,
       sys_connect_by_path(id, '~') path_id
  from t_two_branches
 where mod(level, 3) = 0
start with id_parent is null
connect by prior id = id_parent;
```

RN	LVL	PATH_RN	PATH_LVL	PATH_ID
1	3	111	123	~0~1~2
2	6	111222	123456	~0~1~2~3~4~5
3	9	111222333	123456789	~0~1~2~3~4~5~6~7~8
4	3	444	123	~100~101~102
5	6	444555	123456	~100~101~102~103~104~105
6	9	444555666	123456789	~100~101~102~103~104~105~106~107~108

```
6 rows selected.
```

Для каждой из веток дерева было сгенерировано 9 уровней, при этом при обходе первой ветки было сгенерировано три строки с первой по третью и при обходе второй ветке еще три строки – с четвертой по шестую.

Интересно также отметить разницу при наличии rownum/level в условии connect by.

```
select rownum rn,
       level lvl,
       replace(sys_connect_by_path(rownum, '~'), '~') as path_rn,
       replace(sys_connect_by_path(level, '~'), '~') as path_lvl,
       sys_connect_by_path(id, '~') path_id
  from t_two_branches
start with id_parent is null
connect by prior id = id_parent
and rownum <= 2;
```

RN	LVL	PATH_RN	PATH_LVL	PATH_ID
1	1	1	1	~0
2	2	12	12	~0~1
3	1	3	1	~100

```
select rownum rn,
       level lvl,
       replace(sys_connect_by_path(rownum, '~'), '~') as path_rn,
       replace(sys_connect_by_path(level, '~'), '~') as path_lvl,
       sys_connect_by_path(id, '~') path_id
  from t_two_branches
start with id_parent is null
connect by prior id = id_parent
and level <= 2;
```

RN	LVL	PATH_RN	PATH_LVL	PATH_ID
1	1	1	1	~0
2	2	12	12	~0~1
3	1	3	1	~100
4	2	34	12	~100~101

В первом примере выполняется соединения для первой ветки до получения двух строк, потом выводится одна строка для второй ветки – хоть условие соединения для нее заведомо ложное, но условие старта истинное. При наличии n веток – в результате было бы по одной строке для всех. Во втором случае просто выполняется обход каждой из веток до второго уровня – соответственно имеет четыре строки в результате.

Резюме

Конструкция connect by – специфическая для Оракла и может быть применена для эффективного построения иерархий или генерации наборов данных без зависимостей между дочерними и родительскими элементами.

6. Recursive subquery factoring

Конструкция with была впервые представлена в Oracle 9.2. В то время она не позволяла выполнять рекурсивные запросы. Польза от данной конструкции была в том, что можно было декомпозировать сложный запрос на последовательность более простых именованных запросов в with или уменьшить общее время выполнения за счет материализации при многократном использовании именованного подзапроса. Производительность в случае декомпозиции часто могла ухудшится из-за того, что трансформации запросов применяются более ограниченно в случае subquery factoring чем с inline views.

С версии Oracle 11.2 конструкция with позволяет выполнять запрос рекурсивно, если подзапрос, определенный в конструкции, ссылается на имя подзапроса определенного самой конструкцией. Схематично это можно изобразить следующим образом

```
with rec as
(
anchor_query_text - anchor member
union all
recursive_query_text - recursive member referencing rec
)
select *
from rec
```

Выполнение происходит по следующему алгоритму

1. выполняется anchor member для получения базового результирующего набора Set(0) этот блок не может ссылаться на имя подзапроса определенного в with
2. выполняется recursive member с набором Set(i-1) полученным на предыдущей итерации этот блок обязан ссылаться на имя подзапроса определенного в with
3. шаг два повторяется до тех пор, пока набор, полученный на текущей итерации не пустой
4. результат запроса это все наборы, объединенные union all

Воспользуемся тем же набором данных, что и в предыдущей главе.

```
create table tree as
select 2 id, 1 id_parent from dual
union all select 3 id, 1 id_parent from dual
union all select 4 id, 3 id_parent from dual
union all select 5 id, 4 id_parent from dual
union all select 11 id, 10 id_parent from dual
union all select 12 id, 11 id_parent from dual
union all select 13 id, 11 id_parent from dual;
```

Запрос ниже возвращает все узлы для двух иерархий, однако порядок отличается от аналогичного запроса с connect by.

```
with rec(lvl, id, path) as
(
select 1 lvl, id, cast('->'||id as varchar2(4000))
  from tree where id_parent in (1, 10)
union all
select r.lvl + 1, t.id, r.path||'->'||t.id
  from tree t
  join rec r on t.id_parent = r.id
)
select lvl,
       rpad(' ', (lvl - 1) * 3, ' ') || id as id,
       path
  from rec;
```

```
       LVL ID         PATH
---------- ---------- ----------
         1 2          ->2
         1 3          ->3
         1 11         ->11
         2    4       ->3->4
         2    12      ->11->12
         2    13      ->11->13
         3       5    ->3->4->5
```

Для того, что выводились сначала дочерние элементы текущего узла, а потом остальные элементы на том же уровне необходимо использовать конструкцию `search depth first`. Эта конструкция влияет только на порядок выдачи результата, но не на порядок обхода иерархии. Детальнее об этом в следующем разделе. По умолчанию результат выводится по уровням (`search breadth first`) - обход в ширину.

```
with tree as
(
select 2 id, 1 id_parent from dual
union all select 3 id, 1 id_parent from dual
union all select 4 id, 3 id_parent from dual
union all select 5 id, 4 id_parent from dual
union all select 11 id, 10 id_parent from dual
union all select 12 id, 11 id_parent from dual
union all select 13 id, 11 id_parent from dual
),
rec(root, lvl, id, id_parent, grand_parent) as
(
select id_parent, 1 lvl, id, id_parent, cast(null as number)
  from tree where id_parent in (1, 10)
union all
select r.root, r.lvl + 1, t.id, t.id_parent, r.id_parent
  from tree t
  join rec r on t.id_parent = r.id
)
search depth first by id set ord
select root,
       lvl,
       rpad(' ', (lvl - 1) * 3, ' ') || id as id,
       id_parent,
       grand_parent,
       ord,
       decode(lvl + 1, lead(lvl) over(partition by root order by ord), 0, 1) is_leaf
  from rec;
```

```
      ROOT        LVL ID          ID_PARENT GRAND_PARENT        ORD    IS_LEAF
---------- ---------- ---------- ---------- ------------ ---------- ----------
         1          1 2                   1                       1          1
         1          1 3                   1                       2          0
         1          2    4                3            1          3          0
         1          3       5             4            3          4          1
        10          1 11                  10                      5          0
        10          2    12               11           10          6          1
        10          2    13               11           10          7          1
```

Для рекурсивных запросов не предусмотрено псевдостолбцов и функций, таких как для connect by, поэтому необходимые значения надо считать самостоятельно.

Например, узел не листовой, только если за ним следует потомок на уровень ниже при обходе в глубину. На основании этого правила получен столбец is_leaf.

Более интересен механизм получения path из предыдущего примера. Стоит обратить внимание, что этих значений не было в исходном наборе, и они получены рекурсивно с помощью конкатенации без каких-либо специальных функций. В этом состоит одно из главных отличий recursive with от connect by: в случае recursive with можно вычислять новые элементы на основании полученных (вычисленных) на предыдущем уровне. Если требуется, например, путь не от корня до текущего узла, а наоборот, то его элементарно получить, добавляя текущее значение атрибута не слева от пути, а справа. В случае с connect by встроенными средствами такой путь получить невозможно и пришлось бы преобразовывать полученную строку.

Далее приведены решения для генерации последовательностей из предыдущей главы.

Для случая, когда i-й элемент зависит только от i-1-го.

```
with t(id, result) as
(
select 0 id, 1 result from dual
union all
select t.id + 1, round(100 * sin(t.result + t.id))
  from t
 where t.id < 20
)
select * from t;
```

Для случая когда i-й зависит от i-1-го и i-2-го.

```
with t (lvl, result, tmp) as
(
select 1, 0, 1 from dual
union all
select lvl + 1, tmp, tmp + result
  from t
 where lvl < 15)
select lvl, result from t;
```

Как видно, во втором случае понадобилось ввести фиктивную колонку, чтоб заглянуть на уровень раньше, чем предыдущий, поскольку на каждой итерации recursive subquery factoring имеет доступ только к набору данных с предыдущей итерации. При необходимости получать доступ к значениям атрибута на нескольких предыдущих итерациях – понадобилось бы добавлять несколько вспомогательных столбцов или накапливать данные в столбце-коллекции.

В любом случае, эффективность данных методов несравнимо лучше, чем в случае с connect by, ибо генерация выполняется не путем перебора имеющихся вариантов, а непосредственно.

Как уже было замечено, при использовании recursive subquery factoring возможно обращаться к значениям, полученным на предыдущем уровне. Данный прием может быть использован для поисков корней уравнения методом половинного деления. Это делается исключительно для демонстрации возможностей языка SQL и его не стоит использовать для решения подобных задач.

Итак, пусть есть функция y следующего вида и мы знаем, что в точках 1 и 2 ее значения имеют разные знаки.

```
create or replace function y(x in number) return number as
begin return x*x - 2; end;
```

Поиск корня на отрезке [1; 2] с точностью 0.01 будет выглядеть следующим образом

```
with t(id,x,x0,x1) as
(
  select 0, 0, 1, 2
    from dual
  union all
  select t.id + 1,
         (t.x0 + t.x1) / 2,
         case
           when sign(y(x0)) = sign(y((t.x0 + t.x1) / 2))
           then (t.x0 + t.x1) / 2
           else x0
         end,
         case
           when sign(y(x1)) = sign(y((t.x0 + t.x1) / 2))
           then (t.x0 + t.x1) / 2
           else x1
         end
    from t
   where abs((t.x0 + t.x1) / 2 - t.x) > 1e-2
)
select t.*, (x0+x1)/2 result from t;
```

ID	X	X0	X1	RESULT
0	0	1	2	1.5
1	1.5	1	1.5	1.25
2	1.25	1.25	1.5	1.375
3	1.375	1.375	1.5	1.4375
4	1.4375	1.375	1.4375	1.40625
5	1.40625	1.40625	1.4375	1.421875
6	1.421875	1.40625	1.421875	1.4140625

Алгоритм работает по принципу: если знак функции в середине текущего отрезка совпадает со знаком функции на правом конце, то правый конец сдвигаем в середину; для левого конца применяем тот же принцип. Алгоритм останавливается, когда разница значений корня на текущей итерации и середины отрезка на текущей итерации становится меньшей либо равной

0.01. Желаемая точность была достигнута на шестой итерации и середина отрезка на последней итерации есть приближенное значение корня - **1.4140625**.

Значения алгоритма на каждой итерации вычислялись с учетом значений полученных на предыдущих итерациях, в случае использования конструкции connect by такой подход был бы невозможен. Слово итерация было использовано вместо слово уровень, чтоб подчеркнуть итеративную природу выполнения.

Traversing hierarchies

В документации сказано «subquery_factoring_clause, which supports recursive subquery factoring (recursive WITH) and lets you query hierarchical data. **This feature is more powerful than CONNECT BY in that it provides depth-first search and breadth-first search**, and supports multiple recursive branches». То есть подразумевается, что при connect by всегда выполняется обход в глубину, тогда как при recursive with можно менять алгоритм обхода путем указания search_clause.

Функция ниже устанавливает флаг при обходе идентификатора с заданным значением и возвращает непустое значение если он хоть раз был установлен при выполнении запроса.

```
create or replace function stop_at(p_id in number, p_stop in number)
  return number is
begin
  if p_id = p_stop then
    dbms_application_info.set_client_info('1');
    return 1;
  end if;
  for i in (select client_info from v$session where sid = userenv('sid')) loop
    return i.client_info;
  end loop;
end;
```

Воспользуемся набором данных из предыдущей главы и выполним обход с указанием метода в глубину и в ширину.

```
exec dbms_application_info.set_client_info('');

PL/SQL procedure successfully completed.

with rec(lvl, id) as
(
select 1, id
  from t_two_branches where id_parent is null
union all
select r.lvl + 1, t.id
  from t_two_branches t
  join rec r on t.id_parent = r.id
 where stop_at(t.id, 101) is null
)
search breadth first by id set ord
--search depth first by id set ord
select *
from rec;
```

```
       LVL         ID        ORD
---------- ---------- ----------
         1          0          1
         1        100          2
         2          1          3

exec dbms_application_info.set_client_info('');

PL/SQL procedure successfully completed.

with rec(lvl, id) as
(
select 1, id
  from t_two_branches where id_parent is null
union all
select r.lvl + 1, t.id
  from t_two_branches t
  join rec r on t.id_parent = r.id
 where stop_at(t.id, 101) is null
)
--search breadth first by id set ord
search depth first by id set ord
select *
from rec;

       LVL         ID        ORD
---------- ---------- ----------
         1          0          1
         2          1          2
         1        100          3
```

В обоих случаях мы имеем одинаковое число строк в результате, хотя при обходе в глубину ожидалось получить вес строки первой ветки, ведь ни одно из ее значений не равно 101. Однако когда такое значение встретилось во второй ветке и обход остановился, что говорит о том, что он выполнялся в ширину. Единственная разница в результатах – это порядок строк.

Проверим аналогичный подход для connect by.

```
select rownum rn, level lvl, id, id_parent
  from t_two_branches
 start with id_parent is null
connect by prior id = id_parent
       and stop_at(id, 101) is null;

        RN        LVL         ID  ID_PARENT
---------- ---------- ---------- ----------
         1          1          0
         2          2          1          0
         3          3          2          1
         4          4          3          2
         5          5          4          3
         6          6          5          4
         7          7          6          5
         8          8          7          6
         9          9          8          7
        10         10          9          8
        11         11         10          9
        12          1        100

12 rows selected.
```

В этом случае первая ветка была обойдена полностью, что свидетельствует о том, что обход выполнялся в глубину.

Подытоживая сказанное, в случае connect by обход всегда выполняется в глубину, в случае recursive subquery factoring – всегда в ширину. Если цель поменять порядок результата, то для connect by можно использовать order by level, а для recursive with – search_clause.

Once again about cycles

Рассмотрим работу с циклами для данных из предыдущей главы.

```
with t(id, id_parent) as
(
select * from graph where id_parent = 1
union all
select g.id, g.id_parent
  from t
  join graph g on t.id = g.id_parent
)
search depth first by id set ord
cycle id set cycle to 1 default 0
select * from t;
```

```
        ID  ID_PARENT        ORD CYCLE
---------- ---------- ---------- ----------
         2          1          1 0
         3          1          2 0
         4          3          3 0
         5          4          4 0
         3          5          5 1
```

Для того, чтоб выполнение запроса не зацикливалось использована конструкция **cycle id set cycle to 1 default 0**. Она означает, что если при выполнении обнаруживается цикл по id, то проставляется признак cycle для строки равный единице и получение дальнейших дочерних записей для этой строки приостанавливается. Строка считается замыкающей цикл, если значение атрибута указанного в cycle уже встречалось для одной из родительских записей. Иными словами если для строки проставлен признак цикла, то в результате однозначно существует строка с тем же значением атрибута, по которому вычисляется цикл.

Видно, что циклом признак цикла проставлен для узла, если он уже был обойден (id = 3). В случае connect by (без условия prior id_parent is not null) признак цикла был поставлен в узел, потомком которого был уже обойденный (id = 5). Кроме того, можно указывать, по какому столбцу определять циклы, при указании cycle id_parent результат был бы несколько иным – выполнение остановилось бы при повторном выводе id_parent = 3.

```
with t(id, id_parent) as
(
select * from graph where id_parent = 1
union all
select g.id, g.id_parent
  from t
  join graph g on t.id = g.id_parent
)
search depth first by id set ord
cycle id_parent set cycle to 1 default 0
select * from t;
```

ID	ID_PARENT	ORD	C
2	1	1	0
3	1	2	0
4	3	3	0
5	4	4	0
3	5	5	0
4	3	6	1

На первый взгляд поведение recursive subquery factoring с указанием цикла по id совпадает с connect by при наличии дополнительного условия «prior id_parent is not null», но это не всегда так. Если цикл замыкается в начальной точке построения, то результаты будут отличаться. Ниже результаты при старте с «id = 3».

```
select level lvl, graph.*, connect_by_iscycle cycle
  from graph
 start with id = 3
connect by nocycle prior id = id_parent;
```

LVL	ID	ID_PARENT	CYCLE
1	3	1	0
2	4	3	0
3	5	4	1
1	3	5	0
2	4	3	0
3	5	4	1

```
select level lvl, graph.*, connect_by_iscycle cycle
  from graph
 start with id = 3
connect by nocycle prior id = id_parent
       and prior id_parent is not null;
```

LVL	ID	ID_PARENT	CYCLE
1	3	1	0
2	4	3	0
3	5	4	0
4	3	5	1
1	3	5	0
2	4	3	0
3	5	4	1

```
with t(id, id_parent) as
(
select * from graph where id = 3
union all
select g.id, g.id_parent
  from t
  join graph g on t.id = g.id_parent
)
search depth first by id set ord
cycle id set cycle to 1 default 0
select * from t;
```

ID	ID_PARENT	ORD	C
3	1	1	0
4	3	2	0
5	4	3	0
3	5	4	1
3	5	5	0
4	3	6	0
5	4	7	0
3	5	8	1

В последнем случае ребро (4, 3) выведено дважды, потому что как уже было сказано, атрибут по которому определяется цикл должен повториться. С другой стороны, результат второго запроса наиболее «естественный».

Recursive subquery factoring позволяет реализовывать собственные условия для определения циклов. Так, в примере ниже, cnt1 определяет число вхождений id текущей строки в конкатенации id, аналогично cnt2 определяет число вхождений id в конкатенации родительских id. Если раскомментировать фильтры по cnt2/cnt1, то можно получить наборы данных соответственно как для запросов выше с connect by/recursive subquery factoring, указывать конструкцию cycle при наличии таких фильтров нет необходимости.

```
with t(id, id_parent, path_id, path_id_parent, cnt1, cnt2) as
(
select g.*,
       cast('->' || g.id as varchar2(4000)),
       cast('->' || g.id_parent as varchar2(4000)),
       0,
       0
  from graph g
 where id = 3
union all
select g.id,
       g.id_parent,
       t.path_id || '->' || g.id,
       t.path_id_parent || '->' || g.id_parent,
       regexp_count(t.path_id || '->', '->' || g.id || '->'),
       regexp_count(t.path_id_parent || '->', '->' || g.id || '->')
  from t
  join graph g
    on t.id = g.id_parent
-- and t.cnt1 = 0
-- and t.cnt2 = 0
)
search depth first by id set ord
cycle id set cycle to 1 default 0
select * from t;
```

```
        ID  ID_PARENT PATH_ID          PATH_ID_PARENT        CNT1       CNT2        ORD C
---------- ---------- ---------------- ---------------- ---------- ---------- ---------- -
         3          1 ->3              ->1                       0          0          1 0
         4          3 ->3->4           ->1->3                    0          0          2 0
         5          4 ->3->4->5        ->1->3->4                 0          0          3 0
         3          5 ->3->4->5->3     ->1->3->4->5              1          1          4 1
         3          5 ->3              ->5                       0          0          5 0
         4          3 ->3->4           ->5->3                    0          0          6 0
         5          4 ->3->4->5        ->5->3->4                 0          1          7 0
         3          5 ->3->4->5->3     ->5->3->4->5              1          1          8 1
```

Limitations of current implementation

Запрос в рекурсивном члене имеет множество ограничений, так нельзя использовать: distinct, group by, having, connect by, aggregate functions и так далее. Возникает вопрос: это ограничение текущей реализации или рекурсивное выполнение не имеет смысла при использовании сложной логики. Я склоняюсь к тому, что, возможно, некоторые из текущих ограничений будут убраны в будущем. С другой стороны, часть ограничений можно обойти уже сейчас, но выглядит это несколько неуклюже. Например, невозможность использовать групповые функции можно обойти, используя аналитику, но делать это в реальных задачах однозначно не стоит.

Дальнейшее будет продемонстрировано в академических целях. Итак, представим, что необходимо построить иерархию по связи id <- id_parent, при этом суммировать id всех узлов текущего уровня и для следующего уровня получать узлы такие, что родительский id равен полученной сумме.

```
with t0(id, id_parent, letter) as
(select 1, 0, 'B' from dual
union all select 2, 1, 'D' from dual
union all select 3, 1, 'A' from dual
union all select 10, 5, 'C' from dual
union all select 66, 6, 'X' from dual),
t(id, id_parent, sum_id, lvl, str, rn) as
(select id, id_parent, id, 1, letter, 1 from t0 where id_parent = 0
 union all
 select
   t0.id,
   t0.id_parent,
   sum(t0.id) over (),
   t.lvl + 1,
   listagg(letter, ', ') within group (order by letter) over (),
   rownum
 from t
 join t0 on t.sum_id = t0.id_parent and t.rn = 1
 where lvl <= 3)
select * from t;
```

```
        ID  ID_PARENT     SUM_ID        LVL STR                 RN
---------- ---------- ---------- ---------- ---------- ----------
         1          0          1          1 B                   1
         3          1          5          2 A, D                2
         2          1          5          2 A, D                1
        10          5         10          3 C                   1
```

Вместо групповых функций были использованы аналитические, но поскольку при этом «схлопывания» не происходит, то для получения узлов следующей итерации была использована только первая строка текущей благодаря использованию rownum.

Если в результате интересуют только «агрегированные» значения каждого уровня, то можно было бы использовать, например, такой запрос

```
select sum_id, lvl, str, rn from t where rn = 1;
```

На версии 11.2.0.1, при попытке использовать аналитические функции, возникла бы ошибка «ORA-32486: unsupported operation in recursive branch of recursive WITH clause», на 11.2.0.3 подобной ошибки уже не возникает.

Резюме

Recursive factoring clause соответствует стандарту SQL:1999 для выполнения рекурсивных запросов, тогда как connect by – конструкция специфическая для Oracle. Тем не менее, при работе с иерархическими данными или для генерации последовательностей во всех случаях, когда можно, лучше применять последнюю, поскольку она потребляет меньше ресурсов и быстрее работает. Как уже было сказано, при построении иерархии с помощью connect by и обращении к атрибуту родительской строки с помощью оператора prior, выражения в списке выборки, содержащие обращение к родительским записям, вычисляются для каждого уровня отдельно, а не «накопительно». Поэтому в случаях, когда надо организовать итеративные вычисления по мере обхода древовидной структуры, имеет смысл использовать recursive subquery factoring. Дополнительным плюсом при работе с иерархическими структурами является возможность указывать метод обхода – в глубину или в ширину.

Обход древовидных структур это не единственное применение recursive subquery factoring. Конструкция может быть использована для многих задач, требующих многократной пошаговой обработки набора данных, учитывающей результаты предыдущих шагов. Тем не менее, важно помнить, что результатом запроса будут строки из наборов данных на всех итерациях, тогда как на каждой итерации доступны для обращения строки только из предыдущего набора. Подобная реализация требует крупного объема памяти для рабочей области (workarea) и много других накладных расходов. В случае больших объемов данных, реализация через циклы на PL/SQL и временные таблицы или коллекции однозначно будет выигрывать по производительности, на небольших объемах тоже крайне затруднительно придумать пример запроса с recursive subquery, который давал бы заметный выигрыш при сравнении с PL/SQL подходом, но сопоставимые варианты существуют.

Несмотря на то, что и recursive subquery factoring и connect by содержат встроенные механизмы для обработки циклов, для работы с графами эти инструменты имеет смысл использовать только в достаточно вырожденных случаях. Для всех остальных задач, процедурные решения оказываются предпочтительнее. Все же, в двух SQL подходах имеются отличия при обработке циклов и их важно понимать.

7. Model

Model clause можно назвать наиболее мощной конструкцией SQL, в том смысле что с её помощью можно решать огромное множество таких задач, которые иначе на SQL не решаемы. Это достигается не только благодаря тому, что model значительно расширяет возможности декларативного языка SQL, но и в результате того, что model позволяет выполнять итеративные вычисления правил над набором данных. С другой стороны, класс задач, для которых целесообразно применять model, весьма ограничен, и с производительностью тоже не все гладко. Из-за плохой масштабируемости, при увеличении объемов данных возникает много проблем, в результате чего целесообразнее становится реализовать логику на PL/SQL вместо model clause. Но обо всем по порядку.

Model clause позволяет работать с результатом запроса как с многомерным кубом, указав при этом список мер и измерений.

Для использования model надо как минимум указать

1) Измерения (dimensions). По умолчанию комбинация всех измерений должна однозначно идентифицировать каждую строку (ячейку куба).
2) Результирующие меры (measures). В отличие от некоторых других многомерных средств анализа Oracle поддерживает для мер не только числовые значения, но и строки, даты или, например, raw.

И меры и измерения могут быть указаны либо как маппинг к существующим колонкам либо как выражения, при этом значения мер впоследствии могут быть пересчитаны благодаря применению правил (rules).

Принцип работы лучше объяснить на примере

```
with t(id, value) as
(
select 1, 3 from dual
union all select 2, 9 from dual
union all select 3, 8 from dual
union all select 5, 5 from dual
union all select 10, 4 from dual
)
select *
from t
model
--return updated rows
dimension by (id)
measures (value, 0 result)
--rules
(
  result[id >= 5] = sum(value)[id <= cv(id)],
  result[0] = value[10] + value[value[1]]
);
```

```
        ID      VALUE     RESULT
---------- ---------- ----------
         1          3          0
         2          9          0
         3          8          0
         5          5         25
        10          4         29
         0                    12
```

В запросе определено измерение по столбцу ID и две меры – одна на основании столбца value, другая – со значением 0.

«`result[id >= 5] = sum(value)[id <= cv(id)]`» – означает, что мера будет считаться для всех строк (ячеек) со значением измерения больше либо равного 5 как сумма значений меры value по всем строкам для которых значение измерения меньше либо равно значению измерения для текущей строки. Если выражение в левой части правила охватывает более одной ячейки, то функция **cv** может быть использована в правой части правила для получения текущих значений измерений. В документации к Oracle 10.1 также встречается функция **currentv**, но в последующих версиях ее упоминания нет, хотя, по-видимому, она продолжает работать.

«`result[0] = value[10] + value[value[1]]`» – означает, что мера для значения измерения 0 будет посчитана как сумма мер для значений измерения 10 и 3 (value[1] = 3). Здесь стоит обратить внимание, что поскольку ячейки с id=0 не было, то она будет создана. Как будет показано далее, такое поведение может быть изменено. value[value[1]] – пример вложенных ячеек (nested cell reference).

Иными словами, меры можно рассматривать как многомерные массивы, а измерения – как индексы для доступа к элементам массивов.

Если в примере выше раскомментировать **return updated rows**, то будут возвращены только те строки, к которым были применены правила. По умолчанию значение **return all rows**.

Адресация ячеек может быть выполнена с помощью символьной и позиционной нотации. В случае символьной нотации используется предикат, в котором фигурирует название измерения, в противном случае нотация позиционная – например константа или for с подзапросом.

Поведение с отсутствующими ячейками может регулироваться ключевыми словами **update/upsert all/upsert**. **Update** только обновляет существующие строки, **upsert** (используется по умолчанию) обновляет существующие и добавляет пропущенные, если использована позиционная нотация, **upsert all** возвращает также строки, если использована комбинированная нотация и ячейки для измерений с символьной нотацией существуют.

```
with t(dim1, dim2, value) as
(
select 0, 0, 1 from dual
union all select 0, 1, 2 from dual
union all select 1, 0, 3 from dual
)
select *
from t
model
dimension by (dim1, dim2)
measures (value, cast(null as number) result)
rules upsert all
(
  result[0, 0] = -1,
  result[dim1=1, dim2=0] = -3,
  result[-1, for dim2 in (select count(*) from dual)] = -4,
  result[-2, dim2=1] = -10,
  result[-3, dim2=-1] = -100,
  result[-4, -1] = -1000
)
order by dim1, dim2;
```

```
      DIM1       DIM2      VALUE     RESULT
---------- ---------- ---------- ----------
        -4         -1                 -1000
        -2          1                   -10
        -1          1                    -4
         0          0          1         -1
         0          1          2
         1          0          3         -3
```

Пример выше вернул три исходных строки и три новых.

Ячейки со значением меры result -1, -4, -1000 есть в результате, так как была использована позиционная нотация для обоих измерений, значение -10 есть в результате, так как при использовании символьной нотации для dim2 это значение присутствовало в измерении до начала вычислений (хоть позиционное значение измерения dim1 и отсутствовало). Значение -100 не попало в результат, так как в исходных данных не было значения измерения dim2 равного -1, которое было использовано в символьной нотации. Значение меры для измерений (0, 1) неопределенно, так как для него не было указано правила, значение меры для (1, 0) равно -3, в правиле использовалась символьная нотация для обоих измерений, их значения существовали до применения model.

При использовании **upsert** из результата пропадет значение -10, при использовании **update** из результата пропадут также значения -4 и -1000.

Простым языком, символьная нотация используется, если цель обновить значения в наборе данных, а позиционная - если может быть необходимость добавить новые значения (ячейки) – например, в случае прогнозирования. Их комбинация имеет смысл, когда для некоторых измерений есть необходимость добавить новые значения, а для других – нет.

Для того, чтобы сослаться на все члены измерения может быть использовано ключевое слово **any** в случае позиционной нотации или условие **is any** в случае символьной, однако работает в обоих случаях одинаково: если для одного из измерений использовано any/is any, то новые строки не могут быть добавлены к результату.

```
with t(id, value) as
(select rownum, rownum from dual connect by level <= 3)
select *
from t
model
dimension by (id)
measures (value, 100 r1, 100 r2)
(
  r1[any] order by id asc  = nvl(r1[cv(id)-1], 0) + value[cv(id)],
  r2[id is any] order by id desc = nvl(r2[cv(id)-1], 0) + value[cv(id)]
)
order by id;
```

```
        ID      VALUE         R1         R2
---------- ---------- ---------- ----------
         1          1          1          1
         2          2          3        102
         3          3          6        103
```

Как видно из примера выше, важное значение имеет сортировка. При сортировке по возрастанию был получен нарастающий итог. При сортировке по убыванию получены совсем

иные значения. 102 и 103 получено, так как значение меры value для текущего значения измерения было добавлено к значению меры r2 для предыдущего значения измерения, которое равно 100 – r2 изначально было инициализировано этим значением для всех ячеек.

Всегда имеет смысл указывать сортировку в левой части правил содержащих диапазоны ячеек, поскольку

- в таком случае улучшается производительность
- это вносит ясность
- позволяет избежать ошибки «ORA-32637: Self cyclic rule in sequential order MODEL»

Например, в примере ниже значение меры определено через неё же.

```
with t as
(select rownum id from dual connect by level <= 3)
select *
from t
model
dimension by (id)
measures (id result)
rules
(
  result[any] /*order by id*/ = sum(result)[any]
);
(select rownum id from dual connect by level <= 3)
                        *
ERROR at line 2:
ORA-32637: Self cyclic rule in sequential order MODEL
```

При указании сортировки в определении правила ошибка возникать не будет.

По умолчанию правила вычисляются в том порядке, в котором они описаны в запросе. Это также может быть указано явно с помощью ключевых слова **sequential order**. Данное поведение может быть изменено.

```
with t as
(select rownum id from dual connect by level <= 3)
select *
from t
model
dimension by (id)
measures (0 t1, 0 x, 0 t2)
rules automatic order
(
  t1[id] = x[cv(id)-1],
  x[id] = cv(id),
  t2[id] = x[cv(id)-1]
)
order by id;

        ID         T1          X         T2
---------- ---------- ---------- ----------
         1                     1
         2          1          2          1
         3          2          3          2
```

При указании **automatic order** учитываются зависимости между правилами. По умолчанию значения t1 были бы null, 0, 0.

Стоит отметить, что для одной и той же меры может быть указано более одного правила, при этом правила могут использоваться для одних и тех же ячеек. Например, в примере выше могло бы присутствовать четвертое правило такого вида «`t1[id] = x[cv(id)] + t2[cv(id)]`».

Независимо от порядка правила вычисляются сразу для всех ячеек указанных в левой части. Если в левой части указано any или имя измерения, то мера будет вычислена сразу для всех строк. Иными словами правила в любом случае применяются «по столбцам» а не «по строкам».

Ключевые слова automatic/sequential order определяют план с которым будет выполняться запрос. Для sequential это будет SQL MODEL ORDERED, в случае automatic order может быть SQL MODEL ACYCLIC/SQL MODEL CYCLIC в зависимости от отсутствия/наличия циклических зависимостей между правилами. В простом случае циклическая зависимость может быть как в рамках одной меры (одна ячейка зависит от другой, а вторая от первой), так и для разных мер и одной ячейки. Честно говоря, мне не встречались разумные примеры применения циклических правил, так что я бы рекомендовал всегда перечислять правила в правильном порядке и пользоваться значению по умолчанию «sequential order» для правил, а для каждой из мер в левой части присваивания указывать order by.

В случае ORDERED/ACYCLIC моделей в плане может также присутствовать слово FAST, если в левой части всех правил используется указание конкретных ячеек (single cell reference).

Например, для следующих правил

```
rules automatic order (x[1] = cv(id), x[-1] = cv(id))
```

в плане будет SQL MODEL ACYCLIC **FAST**, тогда как для

```
rules automatic order (x[for id in (1, -1)] = cv(id))
```

или

```
rules automatic order (x[id in (1, -1)] = cv(id))
```

в план SQL MODEL ACYCLIC. При наличии ключевого слова for нотация символьная, а без него – позиционная, соответственно, если какие-то значения из списка отсутствуют в исходном наборе, то результаты будут отличаться.

Если попытаться для упомянутого примера с циклической зависимостью указать «automatic order», получим ошибку, что модель не сходится.

```
with t as
(select rownum id from dual connect by level <= 3)
select *
from t
model
dimension by (id)
measures (id result)
rules automatic order
(
  result[any] /*order by id*/ = sum(result)[any]
);
from t
     *
ERROR at line 4:
ORA-32634: automatic order MODEL evaluation does not converge
```

Согласно документации: «Convergence is defined as the state in which further executions of the model will not change values of any of the cell in the model». Опытным путем можно проверить, что сходимость определяется тремя (четырьмя) проходами.

```
select * from (select 1 x from dual)
model dimension by (x) measures (0 as result, 64 tmp)
rules automatic order
(result[1]=ceil(tmp[1]/4), tmp[1]=result[1]);

         X     RESULT        TMP
---------- ---------- ----------
         1          1          1

select * from (select 1 x from dual)
model dimension by (x) measures (0 as result, 65 tmp)
rules automatic order
(result[1]=ceil(tmp[1]/4), tmp[1]=result[1]);
select * from (select 1 x from dual)
                               *
ERROR at line 1:
ORA-32634: automatic order MODEL evaluation does not converge
```

В первом случае последовательность вычислений для (result, tmp) была 16, 4, 1 и снова 1. Во втором случае значения после третьего и четвертого прохода отличались, и была получена ошибка.

При отсутствии циклических зависимостей automatic order может повлиять на результат запроса, меняя план выполнения с MODEL ORDERED на MODEL ACYCLIC, но такого же эффекта можно добиться, если в «правильном порядке» указать правила.

Во всех показанных примерах правила модели вычислялись однократно, однако они могут быть вычислены определенное число итераций (в случае sequential order моделей). Чтоб продемонстрировать эту возможность будет приведено нахождение корня уравнения методом половинного деления. Функция и начальный отрезок взяты такие же как и в предыдущей главе. Алгоритм выглядит следующим образом:

```
with t as (select 0 id from dual)
select *
from t
model
dimension by (id)
measures (0 x, 1 x0, 2 x1)
rules iterate (1e2) until abs(x[0]-previous(x[0]))<1e-2
(
  x[0] = (x0[iteration_number] + x1[iteration_number])/2,
  x[iteration_number+1] = x[0],
  x0[iteration_number+1] = case when sign(y(x[0])) =
sign(y(x0[iteration_number]))
                               then x[0]
                               else x0[iteration_number]
                          end,
  x1[iteration_number+1] = case when sign(y(x[0])) =
sign(y(x1[iteration_number]))
                               then x[0]
                               else x1[iteration_number]
                          end
)
order by id;
```

```
        ID          X         X0         X1
---------- ---------- ---------- ----------
         0  1.4140625          1          2
         1        1.5          1        1.5
         2       1.25       1.25        1.5
         3      1.375      1.375        1.5
         4     1.4375      1.375     1.4375
         5    1.40625    1.40625     1.4375
         6   1.421875    1.40625   1.421875
         7  1.4140625  1.4140625   1.421875
```

Iteration_number – функция, которая возвращает номер итерации начиная с нуля.

Максимальное число итераций, которое могло быть выполнено в примере выше равно 100 (оно может быть задано только числом без использования выражений), однако было добавлено условие останова: если значение корня, вычисленное на предыдущей итерации (для этого использована функция **previous**), отличается от текущего значения корня менее чем на 0.01, то выполнение прерывается. В результате было выполнено 7 итераций и найдено приблизительное значение корня - **1.4140625**. Обратите внимание, что в реализации с помощью model было выполнено на одну итерацию больше чем в случае с recursive subquery factoring, потому что в условие останова сформулировано несколько иначе (в предыдущей реализации условие останова вычислялось исключительно на данных текущей итерации).

Названия операций в плане для model clause не меняются при использовании итераций, и нет никаких дополнительных операций, указывающих на выполнение итераций. Более того, по столбцу starts для dbms_xplan.display_cursor (при включенных runtime execution statistics) тоже невозможно определить выполнялась ли операция model несколько итераций или нет.

В model clause могут быть определены **reference model**(s) которые удобно использовать как “lookup arrays”.

```
with sales(year, currency, value) as
(select '2015', 'GBP', 100 from dual
union all select '2015', 'USD', 200 from dual
union all select '2015', 'EUR', 300 from dual
union all select '2016', 'GBP', 400 from dual
union all select '2016', 'EUR', 500 from dual)
, usd_rates(currency, rate) as
(select 'GBP', 1.45 from dual
union all select 'USD', 1 from dual
union all select 'EUR', 1.12 from dual)
select *
from sales
model
  reference usd_rates_model on (select * from usd_rates)
  dimension by (currency)
  measures (rate)
main sales_model
dimension by (year, currency)
measures (value, 0 usd_value)
(
  usd_value[any, any] order by year, currency =
    value[cv(year), cv(currency)] * usd_rates_model.rate[cv(currency)]
)
order by 1, 2;
```

```
YEAR CUR      VALUE  USD_VALUE
---- --- ---------- ----------
2015 EUR         300        336
2015 GBP         100        145
2015 USD         200        200
2016 EUR         500        560
2016 GBP         400        580
```

В начале было сказано, что комбинация значений всех измерений должна однозначно идентифицировать ячейку. Но это правило может быть ослаблено, если указать ключевые слова **unique single reference**. По умолчанию применяется **unique dimension**. Пример ниже демонстрирует различия в поведении.

```
with t(id, value) as
(select trunc(rownum/2), rownum from dual connect by level <= 3)
select *
from t
model
unique single reference
dimension by (id)
measures (value, 0 result)
(result[0] = 111)
order by id;
```

```
        ID      VALUE     RESULT
---------- ---------- ----------
         0          1        111
         1          2          0
         1          3          0
```

При значении по умолчанию выполнение запроса выдало бы исключение.

Последнее, что стоит упомянуть из базовых вещей – это особенности работы с null values. В model clause могут быть использованы функции presentv/presentnnv. Работают по аналогии с nvl2, при этом первая проверят существование ячейки до начала вычислений, а вторая не только существование, но и не пустое значение.

```
with t(id, value) as
(select 0, cast(null as varchar2(4000)) from dual)
select *
from t
model
ignore nav
dimension by (id)
measures (value, cast(null as varchar2(4000)) result, to_number(null) num)
(
  value[1] = '1',
  value[2] = presentv(value[0], value[0], 'V'),
  value[3] = presentnnv(value[0], value[0], 'NNV'),
  value[4] = nvl2(value[0], value[0], 'NVL2'),
  result[2] = presentv(value[1], value[1], 'V'),
  result[3] = presentnnv(value[1], value[1], 'NNV'),
  result[4] = nvl2(value[1], value[1], 'NVL2'),
  num[any] = num[-1]
)
order by id;
```

```
        ID VALUE      RESULT               NUM
---------- ---------- ---------- ----------
         0                                  0
         1 1                                0
         2            V                     0
         3 NNV        NNV                   0
         4 NVL2       1                     0
```

При указании ключевых слов **ignore nav** значение численных мер для несуществующих ячеек будут трактоваться как 0. По умолчанию они трактуются как null – такое поведение может быть явно указано при помощи **keep nav**. В случае поведения по умолчанию все значения столбца num были бы null.

Возможности model clause могут быть расширены за счет использования аналитических функций, что продемонстрировано в примере ниже.

```
with t(value) as
(select column_value from table(sys.odcivarchar2list('A','B','C','D','E')))
select *
from t
model
ignore nav
dimension by (row_number() over (order by value) id)
measures (value, cast(null as varchar2(4000)) result, count(*) over () num)
(
  result[mod(id, 2) = 1] = listagg(value, ', ') within group (order by id)
over (),
  num[mod(id, 2) = 1] = count(*) over (order by id desc)
)
order by id;
```

```
        ID VALUE      RESULT               NUM
---------- ---------- ---------- ----------
         1 A          A, C, E              3
         2 B                               5
         3 C          A, C, E              2
         4 D                               5
         5 E          A, C, E              1
```

Правила с использованием аналитики могут быть переписаны с использованием агрегатных функций, в этом случае они будут выглядеть так:

```
  result[mod(id, 2) = 1] = listagg(value, ', ') within group (order by
null)[mod(id, 2) = 1],
  num[mod(id, 2) = 1] = count(*)[mod(id, 2) = 1 and id >= cv(id)]
```

Агрегатные функции в конструкции model кардинальным образом отличаются от обычных агрегатных функций тем, что не требуют группировки. Фактически они используются для того чтобы получить конкретное значение от набора строк указанного в левой части правила.

Агрегатные функции позволяют адресовать строки в конструкции model более гибко чем «аналитическое окно». Но аргументами аналитических функций могут быть не только меры, но и измерения в отличие от агрегатных функций. Также при использовании аналитики нельзя

указывать сортировку в правой части правил. Более конкретно, первое и третье правило в примере ниже будут выдавать ошибки.

```
select *
from (select rownum id from dual connect by rownum <= 3) t
model
dimension by (id)
measures (id value, 0 r1, 0 r2)
(
  -- 1)
  -- ORA-30483: window  functions are not allowed here
  -- r1[any] order by id = sum(id) over (order by id desc)
  -- 2)
  r1[any] /*order by id*/ = sum(id) over (order by id desc),
  -- 3) ORA-00904: : invalid identifier
  -- r2[any] order by id desc = sum(id)[id >= cv(id)]
  -- 4)
  r2[any] = sum(value)[id >= cv(id)]
)
```

Что касается гибкости адресации в случае агрегатных функций – рассмотрим пример, использованный ранее для демонстрации ограничения аналитических функций (глава «Analytic functions»).

Элементарно обходится первое ограничение для аналитических функций, поскольку адресация в конструкции model может быть многомерной, однако второе ограничение обойти чуть сложнее, так как в агрегатной функции нельзя использовать значение меры, соответствующее текущим значениям измерений в левой части правила.

```
with points as
 (select rownum id, rownum * rownum x, mod(rownum, 3) y
    from dual
  connect by rownum <= 6)
, t as
(select p.*,
       -- число точек на расстоянии 5 от текущей
       -- в случае рассмотрения двух координат не решается
       count(*) over(order by x range between 5 preceding and 5 following)
cnt,
       -- сумма расстояний от всех точек _до текущей строки включительно_ до
точки (3, 3)
       -- в случае расстоянии до текущей строки а не константы не решается
       round(sum(sqrt((x - 3) * (x - 3) + (y - 3) * (y - 3)))
             over(order by id),
             2) dist
  from points p)
select *
from t
model
dimension by (x, y)
measures (id, cnt, dist, 0 cnt2)
rules
( cnt2[any,any] = count(*)[x between cv(x)-5 and cv(x)+5, y between cv(y)-1
and cv(y)+1] )
order by id;
```

Для обхода второго ограничения можно воспользоваться итеративной моделью, но подобное решение все равно выглядит неуклюже.

```
with points as
 (select rownum id, rownum * rownum x, mod(rownum, 3) y
    from dual
  connect by rownum <= 6)
select *
from points
model
dimension by (id)
measures (id i, x, y, 0 x_cur, 0 y_cur, 0 dist2)
rules iterate (1e6) until i[iteration_number+2] is null
(
  x_cur[any] = x[iteration_number + 1],
  y_cur[any] = y[iteration_number + 1],
  dist2[any] = case
               when i[cv(id)] = iteration_number + 1
               then round(sum(sqrt((x - x_cur) * (x - x_cur) + (y - y_cur) *
(y - y_cur)))[id <= cv(id)], 2)
               else dist2[cv(id)]
               end
)
order by id;
```

В запросе выполняется столько итераций сколько строк в наборе данных и на каждой итерации вспомогательным мерам x_cur, y_cur присваиваются значения мер для строки с номером итерации + 1 (как уже было замечено номера итераций начинаются с 0), после этого обновляется значение меры для этой же строки, для всех остальных строк значения меры остаются прежними.

Учитывая, что варианты использования агрегатных и аналитических функций в конструкции model частично пересекаются, этот вопрос еще раз будет затронут при анализе производительности.

Перейдем к более конкретным задачам.

Генерации двух уже рассмотренных последовательностей с помощью конструкции model будет выглядеть следующим образом.

```
select *
from dual
model
dimension by (0 id)
measures (1 result)
rules
(
  result[for id from 1 to 20 increment 1] =
  round(100 * sin(result[cv(id)-1] + cv(id) - 1))
);
```

```
select *
  from (select rownum lvl, rownum - 1 result from dual connect by level <= 2)
model
ignore nav
dimension by (lvl)
measures (result)
rules
(
  result[for lvl from 3 to 15 increment 1] =
  result[cv(lvl)-1] + result[cv(lvl)-2]
);
```

Model позволяет обращаться к значениям, вычисленным на предыдущих этапах, по аналогии с recursive subquery factoring. Это то, что невозможно реализовать с помощью конструкции connect by, которая в случае обращения к предыдущему уровню (использование оператора prior) оперирует набором данных, который был до начала выполнения.

В отличие от recursive subquery factoring, model позволяет легко обратиться к значениям на любом из предыдущих уровней без создания дополнительных мер (столбцов).

Несмотря на некоторую схожесть – эти инструменты совершенно разные. Даже одинаковую терминологию применять не совсем корректно. В случае recursive subquery factoring можно говорить про «обращение к значению, вычисленному на предыдущем уровне» или «обращение к значению для родительской записи», тогда как в случае model более корректно будет сказать «обращение к знамению меры для предыдущего члена измерения». Не стоит забывать, что предназначение recursive with – работа с иерархическими данными, тогда как model был спроектирован для работы с многомерными данными.

Подытоживая, model имеет смысл использовать в двух случаях

1. Spreadsheet-like calculations.

   То есть вычисление значений ячеек с помощью выражений, использующих значения других ячеек.

Простейшие выражение часто могут быть выражены с помощью аналитических функций и/или дополнительных соединений.

Например, имеется информация по продажам за 12 месяцев и надо для каждого месяца вычислить отношение объема продаж к значению для первого месяца.

С model это будет выглядеть так

```
with t as
(select rownum id, 100 + rownum - 1 value from dual connect by level <= 12)
select *
from t
model
dimension by (id)
measures (value, 0 ratio)
rules
(ratio[any] order by id = value[cv(id)]/value[1])
```

В то же время запрос элементарно переписывается через аналитические функции

```
select id, value, value / first_value(value) over() ratio from t
```

Или более синтетический пример: подсчитать отношение значения в текущей строке к значению из строки, указанной в ref_id.

```
exec dbms_random.seed(100);
create table t as
select rownum id,
       100 + rownum - 1 value,
       trunc(dbms_random.value(1, 10 + 1)) ref_id
  from dual
connect by level <= 10;
```

Решение с помощью model выглядит следующим образом

```
select *
from t
model
dimension by (id)
measures (value, ref_id, 0 ratio)
rules
(ratio[any] order by id = round(value[cv(id)]/value[ref_id[cv(id)]],3));
```

ID	VALUE	REF_ID	RATIO
1	100	6	.952
2	101	7	.953
3	102	7	.962
4	103	8	.963
5	104	3	1.02
6	105	5	1.01
7	106	10	.972
8	107	4	1.039
9	108	2	1.069
10	109	7	1.028

Тем не менее, тот же результат элементарно можно получить с помощью self join (или даже без него если применить подход описанный в главе про аналитические функции)

```
select t1.*, round(t1.value / t2.value, 3) ratio
  from t t1
  join t t2
    on t1.ref_id = t2.id
 order by t1.id
```

Однако более сложные выражения могут потребовать множества соединений и интенсивного применения аналитики/группировки и других приемов, тогда как с помощью model могут быть выражены одним компактным правилом. В некоторых случаях если в запросе фигурирует, например, несколько self join и группировка, то в случае применения model это всё заменяется на набор простых правил.

Стоит отметить, что model имеет проблемы с масштабируемостью, в результате чего, даже если решение выглядит компактно, надо протестировать его на реальных объемах и рассмотреть другие подходы, включая реализацию на PL/SQL.

С другой стороны, стоит отметить, что клиентские приложения для spreadsheet-like calculations, такие как Excel, не предусмотрены для работы с очень большими объемами. Например, максимально возможное число строк Excel 2016 – 1M, а с таким количеством строк

model clause справляется достаточно неплохо, при этом, все вычисления могут быть вычислены на сервере без передачи данных на клиентский компьютер.

2. Получения сложного результата, который не может быть получен никакими иными методами без помощи model, при этом имеются только привилегии select – типичный случай для внешних отчетных систем. Как альтернативные подходы в подобном случае может быть рассмотрено опять же решение с использованием PL/SQL или вычисления в клиентском приложении. Иногда применяется для материализованных представлений, когда есть дополнительная цель избежать PL/SQL кода и создания вспомогательных SQL типов для коллекций.

Часто model используют там где ее использовать не стоит

- Генерация последовательностей независимых значений – для этого лучше использовать connect by.
- Генерация последовательностей зависимых значений – имеет смысл рассмотреть recursive subquery factoring, если генерируемый элемент зависит от фиксированного числа уже сгенерированных.
- Разделение строки на слова и прочая обработка строковых значений вплоть до вычисления арифметического выражения в строке. Для разделения строк лучше опять же использовать connect by, для более сложных манипуляций лучше реализовать функцию на PL/SQL.
- Определение последовательностей определенного вида в наборе данных. Для этого лучше использовать аналитические функции или pattern matching, если это возможно.
- Подсчет итогов. Для этого лучше использовать group by rollup/grouping sets/cube.
- Транспонирование. Для этого есть операторы pivot/unpivot.
- Все иные случаи, когда можно обойтись без нее. ☺

Brief analysis of the performance

Особенность работы конструкции в том, что набор данных, на основании которого строится многомерная модель, целиком загружается в память. Число столбцов результата вполне конкретное – это объединение столбцов для мер и измерений, а число строк может быть как больше так и меньше числа строк набора данных во фразе from. Но, так или иначе, model достаточно прожорлива к потреблению и памяти и ресурсов процессора.

В книге уже не раз упоминалась задача с генерацией последовательности, в которой текущий член зависит от синуса предыдущего (первый раз логика описана в главе про иерархические запросы). Сравним производительность для разной длины последовательности и трех разных подходов: recursive subquery factoring, model и PL/SQL.

Для PL/SQL функции выполним следующий запрос с числом элементов 1e5, 2e5, 3e5, 4e5, 5e5, 1e6

```
select sum(value(t)) result from table(f(1e5)) t;
```

Аналогично посчитаем сумму элементов последовательности с помощью recursive subquery factoring и model.

Для всех трех подходов основное время расходуется на CPU и на моем ноутбуке результаты следующие:

Число элементов	PL/SQL	Recursive with	Model
1e5	01.18	02.29	03.22
2e5	02.43	04.52	12.68
3e5	03.47	07.58	27.93
4e5	04.70	10.31	53.45
5e5	05.82	12.85	01:18.57
1e6	11.80	27.32	05:01.87

Кроме того, расходуется довольно много PGA памяти, с другой стороны ее было достаточно, соответственно временное табличное пространство не было задействовано. Для recursive subquery factoring и model память процесса попадает в категорию SQL, для PL/SQL соответственно в PL/SQL (значение v$process_memory.category).

Дополнительную информацию для SQL подходов можно посмотреть в v$sql_workarea_active.operation_type. Это будет операции SPREADSHEET и CONNECT-BY (SORT) соответственно.

Динамику роста PGA можно анализировать по v$active_session_history.pga_allocated, а детальный анализ можно проводить на основании v$process_memory_detail.

Как видно по результатам подход model показал не только худшее время работы, но и совершенно нелинейный рост от числа элементов. Это не значит, что model всегда проигрывает recursive subquery factoring. Бывают задачи, кода необходимо «итеративно обработать» набор данных без генерации новых строк, в этом случае model может оказаться эффективнее, поскольку при recursive subquery factoring на каждой итерации (или уровне) добавляются новые строки в результат. Так же model предпочтительнее в плане SQL решений, когда необходимо обращаться к результатам полученным на нескольких предыдущих итерациях, но PL/SQL подход, как правило, для таких задач оказывается быстрее.

Говоря про нелинейное увеличение времени при увеличении объема данных важно понимать, что является причиной.

Выполним запрос ниже для правила с использованием аналитической функции, затем для агрегатной функции. Потом изменим число элементов с 1e6 до 1.2e6. Прежде всего, замечу, что тот же результат можно получить без использования конструкции model и запрос ниже приводится исключительно для анализа производительности этой конструкции.

```
select *
from
(select *
from (select rownum id from dual connect by rownum <= 1e6) t
model
dimension by (id)
measures (id value, 0 result)
(
  -- analytical version
  result[any] = sum(value) over (order by id desc)
  -- aggregate version
  -- result[any] = sum(value)[id >= cv(id)]
)
order by id
)
where rownum <= 3;

        ID      VALUE           RESULT
---------- ---------- ----------------
         1          1     500000500000
         2          2     500000499999
         3          3     500000499997
```

Для 1e6 строк для обоих вариантов время работы было около 4-х секунд - это не удивительно, потому что для обоих запросов идентичный план. При увеличении числа строк на 20% до 1.2e6 время выполнения для обоих вариантов увеличилось до 8-ми секунд.

Причиной послужило то, что при увеличении числа строк, для workarea стало не хватать памяти и стало использоваться временное табличное пространство. Это можно увидеть, выполнив запрос ниже для соответствующих sql_Id.

```
select pga_allocated / (1024 * 1024) pga_mb,
       temp_space_allocated / (1024 * 1024) temp_mb,
       ash.*
  from v$active_session_history ash
 where sql_id = '<sql_id>'
 order by sample_time desc
```

Ситуацию можно исправить, если указать ручное управление workarea и указать большой объем памяти для сортировки (я указал максимально возможный – 2GB)

```
alter session set workarea_size_policy = manual;
alter session set sort_area_size = 2147483647;
```

В этом случае выполнение займет 5 секунд, то есть время линейно возрастет при росте числа элементов. Ручное управление памятью может порождать нежелательные побочные эффекты, так что его по возможности стоит избегать.

Последнее, что стоит отметить – время выполнения запроса без model для 1.2e6 записей и с настройками памяти по умолчанию – 2 секунды. Еще раз подчеркиваю, не стоит конструкцию model употреблять тогда, когда можно обойтись без нее.

```
select *
from
(select t.*, sum(id) over (order by id desc) result
from (select rownum id from dual connect by rownum <= 1.2e6) t
order by id
)
where rownum <= 3;
```

В качестве итога по производительности можно выделить следующие моменты

- При использовании конструкции model интенсивно используется память pga – для различных workarea операций. Это будет всегда операция «SPREADSHEET», но могут быть и другие, например «WINDOW (SORT)». Время выполнения коррелирует с потребляемым объемом памяти для workarea.
- Для вычислений правил и обработки результата в workarea будет активно использоваться CPU при большом числе строк в model.
- Конечно же, всегда стоит анализировать план запроса. При использовании любых нетривиальных правил, требующих дополнительных ресурсов – это будет отражено в плане. Например, для аналитических/агрегатных функций в плане будут отражены дополнительные сортировки.
- Ну и в некоторых случаях хорошее улучшение производительности дает parallel execution и модели с указанием partition by – это детальнее рассмотрено в следующем разделе.

Model parallel execution

Для оценки влияния степени параллельности на выполнение конструкции model рассмотрим следующую задачу: необходимо для каждой секции подсчитать нарастающий итог, при этом сбрасывая его после превышения некоторого предела.

Скрипт создания таблицы ниже

```
create table t (part int, id int, value int);
begin
  for i in 1 .. 80 loop
    dbms_random.seed(i);
    insert into t
      select i, rownum id, trunc(dbms_random.value(1, 1000 + 1)) value
        from dual
      connect by rownum <= 1e5;
  end loop;
  commit;
end;
/
```

Запрос с model может быть написан следующим образом

```
select --+ parallel(2)
*
from t
model
partition by (part)
dimension by (id)
measures (value, 0 x, 0 sid)
rules
(
  x[any] order by id = case when cv(id)=1 then value[cv(id)]
                            when x[cv(id)-1] > 3e3 then value[cv(id)]
                            else x[cv(id)-1] + value[cv(id)]
                       end,
  sid[any] order by id = userenv('sid')
)
```

Ту же логику можно реализовать на PL/SQL с возможностью распараллеливания, если создать конвейерную функцию. Для этого потребуется создать объектный тип и коллекцию, а также и строго-типизированный ref cursor – это необходимо для возможности распараллеливания по конкретной колонке.

```
create or replace type to_3int as object (part int, x int, sid int)
/
create or replace type tt_3int as table of to_3int
/
create or replace package pkg as type refcur_t is ref cursor return
t%rowtype; end;
/
```

Реализация на PL/SQL ниже

```
create or replace function f_running(p in pkg.refcur_t) return tt_3int
  pipelined
  parallel_enable(partition p by hash(part)) order p by(part, id) is
  rec  p%rowtype;
  prev p%rowtype;
  x    int := 0;
begin
  loop
    fetch p
      into rec;
    exit when p%notfound;
    if rec.id = 1 then
      x := rec.value;
    elsif x > 3e3 then
      x := rec.value;
    else
      x := x + rec.value;
    end if;
    pipe row(to_3int(rec.part, x, userenv('sid')));
    prev := rec;
  end loop;
  return;
end;
/
```

В обоих случаях в качестве границы для сбрасывания нарастающего итога была использована константа 3e3 или 3000.

Тестирование было проведено на сервере с 80 CPU, чтоб лучше отследить влияние параллелизма на производительность. Для тестирования PL/SQL подхода был выполнен следующим запрос с различными степенями параллельности (DOP – degree of parallelism).

```
select count(distinct sid) c, sum(x*part) s
  from table(f_running(cursor(select /*+ parallel(2) */ * from t)));
```

Аналогично для тестирования model, вместо выражения table() была использована inline view с model.

Статистика выполнения приведена ниже:

DOP	Actual DOP	PL/SQL	Model
Serial	1	01:47.37	53.34
4	4	36.59	15.83
10	10	19.78	08.72
20	19	16.22	05.24
40	34	18.72	04.35

Видно, что выполнение model быстрее и без параллельности, но при использовании параллельности ускоряется быстрее, чем PL/SQL подход и при DOP равном 20, model превосходит процедурный подход уже в три раза. При увеличении DOP до 40, производительность процедурного подхода не улучшается, а наоборот ухудшается, то есть накладные расходы превышают выгоду.

Важно отметить, что при тестировании генераторов на PL/SQL и model, PL/SQL подход был в выигрыше. Это связано с тем, что там использовалась коллекция чисел, а в этом примере – коллекция объектов. Oracle вынужден тратить довольно много CPU ресурсов для конструирования объекта для каждой строки. Кроме того, в предыдущем примере использовались функции sin и round, в текущем – только операции сравнения, присваивания и для одной из веток условного оператора – сложение. Чем больше ресурсов требует логика, тем значительнее нивелируется эффект от конструирования объектного типа для результирующих строк.

Еще один ключевой момент – использование не только параллельности при выполнении, но и указание partition в конструкции model. Это позволяет более эффективно обрабатывать набор данных, поскольку обработка большого набора данных может быть декомпозирована на независимые поднаборы – при использовании partition они обрабатываются изолированно и невозможно заглянуть из одной секции в другую.

Последнее, на что можно обратить внимание – актуальная степень параллелизма была одинаковая для обоих подходов, но это не удивительно, поскольку, в конце концов, она указывалась для SQL движка.

Резюме

Конструкция model наиболее мощный инструмент языка SQL. Итеративные модели вообще теоретически позволяют реализовать алгоритм любой сложности (дополнительная информация может быть найдена в главе «Turing completeness»). Но с другой стороны эта конструкция очень затратная в плане ресурсов – как CPU так и оперативной памяти и не линейно масштабируема для тех задач, где производительность иных подходов линейно возрастает при увеличении размера набора данных.

При использовании partitions и распараллеливания, model может выступать отличным инструментом для определенных задач, даже учитывая производительность.

Также использовать конструкцию имеет смысл, если требуется выполнить spreadsheet-like computations, что иначе бы потребовало написания сложных запросов в соединениями или в экзотических случаях, когда надо реализовать достаточно хитрую логику не прибегая к PL/SQL и созданию хранимых объектов, либо когда производительность не очень критична и цель ограничить реализацию языком SQL – например, для инкапсуляции логики в materialized view.

8. Row Pattern Matching: match_recognize

Это последняя из специальных конструкций языка и перед ее описанием вспомним кратко эволюцию SQL.

Базовый SQL (тот, который описывает пять основных операций реляционной алгебры) допускает видимость данных в пределах строки.

С добавлением групповых функций допускается видимость в пределах группы, но группа определяется по конкретному выражению (оно равно для всех строк группы) и каждая строка относится ровно к одной группе.

Аналитические функции позволяют достичь видимости в пределах окна. Определение окна одинаковое для всех строк, хотя и имеет некоторую гибкость в случае указания range/row. Ключевой момент, что определение окна одинаково для всех строк.

Pattern matching предоставляет следующий уровень гибкости при обработке наборов данных. Последовательность строк анализируется как цепочка на предмет соответствия некоторому шаблону. Каждая строка может относиться к одному или более совпадению (группе) или не подходить под шаблон вовсе.

Воспользуемся таблицей atm из главы «Analytic Functions». Вернуть все строки, для которых amount равен пяти, очевидно можно запросом

```
select * from atm where amount = 5
```

С помощью конструкции match_recognize эту же задачу можно решить так

```
select *
from atm
match_recognize
( all rows per match
  pattern (five)
  define
    five as five.amount = 5
) mr
order by ts;

TS            AMOUNT
--------- ----------
03-JUL-16          5
03-JUL-16          5
```

Но это явно не то, для чего предназначена конструкция, и например, если бы по столбцу amount был уникальный индекс, он бы все равно не использовался (тем не менее, он может быть использован, если в запросе указать where и match_recognize). Вместо этого можно увидеть такой план (FINITE AUTOMATON не влезло целиком в название операции в выводе dbms_xplan)

```
-------------------------------------------------------------------
| Id  | Operation                                          | Name |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT                                   |      |
|   1 |  SORT ORDER BY                                     |      |
|   2 |   VIEW                                             |      |
|   3 |    MATCH RECOGNIZE BUFFER DETERMINISTIC FINITE AU|      |
|   4 |     TABLE ACCESS FULL                              | ATM  |
-------------------------------------------------------------------
```

Рассмотрим чуть более сложный пример

```
alter session set NLS_DATE_FORMAT = 'mi';

Session altered.

select *
from atm
match_recognize
( order by ts
  measures
    strt.amount start_amount,
    final last(up.amount) end_amount,
    running count(*) as cnt,
    match_number() as match,
    classifier() as cls
  all rows per match
  after match skip past last row
  pattern (strt down* up*)
  define
    down as down.amount < prev(down.amount),
    up as up.amount > prev(up.amount)
) mr
order by ts;
```

TS	START_AMOUNT	END_AMOUNT	CNT	MATCH	CLS	AMOUNT
01	85	100	1	1	STRT	85
03	85	100	2	1	DOWN	15
05	85	100	3	1	UP	100
07	40	85	1	2	STRT	40
09	40	85	2	2	DOWN	30
11	40	85	3	2	UP	50
13	40	85	4	2	UP	85
15	60	100	1	3	STRT	60
17	60	100	2	3	DOWN	5
19	60	100	3	3	UP	100
21	25	80	1	4	STRT	25
23	25	80	2	4	UP	30
25	25	80	3	4	UP	80
27	5	35	1	5	STRT	5
29	5	35	2	5	UP	35

Здесь шаблон определен следующий образом: одна строка с меткой strt, ноль или более строк с меткой down и ноль и более строк с меткой up. Строка помечается down, когда значение amount меньше чем у предыдущей строки, и соответственно up – если значение больше. Если визуализировать зависимость amount от ts, то под шаблон попадут V-образные участки графика, либо же только возрастающая или убывающая их часть при отсутствии второй.

Фраза «`all rows per match`» - означает, что в результат попадают все строки, попавшие под шаблон – это поведение по умолчанию. Вместо этого может быть указано «`one row per match`», тогда на каждое совпадение будет выведена одна строка. То есть в первом случае pattern matching работает по аналогии с аналитическими функциями, а во втором – с групповыми.

Функция «`match_number`» показывает номер совпадения, а «`classifier`» - показывает под какое правило попала строка. За исключением match_number, все остальные вычисления в разделе «measures» происходят в рамках совпавшей группы.

Выражение «running count(*)» использовано для нумерации строк в рамках группы. Можно было бы использовать «final count(*)» для отображения общего числа строк в группе. Аналогично конструкция «final last(up.amount)» означает, что для всех строк группы отображается последнее (максимальное) значение, соответствующее правилу up.

После того как найдена последовательность строк, соответствующих шаблону поиск следующей последовательность начинается со строки идущей за последней помеченной строкой – «after match skip past last row». Это поведение может быть изменено, так что поиск строк для следующей совпадающей группы может начинаться с некоторой строки уже совпавшей группы, в результате чего одна и та же строка может принадлежать более чем одной группе. Поиск текущего шаблона не может начинаться с той же строки что и поиск предыдущего, иначе будет ошибка «ORA-62517: Next match starts at the same point last match started». В граничном случае поиск может начинаться со строки идущей за первой помеченной строкой.

Однако эти же группы и классификаторы можно достаточно просто получить с помощью аналитических функций

```
select ts,
       amount,
       count(decode(cls, 'STRT', 1)) over(order by ts) match,
       cls
  from (select ts,
               amount,
               case
                 when lag(cls) over(order by ts) = 'UP' and cls <> 'UP' then
                  'STRT'
                 else
                  cls
               end cls
          from (select atm.*,
                       nvl(case
                             when amount < lag(amount) over(order by ts) then
                              'DOWN'
                             when amount > lag(amount) over(order by ts) then
                              'UP'
                           end,
                           'STRT') cls
                  from atm))
 order by ts;
```

Если изменить шаблон, так чтоб в каждой группе была как минимум одна строка с убывающим amount и одна с возрастающим – «strt down+ up+», то будут такие строки, которые не попадут под шаблон. Если они тем не менее нужны в результате, то в шаблоне можно указать альтернативу – «strt down+ up+|dummy+?». Как альтернативный вариант – можно было бы использовать конструкцию «all rows per match **with unmatched rows**».

```
select *
from atm
match_recognize
( order by ts
  measures
    strt.amount start_amount,
    final last(up.amount) end_amount,
    running count(*) as cnt,
    match_number() as match,
    classifier() as cls
  all rows per match
  after match skip past last row
  pattern (strt down+ up+|dummy+?)
  define
    down as down.amount < prev(down.amount),
    up as up.amount > prev(up.amount)
) mr
order by ts;
```

TS	START_AMOUNT	END_AMOUNT	CNT	MATCH	CLS	AMOUNT
01	85	100	1	1	STRT	85
03	85	100	2	1	DOWN	15
05	85	100	3	1	UP	100
07	40	85	1	2	STRT	40
09	40	85	2	2	DOWN	30
11	40	85	3	2	UP	50
13	40	85	4	2	UP	85
15	60	100	1	3	STRT	60
17	60	100	2	3	DOWN	5
19	60	100	3	3	UP	100
21			1	4	DUMMY	25
23			1	5	DUMMY	30
25	80	35	1	6	STRT	80
27	80	35	2	6	DOWN	5
29	80	35	3	6	UP	35

Этот пример тоже может быть переписан с помощью аналитических функций.

Попробуем реализовать запрос, определяющий числа Фибоначчи в последовательности

```
with t as (select rownum id from dual connect by rownum <= 55)
select * from t
match_recognize
( order by id
  all rows per match
  pattern ((fib|{-dummy-})+)
  define fib as (id = 1 or id = 2 or id = last(fib.id, 1) + last(fib.id,
2)));
```

ID
1
2
3
5
8
13
21
34

Здесь есть несколько интересных моментов. Функции типа last в define (так же как и в measures) работают в рамках совпавшей группы. Это значит, что если для любой строки мы ходит обращаться к двум предыдущим «помеченным» строкам, то вся последовательность должна попадать в одну группу. Для того, чтобы группа не прервалась и учесть числа не Фибоначчи в шаблоне использована альтернатива. Синтаксис {--} означает, что строки, помеченные этим правилом в результат не попадут, хоть они и попали под шаблон. Ну и главный момент, что строки были сгенерированы предварительно, а с помощью конструкции match_recognize только отмечены интересующие.

Этот пример уже не может быть реализован с помощью аналитических функций, поскольку отмечая строки, мы учитываем значения уже отмеченных.

Рассмотрим следующую задачу. Есть таблица, содержащая цифры от 0 до 9.

```
exec dbms_random.seed(1);
create table digit as
select rownum id, trunc(dbms_random.value(0, 9 + 1)) value
  from dual
connect by rownum <= 2e6;
```

Необходимо найти все возможные комбинации идущих подряд цифр 1, 2 и 3.

С помощью row pattern matching задачу можно решить так

```
select decode(v_id, v1_id, 1, v2_id, 2, v3_id, 3) v1,
       decode(v_id + 1, v1_id, 1, v2_id, 2, v3_id, 3) v2,
       decode(v_id + 2, v1_id, 1, v2_id, 2, v3_id, 3) v3,
       count(*) cnt
  from digit
match_recognize
( order by id
  measures
    least(v1.id, v2.id, v3.id) v_id,
    (v1.id) v1_id,
    (v2.id) v2_id,
    (v3.id) v3_id
  one row per match
  after match skip to next row
  pattern (permute (v1, v2, v3))
  define
    v1 as v1.value = 1,
    v2 as v2.value = 2,
    v3 as v3.value = 3)
 group by decode(v_id, v1_id, 1, v2_id, 2, v3_id, 3),
          decode(v_id + 1, v1_id, 1, v2_id, 2, v3_id, 3),
          decode(v_id + 2, v1_id, 1, v2_id, 2, v3_id, 3)
 order by 1, 2, 3;
```

```
        V1         V2         V3        CNT
---------- ---------- ---------- ----------
         1          2          3       2066
         1          3          2       1945
         2          1          3       2027
         2          3          1       1971
         3          1          2       1962
         3          2          1       2015
```

В отличие от предыдущих примеров здесь использовано «one row per match», чтоб «схлопнуть» три строки, попавшие в шаблон, в одну. Для того, чтоб учесть все комбинации использованы ключевые слова «after match skip to next row». Например, для id на отрезке (709, 719) встречается две пересекающиеся последовательности: 1, 3, 2 и 3, 2, 1. Строки с id равным 715 и 716 учтены в двух различных группах.

```
select * from digit where id between 709 and 719;
```

```
        ID      VALUE
---------- ----------
       709          9
       710          3
       711          2
       712          4
       713          6
       714          1
       715          3
       716          2
       717          1
       718          5
       719          0
```

И последняя деталь – использовано ключевое слово «permute». Оно означает что шаблон – это все возможные перестановки v1, v2, v3. Чтоб определить какая из перестановок попала под шаблон использована логика с least и decode.

Эту же задачу можно решить, скомбинировав аналитические и групповые функции.

```
select v1, v2, v3, count(*) cnt
  from (select row_number() over(order by id) rn,
               value v3,
               lag(value, 1) over(order by id) v2,
               lag(value, 2) over(order by id) v1
          from digit)
 where rn > 2
   and v1 in (1, 2, 3)
   and v2 in (1, 2, 3)
   and v3 in (1, 2, 3)
   and v1 <> v2
   and v1 <> v3
   and v2 <> v3
 group by v1, v2, v3
 order by 1, 2, 3;
```

Посмотрим планы запросов с помощью (колонка Starts всегда равна 1 и вырезана вручную)

```
select * from table(dbms_xplan.display_cursor(format => 'IOSTATS LAST'));
```

```
-------------------------------------------------------------------------------------------
| Id  | Operation            | Name  | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |       |        |      6 |00:00:06.10 |    3708 |   3705 |
|   1 |  SORT GROUP BY       |       |     10 |      6 |00:00:06.10 |    3708 |   3705 |
|*  2 |   VIEW               |       |   2000K|  11986 |00:00:06.10 |    3708 |   3705 |
|   3 |    WINDOW SORT       |       |   2000K|   2000K|00:00:04.97 |    3708 |   3705 |
|   4 |     TABLE ACCESS FULL| DIGIT |   2000K|   2000K|00:00:00.46 |    3708 |   3705 |
-------------------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------------------
| Id  | Operation              | Name  | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |       |        |      6 |00:00:03.27 |    3708 |   3705 |
|   1 |  SORT GROUP BY         |       |   2000K|      6 |00:00:03.27 |    3708 |   3705 |
|   2 |   VIEW                 |       |   2000K|  11986 |00:00:03.26 |    3708 |   3705 |
|   3 |    MATCH RECOGNIZE SORT|       |   2000K|  11986 |00:00:03.25 |    3708 |   3705 |
|   4 |     TABLE ACCESS FULL  | DIGIT |   2000K|   2000K|00:00:00.45 |    3708 |   3705 |
-------------------------------------------------------------------------------------------
```

Как видно на выполнение row pattern matching потребовалось меньше времени, чем на выполнение аналитики, не говоря уже про группировку.

Из ограничений аналитических функций можно выделить следующие:

1) В разделах define/measures можно указывать только ограниченный набор агрегатных функции. Например, нельзя использовать listagg или UDAG.
   ORA-62512: This aggregate is not yet supported in MATCH_RECOGNIZE clause
   Дополнительные особенности при работе с агрегатными функциями можно найти в задаче «resemblance group».
2) Как уже было сказано, с помощью функций, подобных prev/next можно обращаться к совпавшим строкам только текущей группы. В функциях можно использовать смещение, но оно задается константой.

Резюме

Row pattern matching расширяет возможности SQL для анализа наборов данных. Эта возможность позволяет проводить достаточно сложный анализ, который иначе потребовал бы аналитических и групповых функций, соединений и подзапросов. В некоторых случаях использование match_recogzine – это единственная возможность получить необходимый результат с с помощью SQL (не беря во внимание конструкции model и recursive subquery factoring, которые предназначены для иных целей и плохо масштабируются). Даже в тех случаях, когда match_recognize может быть достаточно просто заменена на аналитику, она показывает лучшую производительность. Можно провести аналогию с конструкциями pivot/unpivot, которые могут быть заменены на cross join/group by соответственно, но показывают чуть лучшую производительность чем «старые методы». Если в дополнение к хорошей производительности вспомнить очень гибкий regular expression-like синтаксис задания шаблона (раздел pattern) и правил (раздел define), то это делает match_recogzine поистине незаменимой в некоторых случаях и однозначно полезным нововведением.

9. Logical execution order of query clauses

В запросе могут присутствовать множество разнообразных конструкций, начиная от соединения, фильтрации и группировки и заканчивая model, match_recognize и pivot/unpivot. Речь пойдет про логический порядок, в котором выполняются разные части запроса. Для простоты предполагается

1) Отсутствие трансформаций запросов. Например, distinct placement, который влияет на фактический порядок выполнения конструкции distinct.
2) Отсутствие различных эвристик, которые опять же, когда это возможно, могут менять порядок применений конструкций с целью оптимизации. Например, эвристика «выполнять selection как можно раньше», которая приводит к тому, что предикат фильтрации where по внутренней таблице выполняется до соединения.

В общем виде схема может выглядеть следующим образом

1. from (join)
2. where
3. connect by
4. group by
5. having
6. аналитические функции
7. select-list (distinct, scalar sub-queries и так далее)
8. order by

Про особенности обработки первых трех шагов уже было сказано в главе «Hierarchical queries: connect by».

Рассмотрим конкретный пример

```
select --+ rule
 id,
 count(*) cnt,
 max(level) max_lvl,
 max(rownum) max_rn,
 sum(id + count(*)) over(order by id) summ
from (select column_value id from table(sys.odcinumberlist(0, 0, 1)))
 group by id
 start with id = 0
connect by prior id + 1 = id;

        ID        CNT    MAX_LVL     MAX_RN       SUMM
---------- ---------- ---------- ---------- ----------
         0          2          1          3          2
         1          2          2          4          5
```

Сначала было построено дерево 0, 1, 0, 1 (четыре строки, два уровня) и сгенерированы значения псевдостолбцов rownum и level. Потом была произведена группировка и в последнюю очередь была применена аналитическая функция.

В запросе был использован хинт rule, чтоб подчеркнуть, что возможные трансформации CBO не интересуют, так что на план они влиять не будут.

Обратите внимание, что, например, аналитические функции выполняются после group by, но перед distinct. Именно из-за порядка выполнения distinct не всегда может быть заменен на group by без создания вспомогательных inline views.

Создадим следующую таблицу для демонстрации (таблица t была создана в первой главе)

```
create table tt as
select rownum id, mod(rownum,2) value
  from dual connect by level <= 3;
```

С точки зрения результата и логики работы два следующих запроса эквивалентны

```
select distinct value from tt
select value from tt group by value
```

Однако если запрос содержит distinct и скалярные подзапросы, аналитические функции или некоторые другие специальные функции, то его нельзя переписать через group by без дополнительной inline view, поскольку эти конструкции вычисляются после группировки и не могут быть в ней указаны. Например:

```
select distinct row_number() over (partition by id order by null) rn, value
from tt;

        RN      VALUE
---------- ----------
         1          0
         1          1

select distinct (select count(*) from t) cnt, value from tt;

       CNT      VALUE
---------- ----------
         1          0
         1          1

select distinct sys_connect_by_path(value,'->') path, value from tt connect
by 1 = 0;

PATH            VALUE
---------- ----------
->1                 1
->0                 0
```

Интересна также комбинация аналитических и агрегатных функций. Второй запрос возвращает одну строку, поскольку аналитическая функция применяется уже после группировки.

```
select count(*) over() cnt1
  from (select column_value id from table(sys.odcinumberlist(1, 1)));

      CNT1
----------
         2
         2
```

```
select count(*) over() cnt1, count(*) cnt2
  from (select column_value id from table(sys.odcinumberlist(1, 1)));

      CNT1       CNT2
---------- ----------
         1          2
```

Или чуть более сложный пример

```
select value,
       count(*) agg,
       count(*) over() an,
       sum(count(*)) over(order by count(*)) agg_an
  from tt
 group by value;

     VALUE        AGG         AN     AGG_AN
---------- ---------- ---------- ----------
         0          1          2          1
         1          2          2          3
```

Групповые функции могут быть вложенными.

Начнем с такого примера

```
select (id) i1,
       max(id) i2,
       max(id) keep(dense_rank first order by rn desc) i3,
       sum(id) s,
       count(*) cnt,
       listagg(id, ',') within group(order by id) list
  from (select column_value id, rownum rn
          from table(sys.odcinumberlist(1, 2, 3, 5, 2)))
 group by id;

        I1         I2         I3          S        CNT LIST
---------- ---------- ---------- ---------- ---------- ----------
         1          1          1          1          1 1
         2          2          2          4          2 2,2
         3          3          3          3          1 3
         5          5          5          5          1 5
```

Поскольку группировка по id, то к этому полю можно применить и агрегатные функции, но значение max, очевидно всегда будет совпадать со значением самого поля.

Теперь уберем первую колонку в результате, а для i3 укажем «`order by max(rn)`».

```
select max(id) i2,
       max(id) keep(dense_rank first order by max(rn) desc) i3,
       sum(id) s,
       count(*) cnt,
       listagg(id, ',') within group(order by id) list
  from (select column_value id, rownum rn
          from table(sys.odcinumberlist(1, 2, 3, 5, 2)))
 group by id;

        I2         I3          S        CNT LIST
---------- ---------- ---------- ---------- ----------
         5          2         11          4 1,2,3,5
```

В этом случае происходит сначала группировка по id с получением max(rn), а затем еще одна группировка для получения итоговой строки. С дополнительной inline view это выглядело бы так (производительность и механика работы в этом случае абсолютно идентичная случая со вложенными агрегатами)

```
select max(id) i2,
       max(id) keep(dense_rank first order by max_rn desc) i3,
       sum(id) s,
       count(*) cnt,
       listagg(id, ',') within group(order by id) list
  from (select id, max(rn) max_rn
          from (select column_value id, rownum rn
                  from table(sys.odcinumberlist(1, 2, 3, 5, 2)))
         group by id);
```

Более простой пример ниже. В первом случае выполняется конкатенация всех строк, а во втором сначала выполняется группировка по id, чтоб избавить от дублей и затем конкатенация.

```
select listagg(id, ',') within group(order by id) list
  from (select column_value id, rownum rn
          from table(sys.odcinumberlist(1, 2, 3, 5, 2)));

LIST
----------
1,2,2,3,5

select listagg(max(id), ',') within group(order by max(id)) list
  from (select column_value id, rownum rn
          from table(sys.odcinumberlist(1, 2, 3, 5, 2)))
group by id;

LIST
----------
1,2,3,5
```

При вложенности групповых функций может быть не более одной строки в результате запроса и вложенность групповых функций более двух уровней, соответственно, не имеет смысла.

Бывает без inline view не обойтись, например, когда надо отфильтровать результат запроса по значению аналитической функции, поскольку аналитика вычисляется после конструкции where.

Однако в тех случаях, когда можно обойтись без inline view, иногда бывает смысл его использовать для того, чтоб запрос был более читаемый. Некоторые разработчики стремятся использовать как можно меньше inline views, но на самом деле их наличие может не играть совершенно никакой роли. Это лишь конструкция языка, наличие которой в изначальном запросе невозможно определить по плану если inline view mergeable. Более того, в некоторых случаях наличие inline view наоборот может улучшить производительность.

Создадим функцию, время выполнения которой очень близко к одной секунде.

```
create or replace function f return number is
begin
  dbms_lock.sleep(1);
  return 1;
end f;
/
```

Первый запрос ниже выполнился за 6 секунд, потому что функция вычислялась дважды для каждой строки. Второй выполнился за две секунды, поскольку функция вычислилась дважды для первой строки и далее, благодаря scalar subquery caching больше не вычислялась, и в последнем случае запрос выполнился всего за секунду, поскольку скаляр был вычислен один раз в inline view.

```
select id, value, f + 1 f1, f - 1 f2 from tt t;

        ID      VALUE         F1         F2
---------- ---------- ---------- ----------
         1          1          2          0
         2          0          2          0
         3          1          2          0

Elapsed: 00:00:06.04
select id, value, (select f from dual) + 1 f1, (select f from dual) - 1 f2
from tt t;

        ID      VALUE         F1         F2
---------- ---------- ---------- ----------
         1          1          2          0
         2          0          2          0
         3          1          2          0

Elapsed: 00:00:02.02
select id, value, ff + 1 f1, ff - 1 f2 from (select tt.*, (select f from
dual) ff from tt) t;

        ID      VALUE         F1         F2
---------- ---------- ---------- ----------
         1          1          2          0
         2          0          2          0
         3          1          2          0

Elapsed: 00:00:01.02
```

На одном уровне могут встречаться даже специфические конструкции Oracle, например, match_recognize и model. При этом сначала выполнится martch_recognize, затем model и каждая из конструкций не имеет никакой информации, как был получен набор данных на предыдущем шаге, для нее это просто набор строк.

Резюме

В главе был кратко рассмотрен порядок выполнения разных конструкций в запросе и некоторые примеры обработки различных конструкции на одном уровне. В сложных случаях, для простоты поддержки кода, логику можно вынести на разные уровни запроса даже если это необязательно. В некоторых случаях, тем не менее, inline view избежать нельзя – например where + analytic function. Inline view также могут быть полезны для обхода багов (на предыдущих версиях было много таковых, например, в случае connect by + analytical functions на одном уровне запроса), для контроля или ограничения трансформаций (особенно в случае искусственного создания non-mergeable inline view) и для улучшения производительности, как было показано выше.

10. Turing completeness

В теории вычислимости, исполнитель (множество вычисляющих элементов) называется Тьюринг-полным, если он может быть использован для симуляции любой одно-ленточной машины Тьюринга. Примеры исполнителей: набор инструкций микропроцессора, язык программирования, клеточный автомат. Согласно тезису Чёрча — Тьюринга: любая функция, которая может быть вычислена физическим устройством, может быть вычислена машиной Тьюринга. Иными словами, если некий исполнитель может симулировать машину Тьюринга, то он может быть использован для вычисления любой вычислимой функции.

Существует клеточный автомат Правило 110, который является Тьюринг-полным. Доказательство этого факта может быть найдено в [7]. Соответственно, если Правило 110 реализуемо на некотором языке, то он является Тьюринг-полным.

В простейших клеточных автоматах одномерный массив нулей и единиц обновляется, следуя набору простых правил. Значение клетки на следующем шаге зависит от значений клеток-соседей на текущем шаге и значения самой клетки. Для Правила 110 имеет место следующий набор правил:

Текущее состояние	111	110	101	100	011	010	001	000
Новое состояние центральной клетки	0	1	1	0	1	1	1	0

Для первого символа сосед слева – это последний символ, а для последнего сосед справа – первый.

Правило называется 110, поскольку при переводе 01101110 из двоичной системы в десятичную получается это число.

Пример работы клеточного автомата для 19 шагов с изначальной лентой «00000000001000000000000010000» показан ниже

```
      PART STR
---------- ----------------------------------------
         1 00000000001000000000000010000
         2 00000000011000000000000110000
         3 00000000111000000000001110000
         4 00000001101000000000011010000
         5 00000011111000000000111110000
         6 00000110001000000000100010000
         7 00001110011000000000110010000
         8 00011010111000000001101011110000
         9 00111111101000000111111010000
        10 01100001110000001100000111000
        11 11100001101000011100001101000
        12 10100011111000110100011111000
        13 11100110001001111100110001001
        14 00101110011011000101110011011
        15 01111010111111001111010111111
        16 11001111100001011001111100001
        17 11011000100011111011000100011
        18 01111001100110001111001100110
        19 11001011101110011001011101110
        20 11011110111010111011110110101
```

Это достаточно просто реализуется с помощью конструкции recursive subquery factoring.

```
with t0 as
 (select '00000000001000000000000010000' str from dual),
t1 as
 (select 1 part, rownum rn, substr(str, rownum, 1) x
    from t0
  connect by substr(str, rownum, 1) is not null),
t2(part, rn, x) as
 (select part, rn, cast(x as char(1))
    from t1
  union all
  select part + 1,
         rn,
         case nvl(lag(x) over(order by rn),
              last_value(x) over(order by rn rows
              between current row and unbounded following))
              || x ||
              nvl(lead(x) over(order by rn),
              first_value(x) over(order by rn rows
              between unbounded preceding and current row))
           when '111' then '0'
           when '110' then '1'
           when '101' then '1'
           when '100' then '0'
           when '011' then '1'
           when '010' then '1'
           when '001' then '1'
           else
            '0'
         end
    from t2
   where part < 20)
select part, listagg(x) within group(order by rn) str
  from t2
 group by part
 order by 1;
```

Без использования аналитики и при текущих ограничениях, я считаю, что recursive subquery factoring, является неполным по Тьюрингу. Ибо нельзя использовать в рекурсивном члене запрос, определенный with более одного раза, а также не допускается его использование в подзапросах и в качестве внешнего набора во внешних соединениях. Всё это делает невозможным «увидеть соседей по ленте» для произвольной ее длины. Подчеркну, что это утверждение без доказательства, следовательно, вопрос нельзя считать закрытым.

После того, как доказано, что этот автомат может быть использован для реализации произвольного алгоритма, может возникнуть вопрос: как его использовать, например, хотя бы для сложения двух чисел. Для этого входную последовательность символов 1 и 0 следует трактовать как программу и данные и специальным образом ее составить. То есть алгоритм пишется не на SQL, а кодируется во входной ленте (строке).

Последнее что стоит отметить касательно правила 110, оно достаточно просто реализуется с помощью конструкции model, как с итерациями, так и без них – для произвольного числа шагов и произвольной длины входной ленты.

Подобные имитации крайне медленные и могут быть использованы только для академического интереса.

Конструкция model с академической точки зрения интересна еще тем, что с ее помощью можно элементарно реализовать любой алгоритм, если избавиться от вложенных циклов – теоретически это всегда можно сделать.

Рассмотрим сортировку пузырьком. Базовый алгоритм выглядит следующим образом:

```
declare
  s varchar2(4000) := 'abcd c*de 01';
  n number := length(s);
  j number := 1;
  k number := 1;
  x number := 1;
  i number := 1;
begin
  while x > 0 loop
    x := 0;
    for j in 1 .. n - k loop
      i := i + 1;
      if substr(s, j + 1, 1) < substr(s, j, 1) then
        s := substr(s, 1, j - 1) || substr(s, j + 1, 1) ||
             substr(s, j, 1) || substr(s, j + 2);
        x := 1;
      end if;
    end loop;
    k := k + 1;
  end loop;
  dbms_output.put_line(i || s);
end;
```

Мы повторяем вложенные циклы до тех пор, пока есть хотя бы одна замена за цикл – определяется флагом x.

При использовании единственного while цикла, пришлось ввести вспомогательный счетчик – c. Он выступает аналогом x из первого варианта, сама же переменная x всегда равна единице и обнуляется только в конце «внутреннего прохода», если не было ни одной замены (то есть c=0)

```
declare
  s varchar2(4000) := 'abcd c*de 01';
  n number := length(s);
  j number := 1;
  k number := 1;
  x number := 1;
  i number := 1;
  c number := 0;
begin
  while x > 0 loop
    i := i + 1;
    c := case when substr(s, j + 1, 1) < substr(s, j, 1)
             then 1
             else case when j = 1 then 0 else c end
         end;
    s := case when substr(s, j + 1, 1) < substr(s, j, 1)
             then substr(s, 1, j - 1) || substr(s, j + 1, 1) ||
                  substr(s, j, 1) || substr(s, j + 2)
             else s
         end;
    x := case when j = n - k and c = 0 then 0 else 1 end;
    k := case when j = n - k then k + 1 else k end;
    j := case when j - 1 = n - k then 1 else j + 1 end;
  end loop;
  dbms_output.put_line(i || s);
end;
```

Теперь, чтобы реализовать данный алгоритм с помощью конструкции, достаточно объявить необходимые переменные (столбцы), заменить присваивания на равенства (:= -> =), точку с запятой в конце строк на запятую для разделения правил (; -> ,) и добавить [0] для адресации – работа выполняется над набором данных из одной строки, идентифицируемой rn = 0.

Во всех трех случаях результат одинаков и получается за 64 итерации.

```
with t as (select 'abcd c*de 01' s from dual)
select i, s
from t
model
dimension by (0 rn)
measures (length(s) n, 1 j, 1 k, 1 x, 1 i, 0 c, s)
rules iterate(1e9) until x[0]=0
(
    i[0] = i[0] + 1,
    c[0] = case when substr(s[0], j[0] + 1, 1) < substr(s[0], j[0], 1)
                then 1
                else case when j[0] = 1 then 0 else c[0] end
           end,
    s[0] = case when substr(s[0], j[0] + 1, 1) < substr(s[0], j[0], 1)
                then substr(s[0], 1, j[0] - 1) || substr(s[0], j[0] + 1, 1)
||
                     substr(s[0], j[0], 1) || substr(s[0], j[0] + 2)
                else s[0]
           end,
    x[0] = case when j[0] = n[0] - k[0] and c[0] = 0 then 0 else 1 end,
    k[0] = case when j[0] = n[0] - k[0] then k[0] + 1 else k[0] end,
    j[0] = case when j[0] - 1 = n[0] - k[0] then 1 else j[0] + 1 end
);

         I S
---------- ------------
        64   *01abccdde
```

Резюме

Было продемонстрировано, что recursive subquery factoring вместе с аналитическими функциями делают язык SQL полным по Тьюрингу. Более того, было показано, как с помощью конструкции model можно реализовать практически любой алгоритм. Однако, несмотря на имеющуюся мощь, язык SQL это не подходящий инструмент для итеративных вычислений. Более того, как было показано ранее, recursive subquery factoring и model даже для тривиальных алгоритмов часто проигрывают PL/SQL в производительности, особенно при увеличении объемов данных.

Summary

В этой главе хотелось бы подвести итоги по расширенным возможностям SQL в Oracle. Номер в списке ниже соответствует главе книги в первой части

1) Любые соединения таблиц могут быть выражены явно в запросе, однако иногда имеет смысл использовать подзапросы, например, для получения ANTI/EQUI в плане. Коррелированные скалярные подзапросы могут быть производительнее outer joins из-за эффекта scalar subquery caching.
2) Всегда надо иметь в виду, что Oracle трансформирует запрос и два, на первый взгляд, совершенно разных запроса могут иметь один и тот же план и производительность. С другой стороны важно помнить, что трансформации это не панацея и не для всякого ручного переписывания запроса имеется соответствующая трансформация.
3) Аналитические функции – полезнейший инструмент, позволяющий избежать дополнительных соединений. С другой стороны, они почти всегда требуют сортировки, что на больших объемах данных может создавать проблемы.
4) Групповые функции – позволяют «схлопнуть» набор данных в группы и вычислить агрегированные значения.
5) Connect by удачный инструмент для построения древовидных структур или генерации списков. Тем не менее, не стоит его использовать при работе с графами, не смотря на встроенные механизмы обработки циклов.
6) Recursive subquery factoring несколько расширяет возможности при работе с древовидными структурами и списками, позволяя работать с данными, вычисленными на предыдущем уровне. При итеративной обработке набора данных с помощью этого инструмента важно учитывать, что очень неэффективно используется память, поскольку хранятся и возвращаются наборы данных на всех итерациях. Функциональным преимуществом по сравнению с model является то, что на каждой итерации можно вычислять несколько значений (мер). В случае с model сначала вычисляется правило для одной меры, потом для другой и так далее.
7) Model – наиболее мощная конструкция, но весьма узкоспециализированная. Требует много как процессорных ресурсов, так и памяти, очень плохо масштабируется с ростом объема данных и на относительно больших объемах (более 1M строк) почти всегда проиграет решению на PL/SQL. То же относится и к recursive subquery factoring. Но производительность model в редких случаях может быть значительно улучшена за счет конструкции partition и параллелизма.
8) Row pattern matching вносит дополнительную гибкость при анализе наборов данных. Конструкция не только демонстрируют чуть лучшую производительность в тех задачах, которые решаются аналитическими функциями, но и позволяет решать задачи, которые аналитическими функциями или другими базовыми возможностями не решаемы.
9) Запрос состоит из набора конструкций и может содержать аналитические и групповые функции, псевдостолбцы и ключевые слова distinct на одном уровне. Важно понимать в каком порядке идет выполнение на одном уровне и особенности использования многоуровневых запросов.

10) Было показано, что на языке SQL можно реализовать практически любой алгоритм. Но следует помнить что SQL – декларативный язык и его использование для итеративных вычислений целесообразно в очень ограниченных случаях.

Бытует утверждение, проповедуемое Томом Кайтом, что если задачу можно решить на SQL, то ее надо решать на SQL. Это далеко не всегда так. Даже если исключить из SQL возможности recursive subquery factoring, model и connect by для работы с циклическими структурами, то все равно есть определенные задачи, которые более эффективно решаются на PL/SQL. Перечисленные же инструменты почти всегда будут проигрывать PL/SQL решениям и при росте объема данных преимущество PL/SQL будет все больше. Возможно, в следующих версиях Oracle несколько оптимизирует затраты CPU и памяти для собственных расширений языка SQL, но принципиальных изменений в производительности здесь ждать не стоит.

Part II

1. When PL/SQL is better than vanilla SQL

Ранее уже неоднократно упоминалось, что при использовании специфических конструкций, таких как, например, model или recursive subquery factoring, ту же логику для получения набора данных можно реализовать на PL/SQL с получением существенного выигрыша в производительности. Однако прибегать к использованию PL/SQL при получении наборов данных с целью повышения производительности может быть целесообразно даже если результат может быть получен исключительно с помощью базовых возможностей SQL. Как правило, это связано либо с ограничениями текущей реализации SQL либо с особенностями работы SQL запросов. Язык SQL декларативный и то, что происходит под капотом можно контролировать лишь до определенного предела. Далее попробуем систематизировать такие случаи, однако сразу стоит уточнить, что один и тот же случай можно отнести к нескольким категориям.

Specifics of analytical functions

Аналитические функции – мощнейший инструмент и их появление оказало огромное влияние на возможности SQL, но они имеют логические ограничения (были описаны в главе про аналитические функции) так и некоторые особенности в реализации на которые повлиять нельзя.

Прерывание выборки

Основа первой проблемы, описываемой в этой главе в невозможности в общем случае эффективно указать в запросе, что необходимо возвращать строки до наступления некоторого условия. А аналитические функции – лишь инструмент, который может быть использован для достижения цели при помощи SQL, но не всегда эффективным образом.

Рассмотрим ситуацию, когда необходимо «прервать» выборку или остановить выдачу результата, основываясь на некотором условии. Например, есть таблица с транзакциями, и необходимо выдать все последние транзакции пока их сумма не превысит X (или пока не будет возвращено N *определенных* строк).

Сходу можно придумать три разных подхода

- использование аналитической функции
- реализация логики в табличной функции
- вынести логику на сторону клиента и «фетчить набор данных вручную»

Итак, допустим, имеется следующая таблица

```
exec dbms_random.seed(1);
create table transaction(id int not null, value number not null);
insert --+ append
into transaction
select rownum, trunc(1000 * dbms_random.value + 1) value
  from dual
connect by rownum <= 3e6;
create index idx_tran_id on transaction(id);
```

```
exec dbms_stats.gather_table_stats(user, 'transaction');
```

Тесты проведены на версии 12.1.0.2 с

1) Включенной runtime execution statistics
   ```
   alter session set statistics_level = all;
   ```
2) Выключенными adaptive plans
   ```
   alter session set "_optimizer_adaptive_plans" = false;
   ```

Планы получены с помощью команды «`select * from table(dbms_xplan.display_cursor(null,null,format => 'IOSTATS LAST'))`». IOSTATS было использовано вместо ALLSTATS для того, чтоб планы помещались на странице по ширине. Статистику использования памяти можно посмотреть путем указания MEMSTATS или ALLSTATS (либо запросами из системных представлений).

Сначала рассмотрим более простую задачу, когда необходимо получить 10 последних транзакций.

Первый способ - использование сортировки в inline view и фильтра по rownum.

```
select *
  from (select * from transaction order by id desc)
 where rownum <= 10;

        ID      VALUE
---------- ----------
   3000000        875
   2999999        890
   2999998        266
   2999997        337
   2999996        570
   2999995        889
   2999994        425
   2999993         64
   2999992        140
   2999991        638

10 rows selected.
```

Запрос возвращает результат практически мгновенно – менее сотой доли секунды

```
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name        | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |             |      1 |        |     10 |00:00:00.01 |       7 |
|*  1 |  COUNT STOPKEY                |             |      1 |        |     10 |00:00:00.01 |       7 |
|   2 |   VIEW                        |             |      1 |     10 |     10 |00:00:00.01 |       7 |
|   3 |    TABLE ACCESS BY INDEX ROWID| TRANSACTION |      1 |   3000K|     10 |00:00:00.01 |       7 |
|   4 |     INDEX FULL SCAN DESCENDING| IDX_TRAN_ID |      1 |     10 |     10 |00:00:00.01 |       4 |
-----------------------------------------------------------------------------------------------------------
```

Это достигается за счет того, что выполняется чтение индекса по убыванию с обращением к таблице для получения value, но главный момент, что чтение останавливается после получения 10 строк.

Теперь реализуем аналогичную логику с использованием аналитических функций

```
select t1.id, t1.value
  from (select row_number() over(order by id desc) rn, t0.*
          from transaction t0) t1
 where rn <= 10;
```

Время выполнения стало более двух секунд (что в разы дольше первого варианта), и это при том, что отсутствует столбец Reads, то есть, все данные были получены из памяти – Buffers.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                | Name        | Starts | E-Rows | A-Rows |   A-Time    | Buffers |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |             |      1 |        |     10 |00:00:02.05 |    6047 |
|*  1 |  VIEW                    |             |      1 |     10 |     10 |00:00:02.05 |    6047 |
|*  2 |   WINDOW SORT PUSHED RANK|             |      1 |   3000K|     10 |00:00:02.05 |    6047 |
|   3 |    TABLE ACCESS FULL     | TRANSACTION |      1 |   3000K|   3000K|00:00:00.46 |    6047 |
-------------------------------------------------------------------------------------------------
```

Операция WINDOW SORT **PUSHED RANK** означает, что сортировка выполняется до получения некоторого числа строк, но при этом, во-первых, происходит чтение всей таблицы, а во-вторых, сортировка, выполняется над всем множеством строк (хотя порядок обеспечивается только для первых RN строк).

Учитывая, что данные в индексе упорядочены, этим можно воспользоваться и при использовании аналитики, если прибегнуть к дополнительному соединению.

```
select t2.*
  from (select --+ cardinality(10) index_desc(t0 idx_tran_id)
          row_number() over(order by id desc) rn, rowid row_id
          from transaction t0) t1
  join transaction t2
    on t1.row_id = t2.rowid
 where t1.rn <= 10;
```

В inline view явно указан метод доступа для того, чтоб гарантированно избежать полного сканирования таблицы или индекса, а во вторых дана подсказка по числу строк, для того, чтоб выполнялся метод соединения nested loops с t2 для получения value.

Запрос выполняется менее сотой доли секунды – так же как и первый вариант.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Starts | E-Rows | A-Rows |   A-Time    | Buffers |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |      1 |        |     10 |00:00:00.01 |       6 |
|   1 |  NESTED LOOPS                |             |      1 |     10 |     10 |00:00:00.01 |       6 |
|*  2 |   VIEW                       |             |      1 |     10 |     10 |00:00:00.01 |       4 |
|*  3 |    WINDOW NOSORT STOPKEY     |             |      1 |     10 |     10 |00:00:00.01 |       4 |
|   4 |     INDEX FULL SCAN DESCENDING| IDX_TRAN_ID |      1 |   3000K|     11 |00:00:00.01 |       4 |
|   5 |   TABLE ACCESS BY USER ROWID | TRANSACTION |     10 |      1 |     10 |00:00:00.01 |       2 |
-------------------------------------------------------------------------------------------------
```

Последнее что стоит отметить относительно простой задачи, желаемый план можно получить, если в качестве ограничения числа строк использована константа, или, например, связываемая переменная. Если же использовать скалярный подзапрос «`(select 10 from dual)`», то получим такую картину.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Starts | E-Rows | A-Rows |   A-Time    | Buffers |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |      1 |        |     10 |00:00:03.37 |    6971 |
|   1 |  NESTED LOOPS                |             |      1 |     10 |     10 |00:00:03.37 |    6971 |
|*  2 |   VIEW                       |             |      1 |     10 |     10 |00:00:03.37 |    6969 |
|   3 |    WINDOW NOSORT             |             |      1 |     10 |   3000K|00:00:02.91 |    6969 |
|   4 |     INDEX FULL SCAN DESCENDING| IDX_TRAN_ID |      1 |   3000K|   3000K|00:00:01.11 |    6969 |
|   5 |    FAST DUAL                 |             |      1 |      1 |      1 |00:00:00.01 |       0 |
|   6 |   TABLE ACCESS BY USER ROWID | TRANSACTION |     10 |      1 |     10 |00:00:00.01 |       2 |
-------------------------------------------------------------------------------------------------
```

Можно заметить, что операция WINDOW NOSORT **STOPKEY** превратилась в WINDOW NOSORT, то есть индекс был прочитан целиком. Соответственно, если ограничение на число строк получается запросом, то вместо использования скаляра лучше разбить один запрос на два и использовать связываемую переменную.

Теперь перейдем к более сложной задаче: необходимо выдавать в результат последние транзакции пока их сумма не достигнет некоторого предела, пусть это будет 5000.

Очевидно, подход с сортировкой и rownum для такого случая уже использовать не удастся. Вариант с аналитикой может выглядеть так:

```
select t1.id, t1.value
  from (select sum(value) over(order by id desc) s, t0.*
          from transaction t0) t1
 where s <= 5000;

        ID      VALUE
---------- ----------
   3000000        875
   2999999        890
   2999998        266
   2999997        337
   2999996        570
   2999995        889
   2999994        425
   2999993         64
   2999992        140

9 rows selected.
```

План выполнения следующий

```
-------------------------------------------------------------------------------------------------------------
| Id  | Operation           | Name        | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  | Writes |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |             |      1 |        |      9 |00:00:13.59 |    6055 |  12746 |  11861 |
|*  1 |  VIEW               |             |      1 |   3000K|      9 |00:00:13.59 |    6055 |  12746 |  11861 |
|   2 |   WINDOW SORT       |             |      1 |   3000K|   3000K|00:00:13.07 |    6055 |  12746 |  11861 |
|   3 |    TABLE ACCESS FULL| TRANSACTION |      1 |   3000K|   3000K|00:00:00.47 |    6047 |      0 |      0 |
-------------------------------------------------------------------------------------------------------------
```

Видно, что потребовалось значительно больше времени, чем при использовании row_number. Это потому, что аналитическая сортировка выполнилась для всего множества (хоть в результате нас интересует всего первые 9 строк). Столбцы Reads/Writes говорят о том, что для сортировки было недостаточно места в памяти и было использовано временное табличное пространство.

Попробуем повторить тест, указав максимально возможный объем памяти для сортировки.

```
alter session set workarea_size_policy = manual;
alter session set sort_area_size = 2147483647;
```

```
-----------------------------------------------------------------------------------------------
| Id  | Operation           | Name        | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |             |      1 |        |      9 |00:00:04.95 |    6047 |
|*  1 |  VIEW               |             |      1 |   3000K|      9 |00:00:04.95 |    6047 |
|   2 |   WINDOW SORT       |             |      1 |   3000K|   3000K|00:00:04.48 |    6047 |
|   3 |    TABLE ACCESS FULL| TRANSACTION |      1 |   3000K|   3000K|00:00:00.44 |    6047 |
-----------------------------------------------------------------------------------------------
```

Исчезли чтение и запись из временного табличного пространства, однако нет никакой необходимости сортировать набор строк для всей таблицы.

Если сделать предположение, что нумерация ID непрерывная (без пропусков), что практически не случается в реальной жизни, то можно использовать такой подход с recursive subquery factoring.

```
with rec(id, value, s) as
(
  select id, value, value
    from transaction
   where id = (select max(id) from transaction)
  union all
  select t.id, t.value, rec.s + t.value
    from transaction t
    join rec on rec.id - 1 = t.id
   where rec.s + t.value <= 5000
)
select * from rec;
```

Время выполнения снова стало одна сотая секунды.

```
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                                | Name        | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |             |      1 |        |      9 |00:00:00.01 |      35 |
|   1 |  VIEW                                    |             |      1 |      2 |      9 |00:00:00.01 |      35 |
|   2 |   UNION ALL (RECURSIVE WITH) BREADTH FIRST|            |      1 |        |      9 |00:00:00.01 |      35 |
|   3 |    TABLE ACCESS BY INDEX ROWID BATCHED   | TRANSACTION |      1 |      1 |      1 |00:00:00.01 |       7 |
|*  4 |     INDEX RANGE SCAN                     | IDX_TRAN_ID |      1 |      1 |      1 |00:00:00.01 |       6 |
|   5 |      SORT AGGREGATE                      |             |      1 |      1 |      1 |00:00:00.01 |       3 |
|   6 |       INDEX FULL SCAN (MIN/MAX)          | IDX_TRAN_ID |      1 |      1 |      1 |00:00:00.01 |       3 |
|   7 |    NESTED LOOPS                          |             |      9 |      1 |      8 |00:00:00.01 |      28 |
|   8 |     NESTED LOOPS                         |             |      9 |      1 |      9 |00:00:00.01 |      19 |
|   9 |      RECURSIVE WITH PUMP                 |             |      9 |        |      9 |00:00:00.01 |       0 |
|* 10 |      INDEX RANGE SCAN                    | IDX_TRAN_ID |      9 |      1 |      9 |00:00:00.01 |      19 |
|* 11 |     TABLE ACCESS BY INDEX ROWID          | TRANSACTION |      9 |      1 |      8 |00:00:00.01 |       9 |
-----------------------------------------------------------------------------------------------------------
```

Для версии 12c можно написать запрос без допущений относительно пропусков для ID (на более старых версиях нельзя использовать lateral views, однако способы обхода этого ограничения указаны в первой главе в разделе «Коррелированные inline view и подзапросы»).

```
with rec(id, value, s) as
(
  select id, value, value
    from transaction
   where id = (select max(id) from transaction)
  union all
  select t.id, t.value, rec.s + t.value
    from rec
   cross apply (select max(id) id from transaction where id < rec.id) t0
    join transaction t on t0.id = t.id
   where rec.s + t.value <= 5000
)
cycle id set c to 1 default 0
select * from rec;
```

```
-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation                                  | Name            | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                           |                 |      1 |        |      9 |00:00:00.01 |      54 |
|   1 |  VIEW                                      |                 |      1 |      2 |      9 |00:00:00.01 |      54 |
|   2 |   UNION ALL (RECURSIVE WITH) BREADTH FIRST |                 |      1 |        |      9 |00:00:00.01 |      54 |
|   3 |    TABLE ACCESS BY INDEX ROWID BATCHED     | TRANSACTION     |      1 |      1 |      1 |00:00:00.01 |       7 |
|*  4 |     INDEX RANGE SCAN                       | IDX_TRAN_ID     |      1 |      1 |      1 |00:00:00.01 |       6 |
|   5 |      SORT AGGREGATE                        |                 |      1 |      1 |      1 |00:00:00.01 |       3 |
|   6 |       INDEX FULL SCAN (MIN/MAX)            | IDX_TRAN_ID     |      1 |      1 |      1 |00:00:00.01 |       3 |
|   7 |    NESTED LOOPS                            |                 |      9 |      1 |      8 |00:00:00.01 |      47 |
|   8 |     NESTED LOOPS                           |                 |      9 |      1 |      9 |00:00:00.01 |      38 |
|   9 |      NESTED LOOPS                          |                 |      9 |      1 |      9 |00:00:00.01 |      19 |
|  10 |       RECURSIVE WITH PUMP                  |                 |      9 |        |      9 |00:00:00.01 |       0 |
|  11 |       VIEW                                 | VW_LAT_EC725798 |      9 |      1 |      9 |00:00:00.01 |      19 |
|  12 |        SORT AGGREGATE                      |                 |      9 |      1 |      9 |00:00:00.01 |      19 |
|  13 |         FIRST ROW                          |                 |      9 |      1 |      9 |00:00:00.01 |      19 |
|* 14 |          INDEX RANGE SCAN (MIN/MAX)        | IDX_TRAN_ID     |      9 |      1 |      9 |00:00:00.01 |      19 |
|* 15 |      INDEX RANGE SCAN                      | IDX_TRAN_ID     |      9 |      1 |      9 |00:00:00.01 |      19 |
|* 16 |     TABLE ACCESS BY INDEX ROWID            | TRANSACTION     |      9 |      1 |      8 |00:00:00.01 |       9 |
-----------------------------------------------------------------------------------------------------------------------
```

Здесь можно обратить внимание, что использована конструкция для обработки циклов. Хотя признак цикла для всех строк 0, однако, без использования этой конструкции – запрос выдавал бы ошибку «ORA-32044: cycle detected while executing recursive WITH query». Это не совсем корректное поведение и этот момент будет еще раз затронут дальше в этой главе.

Итого, на современных версиях recursive subquery factoring позволяет достаточно эффективно решить задачу. Однако, что делать, если необходимо реализовать более хитрую логику останова, нежели просто нарастающий итог, которая не может быть эффективно реализована на SQL или если упомянутая конструкция не поддерживается для используемой версии Oracle.

В таком случае самое оптимальное решение – табличная PL/SQL функция.

Создадим типы

```
create or replace type to_id_value as object(id int, value number)
/
create or replace type tt_id_value as table of to_id_value
/
```

И функцию

```
create or replace function f_transaction(p_limit in number)
  return tt_id_value
  pipelined is
  l_limit number := 0;
begin
  for i in (select --+ index_desc(transaction idx_tran_id)
              *
               from transaction
              order by id desc) loop
    l_limit := l_limit + i.value;
    if l_limit <= 5000 then
      pipe row(to_id_value(i.id, i.value));
    else
      exit;
    end if;
  end loop;
end f_transaction;
/
```

Сделав несколько выполнений можно заметить, что в среднем тратится две сотых секунды.

```
select * from table(f_transaction(p_limit => 5000));

        ID      VALUE
---------- ----------
   3000000        875
   2999999        890
   2999998        266
   2999997        337
   2999996        570
   2999995        889
   2999994        425
   2999993         64
   2999992        140

9 rows selected.

Elapsed: 00:00:00.02
```

Теперь перекомпилируем функцию, добавив в хинт некоторую уникальную последовательность символов «*--+ index_desc(transaction idx_tran_id) zzz*», чтоб можно было легко получить «чистую» статистику из v$sql.

После пары выполнений функции статистика будет следующая

```
column sql_text format a50
select executions, rows_processed, sql_text
  from v$sql v
 where sql_text like '%index_desc(transaction idx_tran_id) zzz%'
   and sql_text not like '%v$sql%';

EXECUTIONS ROWS_PROCESSED SQL_TEXT
---------- -------------- --------------------------------------------------
         2             20 SELECT --+ index_desc(transaction idx_tran_id) zzz
                            * FROM TRANSACTION ORDER BY ID DESC
```

То есть, было обработано (отфетчено) всего 20 строк из курсора, как и ожидалось.

Аналогичный проход по курсору можно было бы сделать в некоторых клиентских приложений, но возможность инкапсулировать всю логику для получения набора данных в СУБД очень важна.

Последнее, что стоит заметить касательно такого типа запросов – в 12c появились так называемые Top-N Queries. Никаких принципиальных изменений в SQL движке для этого сделано не было и если для такого типа запросов посмотреть итоговую версию после трансформаций, то там будут аналитические функции. То есть, по сути, Top-N – это синтаксический сахар и при его использовании добиться оптимальной производительности сложнее, чем без оного. Полагаю, эта функциональность была добавлена по следующим причинам

1) Маркетинговые. В другие СУБД есть такая возможность, теперь в Oracle тоже.
2) Упрощение миграции приложений из других СУБД.
3) Простота написания ad-hoc запросов для пользователей, не имеющих глубоких знаний SQL.

Целесообразность использования этого функционала для сложных запросов критических к производительности весьма сомнительна.

Уход от сортировки

Второй пример касательно аналитических функций будет затрагивать сортировку, порождаемую аналитикой.

Предположим, есть таблица фактов с большим числом строк и измерения с низкой кардинальностью.

```
exec dbms_random.seed(1);
create table fact_a as
select date '2010-01-01' + level / (60 * 24) dt,
       trunc(3 * dbms_random.value()) dim_1_id,
       trunc(3 * dbms_random.value()) dim_2_id,
       trunc(1000 * dbms_random.value()) value
  from dual
connect by level <= 3e6;
exec dbms_stats.gather_table_stats(user, 'fact_a');
```

Цель – посчитать нарастающий итог по каждому из измерений и их комбинации, с сортировкой по дате.

```
select dt,
       dim_1_id,
       dim_2_id,
       value,
       sum(val) over(partition by dim_1_id order by dt) dim1_sum,
       sum(val) over(partition by dim_2_id order by dt) dim2_sum,
       sum(val) over(partition by dim_1_id, dim_2_id order by dt)
dim1_dim2_sum
  from fact_a
 order by dt
```

Для дальнейших тестов будем использовать несколько модифицированный запрос, чтоб минимизировать fetch – всего одна строка в результате.

```
select to_char(sum(dim1_sum), lpad('9', 20, '9')) d1,
       to_char(sum(dim2_sum), lpad('9', 20, '9')) d2,
       to_char(sum(dim1_dim2_sum), lpad('9', 20, '9')) d12
  from (select dt,
               dim_1_id,
               dim_2_id,
               value,
               sum(value) over(partition by dim_1_id order by dt) dim1_sum,
               sum(value) over(partition by dim_2_id order by dt) dim2_sum,
               sum(value) over(partition by dim_1_id, dim_2_id order by dt)
dim1_dim2_sum
          from fact_a
         order by dt);

D1                   D2                   D12
-------------------- -------------------- --------------------
      749461709848354       749461230723892       249821726573778
```

В плане имеется три сортировки для аналитики, не смотря на то, что упорядочивание во всех трех функциях по дате. Причина – разные поля в «partition by».

```
-------------------------------------------------------------------------------------------------------
| Id  | Operation               | Name   | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  | Writes |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |        |      1 |        |      1 |00:01:21.85 |    9272 |  96051 |  90717 |
|   1 |  SORT AGGREGATE         |        |      1 |      1 |      1 |00:01:21.85 |    9272 |  96051 |  90717 |
|   2 |   VIEW                  |        |      1 |   3000K|   3000K|00:01:21.06 |    9272 |  96051 |  90717 |
|   3 |    SORT ORDER BY        |        |      1 |   3000K|   3000K|00:01:20.05 |    9272 |  96051 |  90717 |
|   4 |     WINDOW SORT         |        |      1 |   3000K|   3000K|00:01:11.46 |    9267 |  82098 |  76764 |
|   5 |      WINDOW SORT        |        |      1 |   3000K|   3000K|00:00:42.91 |    9257 |  48721 |  45685 |
|   6 |       WINDOW SORT       |        |      1 |   3000K|   3000K|00:00:20.28 |    9249 |  21504 |  20284 |
|   7 |        TABLE ACCESS FULL| FACT_A |      1 |   3000K|   3000K|00:00:00.49 |    9240 |      0 |      0 |
-------------------------------------------------------------------------------------------------------
```

Общее время выполнения составило примерно минута и 20 секунд и, как видно по столбцам Reads/Writes, все три сортировки для аналитики породили чтения/запись во временное табличное пространство.

Попробуем увеличить память для сортировки до максимальной

```
alter session set workarea_size_policy = manual;
alter session set sort_area_size = 2147483647;
```

Ситуация заметно улучшилась – общее время выполнения уменьшилось до 20 секунд и все сортировки выполнялись в памяти.

```
-----------------------------------------------------------------------------------------------
| Id  | Operation               | Name   | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |        |      1 |        |      1 |00:00:21.08 |    9240 |
|   1 |  SORT AGGREGATE         |        |      1 |      1 |      1 |00:00:21.08 |    9240 |
|   2 |   VIEW                  |        |      1 |   3000K|   3000K|00:00:20.41 |    9240 |
|   3 |    SORT ORDER BY        |        |      1 |   3000K|   3000K|00:00:19.59 |    9240 |
|   4 |     WINDOW SORT         |        |      1 |   3000K|   3000K|00:00:17.50 |    9240 |
|   5 |      WINDOW SORT        |        |      1 |   3000K|   3000K|00:00:11.83 |    9240 |
|   6 |       WINDOW SORT       |        |      1 |   3000K|   3000K|00:00:06.58 |    9240 |
|   7 |        TABLE ACCESS FULL| FACT_A |      1 |   3000K|   3000K|00:00:00.47 |    9240 |
-----------------------------------------------------------------------------------------------
```

Учитывая особенности аналитических окон в этом запросе, ту же логику можно реализовать на PL/SQL обойдясь одной сортировкой. Для этого снова понадобиться создать необходимые типы и конвейерную функцию.

```
create or replace function f_fact_a return tt_fact_a
  pipelined is
  type tt1 is table of number index by pls_integer;
  type tt2 is table of tt1 index by pls_integer;
  l_dim1  tt1;
  l_dim2  tt1;
  l_dim12 tt2;
begin
  for r in (select /*+ lvl 0 */ dt, dim_1_id, dim_2_id, value from fact_a order by dt) loop
    -- NoFormat Start
    l_dim1(r.dim_1_id) := case when l_dim1.exists(r.dim_1_id) then l_dim1(r.dim_1_id) else 0 end
+ r.value;
    l_dim2(r.dim_2_id) := case when l_dim2.exists(r.dim_2_id) then l_dim2(r.dim_2_id) else 0 end
+ r.value;
    l_dim12(r.dim_1_id)(r.dim_2_id) := case
                                         when l_dim12.exists(r.dim_1_id) and
l_dim12(r.dim_1_id).exists(r.dim_2_id)
                                         then l_dim12(r.dim_1_id) (r.dim_2_id)
                                         else 0
                                       end + r.value;
    -- NoFormat End
    pipe row(to_fact_a(r.dt,
                       r.dim_1_id,
                       r.dim_2_id,
                       r.value,
                       l_dim1(r.dim_1_id),
                       l_dim2(r.dim_2_id),
                       l_dim12(r.dim_1_id) (r.dim_2_id)));
  end loop;
end;
/
```

```
create or replace type to_fact_a as object
(
  dt            date,
  dim_1_id      number,
  dim_2_id      number,
  value         number,
  dim1_sum      number,
  dim2_sum      number,
  dim1_dim2_sum number
)
/
create or replace type tt_fact_a as table of to_fact_a
/
```

Откомпилируем функцию с выключенной PL/SQL оптимизацией.

```
alter session set plsql_optimize_level = 0;
alter function f_fact_a compile;
set timing on
select to_char(sum(dim1_sum), lpad('9', 20, '9')) d1,
       to_char(sum(dim2_sum), lpad('9', 20, '9')) d2,
       to_char(sum(dim1_dim2_sum), lpad('9', 20, '9')) d12
  from table(f_fact_a)
 order by dt;
```

Среднее время выполнения порядка 45 секунд, то есть хуже SQL варианта.

Теперь изменим значение параметра «`plsql_optimize_level`» на 2 (это значение по умолчанию) и откомпилируем функцию, заменив в коде «`/*+ lvl 0 */`» на «`/*+ lvl 2 */`».

Выполнение стало занимать порядка 12-14 секунд, что лучше SQL версии.

Такая серьезная разница обуславливается тем, что по умолчанию включена оптимизация for loop и fetch выполняется по 100 строк. В этом легко убедиться, посмотрев статистику выполнения в v$sql.

```
select regexp_substr(sql_text, '/.*/') hint,
       executions,
       fetches,
       rows_processed
  from v$sql s
 where sql_text like '%FROM FACT_A%'
   and sql_text not like '%v$sql%';

HINT            EXECUTIONS    FETCHES ROWS_PROCESSED
--------------- ---------- ---------- --------------
/*+ lvl 0 */             2    6000002        6000000
/*+ lvl 2 */             2      60002        6000000
```

Итого, PL/SQL вариант выполняется примерно на 35% быстрее, но большем числе аналитических функций разница была бы более заметна и, главное, при работе с курсором в PL/SQL размер fetch можно было бы увеличить, что дало бы прирост производительности в разы. Дополнительная информация по этому вопросу может быть найдена в работе «Doing SQL from PL/SQL: Best and Worst Practices» [9].

Iterative-like computations

В Oracle SQL есть как минимум два способа для выполнения итеративных вычислений над набором данных: model iterate и recursive subquery factoring. Стоит отметить, что для model iterate, во-первых, все вычисления выполняются над набором данных, и не допускается использование каких-то специальных структур, например, списков или стека, а, во-вторых, счетчик цикла может быть только один (в академических целях реализация сортировки пузырьком с помощью Model приводилась в главе «Turing completeness»). Особенность recursive subquery factoring, что на текущей итерации видны только данные с предыдущей итерации, хотя в результат попадают данные со всех итераций. Коротко говоря, область применения обоих механизмов достаточно узкая, не говоря про далеко не блестящую масштабируемость и интенсивное использование CPU и памяти (эти вопросы были затронуты в соответствующих главах ранее).

В следующей главе будут приведены примеры задач, требующих итеративные вычисления, а также их решения с помощью model/recursive subquery factoring и сравнение с подходами на PL/SQL.

В этом разделе рассмотрим две задачи, которые могут быть решены без упомянутых выше инструментов, но при использовании PL/SQL решаются более эффективно.

Отсутствие желаемых методов доступа

Допустим, есть задача, найти уникальные значения из not nullable колонки с низкой кардинальностью.

```
create table t_str(str varchar2(30) not null, padding varchar2(240));
insert into t_str
select 'AAA', lpad('x', 240, 'x') from dual
union all
select 'BBB', lpad('x', 240, 'x') from dual
union all
select lpad('C', 30, 'C'), lpad('x', 240, 'x') from dual
connect by rownum <= 3e6
union all
select 'DDD', lpad('x', 240, 'x') from dual;
create index t_str_idx on t_str(str);
exec dbms_stats.gather_table_stats(user,'t_str');
```

Тривиальное решение задачи следующее

```
select distinct str from t_str;

STR
------------------------------
BBB
AAA
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
DDD
```

```
-------------------------------------------------------------------------------------------
| Id  | Operation             | Name      | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |           |      1 |        |      4 |00:00:01.04 |   17662 |
|   1 |  HASH UNIQUE          |           |      1 |      4 |      4 |00:00:01.04 |   17662 |
|   2 |   INDEX FAST FULL SCAN| T_STR_IDX |      1 |   3000K|   3000K|00:00:00.53 |   17662 |
-------------------------------------------------------------------------------------------
```

Как видно время выполнения чуть больше секунды, при этом все данные находились в памяти, и было выполнено почти 18к логических чтений.

Если версия Oracle позволяет использовать recursive subquery factoring, то решение может быть следующим

```
with rec(lvl, str) as
(
  select 1, min(str) from t_str
  union all
  select lvl + 1, (select min(str) from t_str where str > rec.str)
    from rec
   where str is not null
)
select * from rec where str is not null;
```

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                                | Name      | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |           |      1 |        |      4 |00:00:00.01 |      13 |
|*  1 |  VIEW                                    |           |      1 |      2 |      4 |00:00:00.01 |      13 |
|   2 |   UNION ALL (RECURSIVE WITH) BREADTH FIRST|           |      1 |        |      5 |00:00:00.01 |      13 |
|   3 |    SORT AGGREGATE                        |           |      1 |      1 |      1 |00:00:00.01 |       3 |
|   4 |     INDEX FULL SCAN (MIN/MAX)            | T_STR_IDX |      1 |      1 |      1 |00:00:00.01 |       3 |
|   5 |    SORT AGGREGATE                        |           |      4 |      1 |      4 |00:00:00.01 |      10 |
|   6 |     FIRST ROW                            |           |      4 |      1 |      3 |00:00:00.01 |      10 |
|*  7 |      INDEX RANGE SCAN (MIN/MAX)          | T_STR_IDX |      4 |      1 |      3 |00:00:00.01 |      10 |
|   8 |    RECURSIVE WITH PUMP                   |           |      5 |        |      4 |00:00:00.01 |       0 |
-----------------------------------------------------------------------------------------------
```

Время выполнения уменьшилось в 100 раз, а число логических чтений более чем в 1000 раз.

Для старых версий Oracle можно было бы применить следующий workaround с помощью PL/SQL.

```
create or replace function f_str return strings
  pipelined is
  l_min t_str.str%type;
begin
  select min(str) into l_min from t_str;
  pipe row(l_min);
  while true loop
    select min(str) into l_min from t_str where str > l_min;
    if l_min is not null then
      pipe row(l_min);
    else
      return;
    end if;
  end loop;
end f_str;
```

Проанализируем производительность с помощью dbms_hprof

```
exec dbms_hprof.start_profiling('UDUMP', '1.trc');

PL/SQL procedure successfully completed.

select column_value str from table(f_str);
```

```
STR
------------------------------
AAA
BBB
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
DDD

exec dbms_hprof.stop_profiling;

PL/SQL procedure successfully completed.

select dbms_hprof.analyze('UDUMP', '1.trc') runid from dual;

     RUNID
----------
         4
```

Статистику выполнения следующая

```
select pci.runid,
       level depth,
       rpad(' ', (level - 1) * 3, ' ') || fi.function as name,
       fi.subtree_elapsed_time,
       fi.function_elapsed_time,
       fi.calls
  from (select runid, parentsymid, childsymid
          from dbmshp_parent_child_info
         union all
         select runid, null, 2 from dbmshp_runs) pci
  join dbmshp_function_info fi
    on pci.runid = fi.runid
   and pci.childsymid = fi.symbolid
   and fi.function <> 'STOP_PROFILING'
connect by prior childsymid = parentsymid
       and prior pci.runid = pci.runid
 start with pci.parentsymid is null
        and pci.runid in (4);
```

RUNID	DEPTH	NAME	SUBTREE_ELAPSED_TIME	FUNCTION_ELAPSED_TIME	CALLS
4	1	__plsql_vm	931	16	3
4	2	__anonymous_block	77	77	1
4	2	F_STR	838	110	2
4	3	__static_sql_exec_**line5**	198	198	1
4	3	__static_sql_exec_**line8**	530	530	4

Как видно 1 раз был выполнен запрос на 5-й строке и 4 раза на 8-й строке функции, общее время составило 931 микросекунду или приблизительно 0.01 секунды так же как и вариант с recursive subquery factoring.

Детальную статистику по SQL можно посмотреть в v$sql, как было показано в предыдущем разделе.

Задачи комбинаторного характера (permutations)

Задача: для таблицы с числами получить сумму их всех возможных комбинаций.

Создадим таблицу следующим образом (особенность данных такова, что сумма любой комбинации чисел уникальна)

```
create table t_num as
select id - 1 id, num
  from (select rownum id, rownum - 1 num from dual connect by level <= 2)
 where id > 1
model
dimension by (id)
measures (num)
(
  num[for id from 3 to 20 increment 1] order by id =
  sum(num)[any] + cv(id)
)
```

Если бы в таблице были только первые три строки,

```
select * from t_num where id < =3;
```

```
        ID        NUM
---------- ----------
         1          1
         2          4
         3          9
```

то были бы такие комбинации чисел

1 + 4 = 5

1 + 9 = 10

4 + 9 = 13

1 + 4 + 9 = 14

и итоговая сумма 5 + 10 + 13 + 14 = 42

На SQL это решается, если сгенерировать все возможные перестановки (как минимум два способа с помощью connect by) и затем для перестановки посчитать сумму.

```
with
t1 as
 (select power(2, rownum-1) row_mask, num from t_num),
t2 as
 (select rownum as total_mask
    from (select count(*) as cnt from t1) connect by rownum < power(2, cnt)
    -- or the same: from t1 connect by num > prior num
 )
select count(*) cnt, sum(num) sum_num
  from (select total_mask as id, sum(num) as num
          from t2, t1
         where bitand(row_mask, total_mask) <> 0
         group by total_mask
        having count(*) > 1);
```

```
       CNT       SUM_NUM
---------- -------------
    524268  343590305814
```

При выполнении порядка 25 секунд, львиная доля ресурсов ушла на nested loops для получения суммы.

```
-----------------------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name  | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  | Writes |
-----------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |       |      1 |        |      1 |00:00:25.81 |    1572K|    868 |    868 |
|   1 |  SORT AGGREGATE                     |       |      1 |      1 |      1 |00:00:25.81 |    1572K|    868 |    868 |
|   2 |   VIEW                              |       |      1 |      1 |    524K|00:00:25.70 |    1572K|    868 |    868 |
|*  3 |    FILTER                           |       |      1 |        |    524K|00:00:25.54 |    1572K|    868 |    868 |
|   4 |     HASH GROUP BY                   |       |      1 |      1 |    524K|00:00:25.36 |    1572K|    868 |    868 |
|   5 |      NESTED LOOPS                   |       |      1 |      1 |   4980K|00:00:22.24 |    1572K|      0 |      0 |
|   6 |       VIEW                          |       |      1 |      1 |    524K|00:00:01.39 |       3 |      0 |      0 |
|   7 |        COUNT                        |       |      1 |        |    524K|00:00:01.17 |       3 |      0 |      0 |
|   8 |         CONNECT BY WITH FILTERING   |       |      1 |        |    524K|00:00:00.98 |       3 |      0 |      0 |
|   9 |          VIEW                       |       |      1 |      1 |      1 |00:00:00.01 |       3 |      0 |      0 |
|  10 |           SORT AGGREGATE            |       |      1 |      1 |      1 |00:00:00.01 |       3 |      0 |      0 |
|  11 |            VIEW                     |       |      1 |     19 |     19 |00:00:00.01 |       3 |      0 |      0 |
|  12 |             COUNT                   |       |      1 |        |     19 |00:00:00.01 |       3 |      0 |      0 |
|  13 |              TABLE ACCESS FULL      | T_NUM |      1 |     19 |     19 |00:00:00.01 |       3 |      0 |      0 |
|* 14 |       VIEW                          |       |    524K|      1 |   4980K|00:00:19.37 |    1572K|      0 |      0 |
|  15 |        COUNT                        |       |    524K|        |   9961K|00:00:10.07 |    1572K|      0 |      0 |
|  16 |         TABLE ACCESS FULL           | T_NUM |    524K|     19 |   9961K|00:00:07.02 |    1572K|      0 |      0 |
-----------------------------------------------------------------------------------------------------------------------
```

Можно попробовать уйти от дополнительного соединения, воспользовавшись sys_connect_by_path и написав функцию для вычисления суммы.

```
select count(*) cnt, sum(f_calc(path)) sum_num
  from (select sys_connect_by_path(num, '+') || '+' as path
          from t_num
         where level > 1
        connect by num > prior num);

create or replace function f_calc(p_str in varchar2) return number is
  pragma udf;
  result number := 0;
  i      int := 0;
  n      varchar2(30);
begin
  while true loop
    i := i + 1;
    n := substr(p_str,
                instr(p_str, '+', 1, i) + 1,
                instr(p_str, '+', 1, i + 1) - instr(p_str, '+', 1, i) - 1);
    if n is not null then
      result := result + n;
    else
      exit;
    end if;
  end loop;
  return result;
end f_calc;
```

При указании «pragma udf» запрос выполняется порядка 7 секунд, без него – порядка 10 секунд, а если вообще закомментировать «sum(f_calc(path))», то менее секунды. По прежнему основанная часть ресурсов уходит на получения суммы для заданной перестановки, однако этот подход заметно быстрее изначального.

Воспользуемся временной таблицей для хранения промежуточных результатов

```
create global temporary table tmp(lvl int, x int, num number);
```

При таком подходе время выполнения всего пол секунды!

```
begin
  insert into tmp (lvl, x, num) select 1, rownum, num from (select num from t_num order by num);
  for c in (select rownum x, num from (select num from t_num order by num))
  loop
    insert into tmp(lvl, x, num) select c.x, 0, tmp.num + c.num from tmp where tmp.x < c.x;
  end loop;
end;
/

PL/SQL procedure successfully completed.

Elapsed: 00:00:00.52
select count(*) cnt, sum(num) sum_num from tmp where lvl > 1;

       CNT       SUM_NUM
---------- -------------
    524268  343590305814

Elapsed: 00:00:00.04
```

Очевидно, SQL подходы значительно уступают по производительности PL/SQL подходу в данном случае. Код наводит на мысль, что аналогичную логику с накопительным вычислением суммы можно реализовать с помощью recursive subquery factoring. Однако здесь есть одна загвоздка, в PL/SQL цикле идет чтение таблицы tmp, которая содержит данные со всех итераций, тогда как в случае recursive subquery factoring видны данные только предыдущей итерации.

```
with
r0(x, num) as
 (select rownum, num from (select num from t_num order by num)),
rec(iter, lvl, x, num) as
 (select 1, 1, rownum, num from r0
  union all
  select rec.iter + 1,
         decode(z.id, 1, rec.lvl, rec.lvl + 1),
         decode(z.id, 1, rec.x, 0),
         decode(z.id, 1, rec.num, rec.num + r0.num)
    from rec
    join r0
      on rec.iter + 1 = r0.x
    join (select 1 id from dual union all select 2 id from dual) z
      on (z.id = 1 or rec.x < r0.x))
select count(*) cnt, sum(num) sum_num
  from rec
 where iter = (select count(*) from t_num)
   and lvl > 1;
```

Здесь использован трюк с дополнительным соединением, для того, чтоб «протянуть» изначальный набор на все итерации. Из-за особенностей текущей реализации, при изменении порядка соединений может быть необходимо указать конструкцию для обработки циклов. То есть

```
…
    from rec
   cross join (select 1 id from dual union all select 2 id from dual) z
    join r0
      on rec.iter + 1 = r0.x
   where (z.id = 1 or rec.x < r0.x))
cycle iter set c to 1 default 0
…
```

Иначе была бы ошибка «ORA-32044: cycle detected while executing recursive WITH query», хотя запросы абсолютно логически эквивалентны. В любом случае, время выполнения запроса порядка полутора минут, то есть, этот вариант абсолютно не конкурент ни подходу с connect by, ни, тем более, PL/SQL подходу. Хотя, казалось бы, PL/SQL цикл крайне примитивен и recursive subquery factoring мог бы пригодиться.

Итого, было продемонстрировано две задачи, в первой итеративность была использована для того, чтоб избежать чтения ненужных данных (или можно сказать «оптимизировать план»), во второй задаче итеративность обусловлена самой постановкой.

Specifics of joins and subqueries

Язык SQL ориентирован на работу с множествами и по факту может быть использовано всего три различных алгоритма, если необходимо соединить два множества – HASH JOIN, NESTED LOOPS, MERGE JOIN. Все они имеют свои плюсы и минусы, а HASH JOIN и MERGE JOIN еще и ограниченную область применения в зависимости от предиката соединения; NESTED LOOPS может быть использован всегда. Касательно подзапросов, далеко не всякая логика может быть реализована на практике, поскольку есть ограничения реализации, например уровни вложенности. Эти и другие моменты будут рассмотрены на конкретных примерах в следующих подразделах.

Особенности соединений

Рассмотрим следующую ситуацию. Есть таблица со списком звонков, содержащая номер и длительность в минутах.

```
create table phone_call (num varchar2(11), duration int);
exec dbms_random.seed(1);
insert --+ append
into phone_call
  select '01' || to_char(trunc(1e9 * dbms_random.value), 'fm099999999'),
         trunc(dbms_random.value(1, 5 + 1))
    from dual
  connect by level <= 1e6;
commit;
exec dbms_stats.gather_table_stats(user, 'phone_call');
```

И таблица со списком кодов и соответствующих направлений. Таблицу можно загрузить по ссылке http://www.area-codes.org.uk/uk-area-codes.xlsx и импортировать вручную или же при наличии необходимых прав просто создать следующим запросом.

```
create table phone_code as
with tbl as
 (select regexp_substr(httpuritype('http://www.area-codes.org.uk/full-uk-
area-code-list.php')
                       .getclob(),
                       '<table class="info">.*?</table>',
                       1,
                       1,
                       'n') c
    from dual)
select *
  from xmltable('/table/tr' passing xmltype((select c from tbl))
                columns
                  code varchar2(6) path '/tr/td[1]',
                  area varchar2(50) path '/tr/td[2]')
 order by 1;
exec dbms_stats.gather_table_stats(user, 'phone_call');
```

Из особенностей данных следует выделить следующие

1) Для простоты все телефоны начинаются с 01, хотя в реальности первые цифры могут быть 01, 02, 03, 05, 07, 08 и 09.
2) Некоторые телефоны недействительны, т.к. для них нет соответствующих префиксов, например, 0119. Это побочный эффект простоты генерации. Данные по таким номерам будут исключены из результата.
3) Номера могут повторяться, что легко наблюдать следующим запросом.

```
select count(*) cnt_all, count(distinct num) cnt_dist from phone_call;

   CNT_ALL   CNT_DIST
---------- ----------
   1000000     999490
```

Относительно кодов стоит отметить, что возможна ситуация когда один код является частью другого, например (в таких случаях префиксом для конкретного номера считается длиннейший)

```
select *
  from phone_code pc1
  join phone_code pc2
    on pc2.code like pc1.code || '%'
   and pc2.code <> pc1.code
 order by 1, 3;

CODE   AREA                                CODE   AREA
------ ----------------------------------- ------ -----------------------------------
01387  Dumfries                            013873 Langholm
01524  Lancaster                           015242 Hornby
01539  Kendal                              015394 Hawkshead
01539  Kendal                              015395 Grange-Over-Sands
01539  Kendal                              015396 Sedbergh
01697  Brampton (6 figure numbers)         016973 Wigton
01697  Brampton (6 figure numbers)         016974 Raughton Head
01697  Brampton (6 figure numbers)         016977 Brampton (4 and 5 figure numbers)
01768  Penrith                             017683 Appleby
01768  Penrith                             017684 Pooley Bridge
01768  Penrith                             017687 Keswick
01946  Whitehaven                          019467 Gosforth

12 rows selected.
```

Возможная длина префиксов, начинающихся с 01 от 3-х до 5-ти цифр

```
select length(code) l, count(*) cnt
  from phone_code
 where code like '01%'
 group by length(code)
 order by 1;

         L        CNT
---------- ----------
         4         12
         5        582
         6         12
```

Для других стран диапазон длин может быть шире и число ситуаций, когда один префикс – часть другого может быть значительно больше.

Итак, задача: получить суммарное число минут по каждому из направлений.

В лоб решение следующее

```
select code, sum(duration) s
  from (select ca.rowid,
               num,
               duration,
               max(code) keep(dense_rank first order by length(code) desc)
code
          from phone_call ca
          left join phone_code co
            on ca.num like co.code || '%'
         group by ca.rowid, num, duration)
 where code is not null
 group by code
 order by code;
```

Далее для простоты будет анализировать несколько модифицированный запрос для краткости результата.

```
select sum(code * sum(duration)) s, count(*) cnt
  from (select ca.rowid,
               num,
               duration,
               max(code) keep(dense_rank first order by length(code) desc)
code
          from phone_call ca
          left join phone_code co
            on ca.num like co.code || '%'
         group by ca.rowid, num, duration)
 group by code;

         S        CNT
---------- ----------
2884843733        607

select * from table(dbms_xplan.display_cursor(null,null,'IOSTATS LAST'));
```

```
-------------------------------------------------------------------------------------------------
| Id  | Operation              | Name       | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |            |      1 |        |      1 |00:01:13.63 |    4002K|
|   1 |  SORT AGGREGATE        |            |      1 |      1 |      1 |00:01:13.63 |    4002K|
|   2 |   HASH GROUP BY        |            |      1 |      1 |    607 |00:01:13.63 |    4002K|
|   3 |    VIEW                |            |      1 |   2000K|   1000K|00:01:13.32 |    4002K|
|   4 |     SORT GROUP BY      |            |      1 |   2000K|   1000K|00:01:13.06 |    4002K|
|   5 |      NESTED LOOPS OUTER|            |      1 |   2000K|   1001K|00:01:10.99 |    4002K|
|   6 |       TABLE ACCESS FULL| PHONE_CALL |      1 |   1000K|   1000K|00:00:00.81 |    2777 |
|*  7 |       TABLE ACCESS FULL| PHONE_CODE |   1000K|      2 |    702K|00:01:09.27 |    4000K|
-------------------------------------------------------------------------------------------------
```

Из плана видно, что все чтения были из памяти и выполнение не потребовало временного табличного пространства (нет столбцов Reads/Writes). Из-за того, что предикат содержит like – единственный возможный метод соединения – NESTED LOOPS. Собственно основная часть времени выполнения и ушла на соединение, последующие группировки добавили менее трех секунд.

Учитывая особенности кодов можно добиться соединения HASH JOIN с таблицей кодов, если добавить вспомогательное соединение.

```
select sum(code * sum(duration)) s, count(*) cnt
  from (select ca.rowid,
               num,
               duration,
               max(code) keep(dense_rank first order by length(code) desc
nulls last) code
          from phone_call ca
         cross join (select rownum + 3 idx from dual connect by rownum <= 3)
x
          left join phone_code co
            on substr(ca.num, 1, x.idx) = co.code
         group by ca.rowid, num, duration)
 group by code;
```

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name       | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |            |      1 |        |      1 |00:00:08.04 |    2781 |
|   1 |  SORT AGGREGATE                    |            |      1 |      1 |      1 |00:00:08.04 |    2781 |
|   2 |   HASH GROUP BY                    |            |      1 |      1 |    607 |00:00:08.04 |    2781 |
|   3 |    VIEW                            |            |      1 |   1000K|   1000K|00:00:07.73 |    2781 |
|   4 |     SORT GROUP BY                  |            |      1 |   1000K|   1000K|00:00:07.47 |    2781 |
|*  5 |      HASH JOIN RIGHT OUTER         |            |      1 |   1000K|   3000K|00:00:03.98 |    2781 |
|   6 |       TABLE ACCESS FULL            | PHONE_CODE |      1 |    611 |    611 |00:00:00.01 |       4 |
|   7 |       MERGE JOIN CARTESIAN         |            |      1 |   1000K|   3000K|00:00:01.50 |    2777 |
|   8 |        VIEW                        |            |      1 |      1 |      3 |00:00:00.01 |       0 |
|   9 |         COUNT                      |            |      1 |        |      3 |00:00:00.01 |       0 |
|  10 |          CONNECT BY WITHOUT FILTERING|          |      1 |        |      3 |00:00:00.01 |       0 |
|  11 |           FAST DUAL                |            |      1 |      1 |      1 |00:00:00.01 |       0 |
|  12 |        BUFFER SORT                 |            |      3 |   1000K|   3000K|00:00:00.83 |    2777 |
|  13 |         TABLE ACCESS FULL          | PHONE_CALL |      1 |   1000K|   1000K|00:00:00.05 |    2777 |
-------------------------------------------------------------------------------------------------
```

Данный подход значительно улучшил производительность – время выполнения упало с 73 секунд до 8 секунд – примерно в 9 раз!

Принцип работы следующий: сначала для телефона генерируются коды направлений от трех до 6 символов (то есть, по три строки на каждую входную строку), потом выполняется соединение с таблицей кодов и выбирается максимальный из тех, что удовлетворяет соединению. Можно заметить, что если соединение прошло для кода из шести символов, то нет никакой необходимости выполнять соединение для того же номера и кодов меньшей длины, но подобное

невозможно регулировать с помощью SQL. Иными словами, соединяя текущую строку невозможно брать во внимание, насколько успешно были соединены предыдущие строки.

Однако соединений можно вообще избежать, если воспользоваться ассоциативным массивом.

```
create or replace package phone_pkg is

  type tp_phone_code is table of int index by varchar2(6);
  g_phone_code tp_phone_code;
  function get_code(p_num in varchar2) return varchar2 deterministic;

end phone_pkg;
/
create or replace package body phone_pkg is

  function get_code(p_num in varchar2) return varchar2 deterministic is
    l_num varchar2(6);
  begin
    l_num := substr(p_num, 1, 6);

    while (l_num is not null) and (not g_phone_code.exists(l_num)) loop
      l_num := substr(l_num, 1, length(l_num) - 1);
    end loop;

    return l_num;
  end;

begin
  for cur in (select * from phone_code) loop
    g_phone_code(cur.code) := 1;
  end loop;
end phone_pkg;
/


select sum(code * sum(duration)) s, count(*) cnt
  from (select ca.rowid, num, duration, phone_pkg.get_code(num) code
          from phone_call ca
         group by ca.rowid, num, duration)
 group by code;

         S        CNT
---------- ----------
2884843733        607

-------------------------------------------------------------------------------------------------
| Id  | Operation            | Name       | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |            |      1 |        |      1 |00:00:06.34 |    2777 |
|   1 |  SORT AGGREGATE      |            |      1 |      1 |      1 |00:00:06.34 |    2777 |
|   2 |   HASH GROUP BY      |            |      1 |      1 |    607 |00:00:06.34 |    2777 |
|   3 |    VIEW              | VM_NWVW_0  |      1 |   1000K|   1000K|00:00:05.71 |    2777 |
|   4 |     HASH GROUP BY    |            |      1 |   1000K|   1000K|00:00:00.92 |    2777 |
|   5 |      TABLE ACCESS FULL| PHONE_CALL |      1 |   1000K|   1000K|00:00:00.06 |    2777 |
-------------------------------------------------------------------------------------------------
```

Как видно, производительно стала еще несколько лучше, однако если добавить «pragma udf», то будет следующая картина.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation             | Name       | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |            |      1 |        |      1 |00:00:03.50 |    2777 |
|   1 |  SORT AGGREGATE       |            |      1 |      1 |      1 |00:00:03.50 |    2777 |
|   2 |   HASH GROUP BY       |            |      1 |      1 |    607 |00:00:03.50 |    2777 |
|   3 |    VIEW               | VM_NWVW_0  |      1 |   1000K|   1000K|00:00:03.03 |    2777 |
|   4 |     HASH GROUP BY     |            |      1 |   1000K|   1000K|00:00:00.85 |    2777 |
|   5 |      TABLE ACCESS FULL| PHONE_CALL |      1 |   1000K|   1000K|00:00:00.05 |    2777 |
-------------------------------------------------------------------------------------------------
```

То есть, выполнение уже в два раза быстрее, чем 8 секунд. При увеличении числа направлений в справочнике, а также расширение диапазона цифр в направлениях PL/SQL подход будет все больше выигрывать.

Ограничения подзапросов

Как уже упоминалось в первой главе, всякий запрос, требующий сопоставления строк двух и более множеств по некоторым критериям, может быть написан с явным использованием соединений. Однако в некоторых случаях имеет смысл использовать, например, скалярные подзапросы. Если для каждой строки набора данных необходимо получить некоторый атрибут согласно замысловатой логике, то использование скалярных подзапросов (и соответственно механизма scalar subquery caching) может быть выгоднее, чем соединение с последующей группировкой, аналитикой или еще более сложной логикой. Однако для реализации логики есть ограничения, например видимость атрибутов основного запроса и максимальный уровень вложенности. В документации не указаны подобные ограничения, однако их достаточно легко продемонстрировать для разных версий.

```
select t1.*,
       (select *
          from (select t2.name
                  from t2 where t2.id = t1.id order by t2.name) t
         where rownum = 1) scalar
  from t1;

        ID N S
---------- - -
         0 X X
         1 A

select t1.*
  from t1
 where exists
 (select 1
          from t2
         where t2.id = t1.id
           and t2.id =
               (select id
                  from (select id from t3 where t3.id = t1.id order by 1)
                 where rownum = 1));

        ID N
---------- -
         0 X
```

Запросы корректно отработали на версии 12c, однако на 11g для обоих была бы ошибка «ORA-00904: "D1"."DUMMY": invalid identifier».

В отличие от скалярных запросов, достаточно сложно придумать, когда есть смысл в подобной вложенности в подзапросах во фразе where. Иногда причина может быть не производительность, а то, что запрос строится динамически и предикаты в where генерируются в зависимости от некоторых условий.

Так или иначе, для того, чтоб обойти подобную ошибку можно реализовать логику подзапроса в UDF. Даже в 12-й версии можно натолкнуться на упомянутые ограничения при использовании скалярных подзапросов, и вместо того, что переписывать логику с использованием соединений и потерей производительности – воспользоваться UDF с указанием «pragma udf».

Если же для примеров выше посмотреть запросы после трансформаций, можно заметить, что была искусственно введена связываемая переменная.

```
select "T1"."ID" "ID",
       "T1"."NAME" "NAME",
       (select "T"."NAME" "NAME"
          from (select "T2"."NAME" "NAME"
                  from "T2" "T2"
                 where "T2"."ID" = :b1
                 order by "T2"."NAME") "T"
         where rownum = 1) "SCALAR"
  from "T1" "T1"

select "T1"."ID" "ID", "T1"."NAME" "NAME"
  from "T1" "T1"
 where exists (select 0
          from "T2" "T2"
         where "T2"."ID" = "T1"."ID"
           and "T2"."ID" = (select "from$_subquery$_003"."ID" "ID"
                              from (select "T3"."ID" "ID"
                                      from "T3" "T3"
                                     where "T3"."ID" = :b1
                                     order by "T3"."ID")
                                     "from$_subquery$_003"
                             where rownum = 1))
```

Ключевое отличие при реализации логики подзапроса в функции – то, что функция возвращает результат на момент ее вызова, а не на момент начала выполнения запроса.

Для демонстрации создадим таблицу и функцию

```
create table t_scn(id, name) as
select 1, 'A' from dual
union all select 2, 'B' from dual
union all select 3, 'C' from dual;

create or replace function f_get_name(p_id in int) return varchar2 is
begin
  dbms_lock.sleep(5);
  for i in (select * from t_scn where id = p_id) loop
    return(i.name);
  end loop;
end f_get_name;
/
```

Во время выполнения запроса

```
select t_main.*,
       (select name from t_scn where id = t_main.id) name1,
       f_get_name(t_main.id) name2
  from t_scn t_main;
```

Выполним в соседней сессии

```
update t_scn set name = 'X' where id = 3;
commit;
```

Результат в основной сессии будет следующим

```
select t_main.*,
       (select name from t_scn where id = t_main.id) name1,
       f_get_name(t_main.id) name2
  from t_scn t_main;

        ID NAME  NAME1 NAME2
---------- ----- ----- -----
         1 A     A     A
         2 B     B     B
         3 C     C     X
```

Для того, чтоб этого избежать можно создать оператор

```
create operator op_get_name binding (int) return varchar2 using f_get_name;
```

При повторении теста результаты будут следующими

```
select t_main.*,
       (select name from t_scn where id = t_main.id) name1,
       f_get_name(t_main.id) name2,
       op_get_name(t_main.id) name3
  from t_scn t_main;

        ID NAME  NAME1 NAME2 NAME3
---------- ----- ----- ----- -----
         1 A     A     A     A
         2 B     B     B     B
         3 C     C     X     C
```

Еще одним достаточно классическим примером ограничений на скалярный подзапрос является тот случай, когда в скалярном подзапросе надо вычислить listagg + distinct. Поскольку listagg не позволяет указывать distinct, то для этого требуется inline view и подобный подход будет работать только начиная с 12c.

Резюме

Были показаны некоторые классы задач, которые эффективнее решаются с использованием PL/SQL, не смотря на то, что результат может быть получен с помощью базовых конструкций SQL. Была использована следующая категоризация

- задачи с использованием аналитики
- задачи, требующие итеративных вычислений
- особенности соединений и подзапросов

Это деление весьма условное, и важно отметить, что при использовании PL/SQL технически применялся один из трех подходов

- Проход по курсору в цикле и построчная обработка на PL/SQL
- Использование в запросе UDF
- Итеративное выполнение операторов SQL

Еще раз стоит обратить внимание, что при построчной обработке с точки зрения производительности очень важен размер fetch.

С появлением конструкции recursive subquery factoring появилась возможность написать более производительные запросы для некоторых специфических задач, однако, по эффективности сопоставимые с PL/SQL подходом. В следующей главе будет показано, как конструкция pattern matching позволяет быстро получить результат, что ранее было доступно только с помощью PL/SQL.

2. Solving SQL quizzes

В последней главе хотелось бы рассмотреть решение некоторых задач с помощью SQL, для того, чтоб лучше раскрыть возможности языка и его специфику. Уровень задач будет достаточно сильно варьироваться так же как и глубина анализа каждой конкретной задачи.

Converting into decimal numeral system

Предположим есть некая строка символов в системе счисления с заданным алфавитом. Задача получить эквивалент в десятичной системе счисления.

Решение

Начнем со случая когда алфавит – шестнадцатеричная система счисления. В такой ситуации можно просто использовать функцию to_number с соответствующей форматной маской.

```
var x varchar2(30)
var alphabet varchar2(30)
exec :alphabet := '0123456789ABCDEF';

PL/SQL procedure successfully completed.

exec :x := '1A0A';

PL/SQL procedure successfully completed.

select to_number(:x, 'XXXX') num from dual;

       NUM
----------
      6666
```

Если реализовать на SQL алгоритм для произвольного алфавита, то это будет выглядеть так

```
select sum(power(base, level - 1) *
           (instr(:alphabet, substr(:x, -level, 1)) - 1)) num
  from (select length(:alphabet) base from dual)
connect by level <= length(:x);

       NUM
----------
      6666
```

Аналогичная реализация на PL/SQL может выглядеть так

```
create or replace function f_10base(p_x        in varchar,
                                    p_alphabet in varchar :=
'0123456789ABCDEF')
  return number is
  result number := 0;
  l_base int := length(p_alphabet);
begin
```

```
  for i in 1 .. length(p_x) loop
    result := result + power(l_base, i - 1) *
              (instr(p_alphabet, substr(p_x, -i, 1)) - 1);
  end loop;
  return result;
end f_10base;
```

Сравним производительность двух подходов

```
select sum(f_10base('ABC' || rownum)) f from dual connect by level <= 1e6;

         F
----------
4.1760E+16

Elapsed: 00:00:16.61

select sum(num) f
  from (select (select sum(power(base, level - 1) *
                           (instr(:alphabet, substr(x, -level, 1)) - 1)) num
                  from (select length(:alphabet) base from dual)
                 connect by level <= length(x)) num
          from (select 'ABC' || rownum x from dual connect by level <= 1e6));

         F
----------
4.1760E+16

Elapsed: 00:00:25.53
```

Как видно, не смотря на переключение контекста, подход с PL/SQL функцией предпочтительнее. Детальнее можно проанализировать куда уходит время если воспользоваться инструментарием описанным в предыдущей главе - dbms_hprof. Добавим для полноты картины вариант с использованием встроенной функцией.

```
select sum(to_number('ABC' || rownum, lpad('X', 10, 'X'))) f
  from dual
connect by level <= 1e6;

         F
----------
4.1760E+16

Elapsed: 00:00:01.11
```

Примерно такой же будет результат, если создать external C function. Эта задача красноречиво демонстрирует, что в некоторых случаях ни SQL ни PL/SQL не являются предпочтительными подходами, если задача критична к производительности.

Connected components

Реляционные СУБД – не лучший инструмент для работы с графами, кроме того графы – очень обширная тема начиная с терминологии и описания различных способов задания и

хранения. Однако задачи на графах периодически встречаются в реальной практике, поэтому кратко затронем эту тему.

Итак, имеется неориентированный (ненаправленный) граф, заданный списком ребер и требуется получить компоненты связности.

Для данных из таблицы ниже

```
create table edge(x1, x2) as
select 10,20 from dual
union all select 50,40 from dual
union all select 20,30 from dual
union all select 20,40 from dual
union all select 60,70 from dual
union all select 80,60 from dual
union all select 20,90 from dual;
```

Ожидается следующий результат (порядок нумерации компонент не критичен)

```
         X        GRP
---------- ----------
        10          1
        20          1
        30          1
        40          1
        50          1
        60          2
        70          2
        80          2
        90          1

9 rows selected.
```

Решение

В лоб решение будет следующим

```
select x, dense_rank() over(order by min(root)) grp
  from (select connect_by_root x1 root, x1, x2
          from edge
         connect by nocycle prior x1 in (x1, x2)
                         or prior x2 in (x1, x2))
 unpivot(x for x12 in(x1, x2))
 group by x
 order by 1, 2;
```

Поскольку граф ненаправленный, то связь X1 -> X2 подразумевает также X2 -> X1, соответственно в connect by были использованы все возможные комбинации соединений ребер. После этого с помощью unpivot все узлы для каждого из корней были получены в одном столбце и для каждого из узлов найден минимальный корень с последующим ранжированием. Несмотря на лаконичность, данный подход экстремально неэффективный.

Учитывая, что задача стоит обойти все вершины и каждой проставить номер компоненты связности, то это можно очень эффективно решить за один проход на PL/SQL. Создадим вспомогательные типы

```
create or replace type to_2int as object (x int, grp int)
/
create or replace type tt_2int as table of to_2int
/
```

И функцию

```
create or replace function f_connected_component return tt_2int
  pipelined is
  i_list number := 0;
  i      number;
  n      number;
  k      number;
  type tp1 is table of binary_integer index by binary_integer;
  type tp2 is table of tp1 index by binary_integer;
  t1 tp1;
  t2 tp2;
begin
  for c in (select x1, x2 from edge) loop
    if not t1.exists(c.x1) and not t1.exists(c.x2) then
      i_list := i_list + 1;
      t1(c.x1) := i_list;
      t1(c.x2) := i_list;
      t2(i_list)(c.x1) := null;
      t2(i_list)(c.x2) := null;
    elsif t1.exists(c.x1) and not t1.exists(c.x2) then
      t1(c.x2) := t1(c.x1);
      t2(t1(c.x1))(c.x2) := null;
    elsif t1.exists(c.x2) and not t1.exists(c.x1) then
      t1(c.x1) := t1(c.x2);
      t2(t1(c.x2))(c.x1) := null;
    elsif t1.exists(c.x1) and t1.exists(c.x2) and t1(c.x1) <> t1(c.x2) then
      n := greatest(t1(c.x1), t1(c.x2));
      k := least(t1(c.x1), t1(c.x2));
      i := t2(n).first;
      while (i is not null) loop
        t2(k)(i) := null;
        t1(i) := k;
        i := t2(n).next(i);
      end loop;
      t2.delete(n);
    end if;
  end loop;

  i := t1.first;
  for idx in 1 .. t1.count loop
    pipe row(to_2int(i, t1(i)));
    i := t1.next(i);
  end loop;
end;
```

Массив t1 содержит для каждой вершины номер компоненты связности. Массив t2 содержит массив компонент, где компонента – это массив вершин. По сути t2 использован для повышения эффективности, поскольку если узлы некоторого ребра принадлежат разным компонентам, необходимо перенумеровать номер компоненты одной из них и в этом сильно помогает список вершин для каждой из компонент.

```
select x, dense_rank() over(order by grp) grp
  from table(f_connected_component)
 order by x;
```

Алгоритмы абсолютно не сопоставимы по эффективности, поэтому не будем углубляться в анализ производительности. Важно отметить, что если стоит задача получить не все компоненты связности, а одну из них начиная с заданной вершины, то полный проход таблицы может быть далеко не лучшим решением, если имеются индексы. В таком случае достаточно эффективен комбинированный подход SQL и PL/SQL, например реализовав обход графа в ширину. Итеративно выполняется поиск смежных вершин для вершин текущего уровня с добавлением не обойденных вершин в результат.

Ordering dependencies

Теперь рассмотрим задачу на ориентированном (направленном) графе.

Есть таблица с зависимостями между объектами, не содержащая циклических зависимостей. Однако между парами вершин может существовать более одного пути, поэтому такую структуру нельзя назвать деревом.

```
create table d(name, referenced_name) as
(select null, 'a' from dual
union all select null, 'd' from dual
union all select 'a', 'b' from dual
union all select 'd', 'b' from dual
union all select 'b', 'e' from dual
union all select 'b', 'c' from dual
union all select 'e', 'c' from dual);
```

Необходимо обойти все объекты за минимальное число шагов, при этом на каждом шаге можно обходить только те объекты, для которых обойдены все зависимые объекты. То есть, на первом шаге обходятся объекты не имеющие зависимостей, на втором шаге те, которые зависят от объектов первого шага и так далее. Иными словами нумеруются зависимости по глубине.

Решение

В лоб с помощью connect by решение может быть следующим

```
select referenced_name, max(level) ord
  from d
 start with not exists
 (select 1 from d d_in where d_in.name = d.referenced_name)
connect by prior name = referenced_name
 group by referenced_name
 order by 2, 1;
```

```
R        ORD
- ----------
c          1
e          2
b          3
a          4
d          4
```

Можно заметить, что в запросе отсутствует конструкция «nocycle» ибо по условию нет циклов. Однако алгоритм не очень эффективен т.к. из каждой вершины без зависимостей выполняется построение цепочки зависимостей в результате некоторые узлы могут обходиться многократно. Здесь нельзя сказать, что выполняется построение пути от листьев, поскольку структура не является деревом. Более эффективно алгоритмически было бы выполнять обход в ширину, но для этого необходимо «накапливать» посещенные узлы, а это невозможно сделать с помощью connect by.

Для «накопления» обойденных объектов воспользуемся глобальной переменной пакета и процедурами инициализации, добавления объекта и получения массива обойденных объектов.

```
create or replace package pkg_d is

  x strings;
  function add_(node in varchar2) return varchar2;
  function get return strings;
  function init return number;

end pkg_d;
/
create or replace package body pkg_d is

  function add_(node in varchar2) return varchar2 as
  begin
    x.extend;
    x(x.count) := node;
    return node;
  end;

  function get return strings as
  begin
    return x;
  end;

  function init return number as
  begin
    x := strings();
    return 1;
  end;

end pkg_d;
/
```

Алгоритм на SQL может быть реализован с помощью recursive subquery factoring следующим образом

```
with rec(node, ord, rn) as
 (select pkg_d.add_(d1.referenced_name), 1 ord,
         row_number() over(partition by d1.referenced_name order by null)
    from d d1
  -- выбираем объекты не имеющие зависимостей
   where not exists (select 1 from d d2 where d1.referenced_name = d2.name)
     and (select pkg_d.init from dual) = 1
  union all
  select pkg_d.add_(d1.name), r.ord + 1,
         row_number() over(partition by d1.name order by null)
    from rec r
  -- предикат rn = 1 использован для того,
  -- чтоб исключить дубли предыдущего шага
    join d d1
      on d1.referenced_name = r.node
     and d1.name is not null
     and r.rn = 1
  -- отсекаем вершины, которые имеют не обойденные зависимые объекты
   where not exists (select 1
                       from d d2
                      where d2.name = d1.name
                        and d2.referenced_name not member pkg_d.get))
select node, ord, rn from rec where rn = 1 order by 2, 1;
```

При наличии индексов по обоим столбам таблицы и сложных зависимостях такой подход может быть эффективнее connect by. Однако наилучшим решением по производительности и сопровождаемости будет PL/SQL реализация с итеративным выполнением SQL запросов пока итерация возвращает хотя бы один новый объект. По сути, это, опять же, обход графа в ширину.

Percentile with shift

Перейдем к задачам на аналитические функции.

Формулировка достаточна проста: получить для каждой строки значение персентиль уровня x, беря в рассмотрение набор строк от текущей и n следующих.

Детальное описание как вычисляется персентиль можно найти в документации Oracle для функции percentile_cont (также для тестов можно использовать функцию Excel - PERCENTILE).

Для значений x = 0.3 и n = 4 результат будет следующим

Порядок	Значение	Персентиль
1	10	64
2	333	95.5
3	100	82
4	55	338.5
5	1000	1000

Решение

Возвращаясь к описанию функции percentile_cont, при использовании ее как аналитической, нельзя указывать windowing_clause, то есть функция применяется к секции (partition) целиком. Поскольку требуется вычислить персентиль по определенному подмножеству строк начиная с текущей, то можно это подмножество получить через self join и воспользоваться агрегатной версией percentile_cont.

```
create table flow(ord, value) as
select 1, 10 from dual
union all select 2, 333 from dual
union all select 3, 100 from dual
union all select 4, 55 from dual
union all select 5, 1000 from dual;

select t1.*, percentile_cont(0.3) within group(order by t2.value) pct
  from flow t1
  join flow t2 on t2.ord between t1.ord and t1.ord + 4
 group by t1.ord, t1.value;
```

ORD	VALUE	PCT
1	10	64
2	333	95.5
3	100	82
4	55	338.5
5	1000	1000

Представим, что функции percentile_cont нет и реализуем логику самостоятельно.

```
select tt.*,
       decode(frn, crn, frn_value,
              (crn - rn) * frn_value + (rn - frn) * crn_value) percentile
  from (select t.ord,
               t.value,
               t.rn,
               t.frn,
               t.crn,
               max(decode(rnum, frn, v)) frn_value,
               max(decode(rnum, crn, v)) crn_value
          from (select t1.*,
                       t2.value v,
                       row_number() over(partition by t1.ord order by t2.value) rnum,
                       1 + 0.3 * (count(*) over(partition by t1.ord) - 1) rn,
                       floor(1 + 0.3 * (count(*) over(partition by t1.ord) - 1)) frn,
                       ceil(1 + 0.3 * (count(*) over(partition by t1.ord) - 1)) crn
                  from flow t1
                  join flow t2
                    on t2.ord between t1.ord and t1.ord + 4) t
         group by t.ord, t.value, t.rn, t.frn, t.crn) tt
 order by tt.ord;
```

ORD	VALUE	RN	FRN	CRN	FRN_VALUE	CRN_VALUE	PERCENTILE
1	10	2.2	2	3	55	100	64
2	333	1.9	1	2	55	100	95.5
3	100	1.6	1	2	55	100	82
4	55	1.3	1	2	55	1000	338.5
5	1000	1	1	1	1000	1000	1000

Здесь сначала вычисляются индексы frn, crn, потом выполняется «поиск» соответствующих им значений из таблицы и выполняется линейная интерполяция.

Наиболее интересно решение выглядит, если значения в исходной таблице упорядочены. Тогда можно обойтись исключительно аналитическими функциями без соединений. Результат запроса ниже отличается от предыдущих поскольку отличаются исходные данные – value упорядочены по ord.

```
select ttt.*,
       decode(frn, crn, frn_value, (crn - rn) * frn_value + (rn - frn) * crn_value) percentile
  from (select tt.*,
               nth_value(value, ord + frn - 1)
               over(order by ord range between unbounded preceding and unbounded following) frn_v,
               nth_value(value, ord + crn - 1)
               over(order by ord range between unbounded preceding and unbounded following) crn_v,
               last_value(value)
               over(order by ord range between frn - 1 following and frn - 1 following) frn_value,
               last_value(value)
               over(order by ord range between crn - 1 following and crn - 1 following) crn_value
          from (select t.*, floor(rn) frn, ceil(rn) crn
                  from (select t0.*,
                               1 +
                               0.3 * (count(*)
                               over(order by ord range between current row and 4 following) - 1) rn
                          from flow t0) t) tt) ttt;
```

ORD	VALUE	RN	FRN	CRN	FRN V	CRN V	FRN VALUE	CRN VALUE	PERCENTILE
1	10	2,2	2	3	55	100	55	100	64
2	55	1,9	1	2	55	100	55	100	95,5
3	100	1,6	1	2	55	100	100	333	239,8
4	333	1,3	1	2	55	100	333	1000	533,1
5	1000	1	1	1	55	100	1000	1000	1000

Поиск значений соответствующих crn, frn для каждой строки выполнен двумя совершенно различными способами – nth_value/last_value, при этом функция nth_value (столбцы crn_v, frn_v) дает неверные результаты для Oracle 11g если второй ее параметр не константа (на 12c этой проблемы нет).

Для этого набора данных можно выполнить исходный запрос с percentile_cont и получить тот же результат, но, как уже было замечено, подход работает только если value упорядочены по ord. Если же нет, то индексы crn, frn найти проблемы не составляет, а вот соответствующие им значения value с помощью аналитических функций найти невозможно – для каждой строки сортируется «свой» набор данных (эта логика реализована в предыдущем запросе как «`row_number() over(partition by t1.ord order by t2.value) rnum`»).

N consequent 1s

Есть таблица

```
exec dbms_random.seed(1);
create table t_sign as
select rownum id,
       case when trunc(dbms_random.value(1, 10 + 1)) > 3
            then 1
            else 0
       end sign
  from dual
connect by rownum <= 1e6;
```

Задача: найти число последовательностей из идущих подряд единиц длины 10. Если имеется, скажем, 11 идущих подряд единиц, то это означает две последовательности: от первой до десятой и от второй до одиннадцатой единицы.

Решение

Далее будут продемонстрированы 4 решения

1) С использованием аналитических функций
2) С использованием аналитических функций и указанием спецификации окна (rows between)
3) Model
4) Pattern matching

Для того, чтоб выполнение было исключительно в памяти предварительно выполнено

```
alter session set workarea_size_policy = manual;
alter session set sort_area_size = 2147483647;
```

Результаты следующие

```
select count(*) cnt
  from (select t.*, sum(sign) over(partition by g order by id) s
          from (select id, sign, sum(x) over(order by id) g
                  from (select t0.*,
                               decode(nvl(lag(sign) over(order by id), -1),
                                      sign,
                                      0,
                                      1) x
                          from t_sign t0)
                 where sign <> 0) t)
 where s >= 10;

       CNT
----------
     28369

Elapsed: 00:00:03.11

select count(*)
  from (select id,
               sum(sign) over(order by id rows between 9 preceding and
current row) s
          from t_sign)
 where s = 10;

  COUNT(*)
----------
     28369

Elapsed: 00:00:01.45

select count(*) cnt from
(
  select *
  from t_sign
```

```
  model
  ignore nav
  dimension by (id)
  measures (sign, 0 s)
  rules
  (
   s[any] order by id = decode(sign[cv()], 0, 0, s[cv()-1]+sign[cv()])
  )
)
where s >= 10;

       CNT
----------
     28369

Elapsed: 00:00:04.64

select count(*)
from t_sign
match_recognize
(
  order by id
  one row per match
  after match skip to first one
  pattern (strt one{9})
  define
     strt as strt.sign = 1,
     one as one.sign = 1
) mr;

  COUNT(*)
----------
     28369

Elapsed: 00:00:01.55
```

То есть лидерами по быстродействию оказались подходы с pattern matching и аналитика с windowing_clause, затем простая аналитика (здесь понадобилось большее число сортировок) и на последнем месте model.

Next value

Для каждого значения из таблицы

```
exec dbms_random.seed(1);
create table t_value as
select trunc(dbms_random.value(1, 1000 + 1)) value
  from dual
connect by level <= 1e5;
```

Получить следующее по величине.

Пример результата:

value	next_value
1	2
2	3
2	3
3	null

Решение

Из-за того, что значения могут повторяться, нельзя просто ограничиться использованием функции lead. Функция lead возвращает значение следующее по порядку, а не по величине.

Далее, как и для предыдущей задачи, будут продемонстрированы 4 решения

5) С использованием аналитических функций
6) С использованием аналитических функций и указанием спецификации окна (range between)
7) Model
8) Pattern matching

Чтоб избежать затрат на fetch использована агрегатная функция «`sum(nvl(next_value, 0) - value)`».

```
select sum(nvl(next_value, 0) - value) s
  from (select value, max(next_value) over(partition by value) next_value
          from (select value,
                       decode(lead(value, 1) over(order by value),
                              value,
                              to_number(null),
                              lead(value, 1) over(order by value)) next_value
                  from t_value));

         S
----------
     69907

Elapsed: 00:00:02.33

select sum(nvl(next_value, 0) - value) s
  from (select value,
               min(value) over(order by value range between 1 following and 1
following) next_value
          from t_value);

         S
----------
     69907
```

```
Elapsed: 00:00:01.79

select sum(nvl(next_value, 0) - value) s
from
(
  select value, next_value
  from t_value
  model
  dimension by (row_number () over (order by value desc) rn)
  measures(value, cast(null as number) next_value)
  rules
  (
    next_value[rn > 1] order by rn =
    decode(value[cv()], value[cv()-1], next_value[cv()-1], value[cv()-1])
  )
);

         S
----------
     69907

Elapsed: 00:00:05.94

select sum(nvl(next_value, 0) - value) s
from (select * from t_value union all select null from dual)
match_recognize
(
  order by value nulls last
  measures
    final first (next_val.value) as next_value
  all rows per match
  after match skip to next_val
  pattern (val+ {-next_val-})
  define
    val as val.value = first(val.value)
) mr;
         S
----------
     69907

Elapsed: 00:00:01.53
```

Как и для предыдущей задачи, лидерами по быстродействию оказались подходы с pattern matching и аналитика с windowing_clause, затем простая аналитика и на последнем месте model. При указании windowing_clause важно указывать минимально необходимый размер окна, так, при указании «`range between 1 following and unbounded following`» результат будет верный, однако время выполнения будет на несколько порядков дольше.

Для решения с pattern matching стоит обратить внимание, что была добавлена одна строка «`union all select null from dual`» для того, чтобы у последнего по величине значения было следующее.

Next branch

Будем называть «следующей веткой» ближайшую строку к текущей, для которой порядковый номер строки больше, а уровень тот же или меньше чем у текущей.

Цель – решить задачу без соединений и подзапросов.

С соединениями решение было бы тривиальным:

```
with t(id, parent_id, description, amount) as
(
  select 1 id, null, 'top', 10  from dual
  union all select 2, 1, 'top-one', 100 from dual
  union all select 3, 2, 'one-one', 2000 from dual
  union all select 4, 2, 'one-two', 3000 from dual
  union all select 5, 1, 'top-two', 1000 from dual
  union all select 6, 2, 'one-three', 300 from dual
  union all select 7, 6, 'three-one', 1 from dual
)
, h as
(
  select id, parent_id, description, amount, level l, rownum rn
    from t
   start with id = 1
 connect by parent_id = prior id
)
select h.*,
       (select min(rn)
          from h h0
         where h0.rn > h.rn
           and h0.l <= h.l) next_branch
  from h;
```

ID	PARENT_ID	DESCRIPTI	AMOUNT	L	RN	NEXT_BRANCH
1		top	10	1	1	
2	1	top-one	100	2	2	7
3	2	one-one	2000	3	3	4
4	2	one-two	3000	3	4	5
6	2	one-three	300	3	5	7
7	6	three-one	1	4	6	7
5	1	top-two	1000	2	7	

```
7 rows selected.
```

Для строк с ID in (3, 4) следующей веткой является следующая строка, т.к. у нее тот же уровень. Для строк с ID in (2, 6, 7) следующей веткой является строка с RN = 7, т.к. у нее меньший уровень.

Решение

Казалось бы, в решении достаточно просто использовать аналитический функции, но тут мы упираемся в одно из двух ограничений описанных ранее для них

1) При сортировке более чем по одному полю нельзя указывать rows/range.

   В данном случае нельзя указать что мы рассматриваем

```
rn range between 1 following and unbounded following
```
и в то же время
```
l range between unbounded preceding and current row
```

И представляется затруднительным объединить rn и l в одну величину.

2) Если попробовать ограничиться сортировкой только по одному из l или rn, то возникает необходимость отсечь строки по второму атрибуту в самой функции, но в ней нельзя обратиться к текущей строке.

В главе про model уже упоминалось, что первый случай достаточно тривиально реализуется через model clause, а второй случай требует итераций для присваивания значения некоторой меры текущей строки интересующему диапазону строк.

Конкретная реализация для текущей задачи может выглядеть так:

```
select *
from h
model
dimension by (l, rn)
measures (id, parent_id, rn xrn, 0 next_branch)
rules
(
  next_branch[any, any] order by rn, l =
    min(xrn)[l <= cv(l), rn > cv(rn)]
);
```

L	RN	ID	PARENT_ID	XRN	NEXT_BRANCH
1	1	1		1	
2	2	2	1	2	7
3	3	3	2	3	4
3	4	4	2	4	5
3	5	6	2	5	7
4	6	7	6	6	7
2	7	5	1	7	

```
7 rows selected.
```

```
select *
  from h
model
dimension by (rn)
measures (id, parent_id, l, 0 l_cur, rn xrn, 0 next_branch)
rules iterate (1e6) until l[iteration_number+2] is null
(
  l_cur[rn > iteration_number + 1] = l[iteration_number + 1],
  next_branch[iteration_number + 1] =
    min(case when l <= l_cur then xrn end)[rn > cv(rn)]
)
order by rn;
```

```
        RN         ID  PARENT_ID          L      L_CUR        XRN NEXT_BRANCH
---------- ---------- ---------- ---------- ---------- ---------- -----------
         1          1                     1          0          1
         2          2          1          2          1          2           7
         3          3          2          3          2          3           4
         4          4          2          3          3          4           5
         5          6          2          3          3          5           7
         6          7          6          4          3          6           7
         7          5          1          2          4          7

7 rows selected.
```

Однако задача достаточно изящно может быть решена и с использованием аналитических функций.

```
select h0.*,
       nullif(max(rn) over(order by s range between current
                   row and x - 1e-38 following),
              count(*) over()) + 1 next_branch
  from (select h.*,
               power(2 * 10, 1 - l) x,
               sum(power(2 * 10, 1 - l)) over(order by rn) s
          from h) h0;
```

```
        ID  PARENT_ID          L         RN          X          S NEXT_BRANCH
---------- ---------- ---------- ---------- ---------- ---------- -----------
         1                     1          1          1          1
         2          1          2          2        ,05       1,05           7
         3          2          3          3      ,0025     1,0525           4
         4          2          3          4      ,0025      1,055           5
         6          2          3          5      ,0025     1,0575           7
         7          6          4          6     ,000125   1,057625           7
         5          1          2          7        ,05   1,107625

7 rows selected.
```

Решение написано из предположения, что каждый узел имеет не более 10 прямых потомков. Если это условие выполняется, то предел суммы x для всех дочерних записей для конкретного узла уровня n будет равен

$$\sum_{i=1}^{\infty}\frac{10^i}{(10*2)^{i+n-1}}=\sum_{i=1}^{\infty}\frac{10^i}{20^{n-1}*10^i*2^i}=\frac{1}{20^{n-1}}\sum_{i=1}^{\infty}\frac{1}{2^i}=\frac{1}{20^{n-1}}$$

То есть предел равен значению x для самого узла, или более конкретно: для узла первого уровня предел суммы x дочерних записей будет стремится к 1, для узла второго уровня к 0.05 и так далее. На практике глубина ограничена точностью number, и предполагается, что разница между x и суммой x для всех дочерних записей будет превосходить 1e-38, отсюда определение окна «`range between current row and x - 1e-38 following`» - при такой конструкции в окно попадет текущий узел и все его дочерне записи. Если бы было указано «`range between 1e-38 following and x - 1e-38 following` », то в окно попадали бы только все дочерние записи.

Используя эту технику достаточно просто решать задачи, которые предполагают обработку всех дочерних узлом текущего узла и, возможно, сам узел.

Например, если требуется подсчитать число дочерних записей, или сумму всех дочерних записей и текущего узла, то это можно сделать следующим запросом.

```
select h0.*,
       count(*) over(order by s range between 1e-38 following and x - 1e-38 following) cnt_children,
       sum(amount) over(order by s range between current row and x - 1e-38 following) h_sum
  from (select h.*,
               power(2 * 10, 1 - l) x,
               sum(power(2 * 10, 1 - l)) over(order by rn) s
          from h) h0;
```

ID	PARENT_ID	DESCRIPTI	AMOUNT	L	RN	X	S	CNT_CHILDREN	H_SUM
1		top	10	1	1	1	1	6	6411
2	1	top-one	100	2	2	,05	1,05	4	5401
3	2	one-one	2000	3	3	,0025	1,0525	0	2000
4	2	one-two	3000	3	4	,0025	1,055	0	3000
6	2	one-three	300	3	5	,0025	1,0575	1	301
7	6	three-one	1	4	6	,000125	1,057625	0	1
5	1	top-two	1000	2	7	,05	1,107625	0	1000

```
7 rows selected.
```

Аналогичный подход можно реализовать и с помощью pattern matching

```
select *
  from (select h.*, power(2 * 10, 1 - l) x from h)
match_recognize
(
  order by rn
  measures
    first (id) as id,
    first (parent_id) as parent_id,
    first (l) as l,
    first (rn) as rn,
    final count(*)-1 cnt_children,
    final sum(amount) h_sum
  one row per match
  after match skip to next row
  pattern (y+)
  define
    y as sum(x) < 2 * first(x)
) mr;
```

ID	PARENT_ID	L	RN	CNT_CHILDREN	H_SUM
1		1	1	6	6411
2	1	2	2	4	5401
3	2	3	3	0	2000
4	2	3	4	0	3000
6	2	3	5	1	301
7	6	4	6	0	1
5	1	2	7	0	1000

```
7 rows selected.
```

В этом решении не была использована накопительная сумма по x, вместо этого было использовано условие «`sum(x) < 2 * first(x)`», поскольку сумма x для всех дочерних записей меньше x для родительской. Эквивалентным условием могло бы быть «`last(s) -`

`first(s) < first(x)`». Если же использовать «`max(s) - min(s) < first(x)`», то возникает ошибка ORA-03113. Использование last/first предпочтительнее потому что мы заем, что s монотонно возрастает с каждой строкой и нет необходимости использовать агрегатные функции. Однако, если бы цель была заставить работать пример с max/min, то можно было бы использовать all rows вместо one row с последующей фильтрацией – очевидно что добавление строк в набор данных с последующим отсечением вносит дополнительные затраты.

```
select *
  from (select h.*,
               power(2 * 10, 1 - l) x,
               sum(power(2 * 10, 1 - l)) over(order by rn) s
          from h) h0
match_recognize
(
  order by rn
  measures
    final count(*)-1 cnt_children,
    final sum(amount) h_sum,
    count(*) cnt
  all rows per match
  after match skip to next row
  pattern (y+)
  define
    y as max(s) - min(s) < first(x)
) mr
where cnt = 1;
```

Ну и главный момент в том, что при использовании pattern matching нет необходимости прибегать к трюку с суммой ряда, если требуется найти следующую ветку или применить некоторый агрегат ко всем дочерним записям. Достаточно просто составить шаблон под который попадают все дочерние записи для текущей и выполнять их поиск для каждой строки «`after match skip to next row`».

```
select *
  from h
match_recognize
(
  order by rn
  measures
    classifier() cls,
    first (id) as id,
    first (parent_id) as parent_id,
    first (l) as l,
    first (rn) as rn,
    first (amount) as amount,
    final count(child.*) cnt_children,
    final sum(amount) h_sum
  one row per match
  after match skip to next row
  pattern (strt child+|no_children)
  define
    child as child.l > strt.l
) mr;
```

```
CLS                    ID  PARENT_ID    L   RN     AMOUNT CNT_CHILDREN      H_SUM
--------------- ---------- ---------- --- --- ---------- ------------ ----------
CHILD                   1                1   1         10            6       6411
CHILD                   2          1     2   2        100            4       5401
NO_CHILDREN             3          2     3   3       2000            0       2000
NO_CHILDREN             4          2     3   4       3000            0       3000
CHILD                   6          2     3   5        300            1        301
NO_CHILDREN             7          6     4   6          1            0          1
NO_CHILDREN             5          1     2   7       1000            0       1000

7 rows selected.
```

Random subset

Предположим что есть таблица n строк с первичным ключом от 1 до n без пропусков.

```
create table t_id_value as
select rownum id, 'name' || rownum value from dual connect by rownum <= 2e6;
alter table t_id_value add constraint pk_t_id_value primary key (id);
```

Задача получить k уникальных случайных строк с равномерным распределением. Пусть для определенности k = 10.

Решение

Тривиальное решение может быть такое

```
select *
  from (select * from t_id_value order by dbms_random.value)
 where rownum <= 10;
```

В этом случае сначала для всех строк генерируется dbms_random.value, потом выполняется сортировка всех строк таблицы и на последнем этапе выбираются первые 10.

Если таблица «широкая», то сортировку несколько можно оптимизировать, выполняя ее только по первичному ключу и потом выбирая соответствующие строки по rowid.

```
select *
  from t_id_value
 where rowid in
       (select *
          from (select rowid from t_id_value order by dbms_random.value)
         where rownum <= 10);
```

Для имеющейся таблицы, однако, данный подход никаких особых преимуществ не даст.

Поскольку идентификаторы в таблице без пропусков и начинаются с единицы, то задачу можно решить значительно более эффективно следующим способом.

```
select *
  from t_id_value
 where id in (select trunc(dbms_random.value(1,
                     (select max(id) from t_id_value) + 1))
                from dual
              connect by level <= 10);
```

При таком подходе есть риск, что среди сгенерированных идентификаторов будут повторения. Например в запросе ниже получаем 9 строк вместо 10-ти:

```
exec dbms_random.seed(48673);

PL/SQL procedure successfully completed.

select *
  from t_id_value
 where id in (select trunc(dbms_random.value(1,
                     (select max(id) from t_id_value) + 1))
                from dual
              connect by level <= 10);

        ID VALUE
---------- --------------------------------------------
    564703 name564703
    917426 name917426
   1230672 name1230672
   1837951 name1837951
   1367140 name1367140
    248223 name248223
    873017 name873017
    581109 name581109
   1206874 name1206874

9 rows selected.
```

Этого можно избежать, если сгенерировать несколько «запасных строк» и потом выбрать m уникальных. Однако, чтоб гарантированно избежать повторений (даже теоретически) необходимо учитывать то, что уже сгенерировано (выполняя проверку с m-го шага). Подобные подходы с рассмотрением уже сгенерированных можно реализовать с использованием recursive subquery factoring или model.

```
select *
  from t_id_value
 where id in
 (
  select distinct x
    from dual
  model return updated rows
  dimension by (0 id)
  measures(0 i, 0 x, (select max(id) from t_id_value) max_id)
  rules
  iterate (1e9) until i[0] = 10
  (
   x[iteration_number] = trunc(dbms_random.value(1, max_id[0] + 1)),
   i[0] = case when iteration_number < 10 - 1
               then 0 else count(distinct x)[any] end
  )
 );
```

Если использовать «`exec dbms_random.seed(48673)`» то проверка на число уникальных элементов будет выполнена дважды: после десятого сгенерированного элемента и после одиннадцатого, так как после десятого будет всего девять уникальных. В подавляющем большинстве случаев однако потребуется только однократная проверка.

Решение с использованием recursive subquery factoring может выглядеть так

```
with rec(lvl, batch)
     as (select 1,
                numbers(trunc(dbms_random.value(1, 2e6 + 1)))
           from dual
         union all
         select lvl + 1,
                batch multiset union all
                numbers(trunc(dbms_random.value(1, 2e6 + 1)))
           from rec
          where case when lvl < 10 then 0
                -- cardinality(set()) не работает в рекурсивном члене
                     else (select count(*) from table(set(rec.batch)))
                end < 10)
select *
  from t_id_value
 where id in (select column_value
                from (select *
                        from (select * from rec t order by lvl desc)
                       where rownum = 1),
                     table(batch));
```

Здесь мы накапливаем сгенерированные элементы в столбе batch, который имеет тип коллекции. Проверку выполняем по аналогии как было в примере с model. Для простоты, максимальное значение указано константой вместо скалярного подзапроса, иначе для того, чтоб гарантировать, что скалярный подзапрос выполнится однократно пришлось бы реализовывать дополнительную логику.

Недостаток обоих SQL подходов в том, что для проверки уникальности надо выполнять проход по всем сгенерированным числам, что может быть затратно для большого числа k. PL/SQL позволяет этого избежать, если использовать ассоциативный массив. Пример реализации ниже (думаю очевидно, что эту логику можно реализовать в pipelined функции и использовать сгенерированные числа в запросе).

```
declare
  type tp_arr is table of binary_integer index by binary_integer;
  arr tp_arr;
  i   int := 0;
begin
  while true loop
    arr(trunc(dbms_random.value(1, 2e6 + 1))) := null;
    i := i + 1;
    if i >= 10 and arr.count = 10 then
      exit;
    end if;
  end loop;

  i := arr.first;
  while (i is not null) loop
    dbms_output.put_line(i);
    i := arr.next(i);
  end loop;
end;
/
```

С другой стороны, чем больше чисел надо сгенерировать тем разумнее будет самый первый подход с сортировкой всего множества. Также стоит еще раз подчеркнуть все подходы с генерацией работают только если известны верхняя и нижняя границы и нет пропусков. Если же, например, первичный ключ - строка, то надо, как минимум, сначала вычитать все данные, чтоб узнать число элементов.

Covering ranges

Предположим есть таблица следующего вида

```
create table t_range(a, b) as
(select 1, 15 from dual
union all select 3, 17 from dual
union all select 6, 19 from dual
union all select 10, 21 from dual
union all select 17, 26 from dual
union all select 18, 29 from dual
union all select 20, 32 from dual
union all select 24, 35 from dual
union all select 28, 45 from dual
union all select 30, 49 from dual);
```

b > a для каждой пары a, b; a уникально

Необходимо получить отрезки (1:15), (17:26), (28:45), то есть начинаем со строки с минимальным a и следующую строку берем такую что для нее a больше b из текущей строки и так далее.

Решение

Достаточно просто (хотя и неэффективно в плане производительности) решается с помощью connect by.

```
select a, b
  from (select a,
               b,
               min(a) over(order by a range between b - a following
                           and unbounded following) as next_a,
               min(a) over() start_a
          from test)
 start with a = start_a
connect by prior next_a = a;

         A          B
---------- ----------
         1         15
        17         26
        28         45

select a, b
  from (select a, b, lag(a) over(order by a) as lag_a from test)
 start with lag_a is null
connect by a >= prior b and lag_a < prior b;
```

```
         A          B
---------- ----------
         1         15
        17         26
        28         45
```

Однако очень элегантно и эффективно задачу можно решить с помощью pattern matching.

```
select *
 from t_range
match_recognize
(
  order by a
  all rows per match
  pattern((x|{-dummy-})+)
  define
    x as nvl(last(x.b, 1), 0) <= x.a
) mr;
```

```
         A          B
---------- ----------
         1         15
        17         26
        28         45
```

Можно привести еще несколько альтернативных решений с помощью model, но их производительность будет уступать pattern matching.

Zeckendorf representation

Теорема Цекендорфа гласит, что всякое натуральное число можно единственным образом представить в виде суммы одного или нескольких различных чисел Фибоначчи так, чтобы в этом представлении не оказалось двух соседних чисел из последовательности Фибоначчи.

Разложение выполняется с помощью жадного алгоритма, когда на каждом этапе выбирается наибольшее возможное число Фибоначчи.

Итак, путь есть таблица fib(lvl, value) содержащая числа Фибоначчи и их порядок – данные могут быть сгенерированы одним из описанных в книге способов (ограничимся первыми двадцатью). Задача получить для таблицы, содержащей произвольные натуральные числа их представление Цекендорфа.

Для таблицы

```
create table n(num) as select 222 from dual union all select 3690 from dual;
```

Ожидается результат

```
       NUM PATH
---------- ------------------------------
       222 144+55+21+2
      3690 2584+987+89+21+8+1
```

Решение

Вариант решения в лоб – через connect by.

- перебираем все возможные комбинации чисел Фибоначчи, которые меньше данного числа
- отсекаем те, что в сумме дают нужное значение
- из отфильтрованных комбинаций выбираем ту, что с минимальным числом элементов

```
with n_fib as
 (select num, value, lvl, max(lvl) over(partition by num) max_lvl
    from n
    join fib
      on fib.value <= n.num),
permutation as
 (select num, sys_connect_by_path(value, '+') path, level p_lvl
    from n_fib
   start with lvl = max_lvl
  connect by prior num = num
         and prior value > value
         and sys_guid() is not null)
select num,
       max(substr(path, 2)) keep(dense_rank first order by p_lvl) path
  from (select num, path, p_lvl
          from permutation p
          join fib
            on instr(p.path || '+', '+' || fib.value || '+') > 0
         group by num, path, p_lvl
        having sum(value) = num)
 group by num
 order by num;
```

Очевидно что такой подход абсолютно неэффективен и можно просто для каждого числа выполнять проход по числам Фибоначчи, которые меньше него, в сторону убывания. На каждом шаге отмечать число, если его сумма с уже отмеченными числами не превышает заданного. Такие пошаговые вычисления могут быть выполнены с помощью recursive subquery factoring.

```
with n_fib as
 (select num, value, lvl, max(lvl) over(partition by num) max_lvl
    from n
    join fib
      on fib.value <= n.num),
rec(lvl, num, f, s) as
 (select 1, n_fib.num, n_fib.value, 0
    from n_fib
   where n_fib.lvl = n_fib.max_lvl
  union all
  select rec.lvl + 1, l.num, l.value, rec.f + rec.s
    from rec
   cross apply (select *
                  from (select *
                          from n_fib
                         where n_fib.num = rec.num
                           and n_fib.value + rec.s + rec.f <= rec.num
                         order by lvl desc)
```

```
                 where rownum = 1) l)
cycle lvl set c to 1 default 0
select num, listagg(f, '+') within group(order by f desc) path
  from rec
 group by num
 order by num;
```

Здесь lateral view была использована для того, чтобы получить максимальное число Фибоначчи на каждом шаге. Как уже было замечено в предыдущей главе, при использовании нескольких соединений или lateral views в рекурсивном члене может быть ложное срабатывание зацикленности, поэтому была использована конструкция «cycle».

Вместо lateral views логика для получения максимального числа Фибоначчи может быть реализована в виде скалярного подзапроса – в таком случае запрос может быть выполнен для версии 11g.

```
with rec(lvl, num, f, s) as
 (select 1,
         n.num,
         (select max(fib.value) from fib where fib.value <= n.num),
         0
    from n
  union all
  select lvl + 1,
         d.num,
         (select max(fib.value)
            from fib
           where fib.value <= d.num - (d.f + d.s)),
         d.f + d.s
    from rec d
   where d.s + d.f < d.num)
select num, listagg(f, '+') within group(order by f desc) path
  from rec
 group by num
 order by num;
```

Еще более изящно и эффективно можно применить подход с pattern matching.

```
select num,
       (select listagg(value, '+')
               within group(order by value desc) path
          from (select n.num, fib.value from fib) y
         match_recognize
         (
           order by value desc
           all rows per match
           pattern((x|{-dummy-})+)
           define
             x as sum(x.value) <= num
         ) mr
       ) path
  from n;
```

Вместо скалярного запроса логику можно было бы применить после соединения чисел для представления и чисел Фибоначчи и воспользоваться «partition by num order by value desc» в конструкции match_recognize. Логику с recursive subquery factoring невозможно

реализовать с помощью скалярного подзапроса поскольку для этой конструкции будут не видны поля основной таблицы. Продемонстрируем это на простом примере

```
select t.*,
       (with rec(lvl) as (select /*t.id*/ 5 lvl from dual
                          union all
                          select rec.lvl + 1 from rec where lvl < 10)
          select listagg(lvl, ', ') within group(order by lvl)
            from rec) str
  from (select 5 id from dual) t;
```

Если раскомментировать t.id, то будет ошибка «ORA-00904: "T"."ID": invalid identifier».

Последнее что стоит заметить – аналогичные вычисление можно реализовать с помощью model, но производительность будет заметно уступать pattern matching, но опережать recursive subquery factoring, поскольку как и в случае pattern matching будет всего один проход.

Top paths

Для таблицы со списком директорий в файловой системы вывести только те, что не имеют поддиректорий.

```
create table t_path(path) as
select '/tmp/cat/' from dual
union all select '/tmp/cata/' from dual
union all select '/tmp/catb/' from dual
union all select '/tmp/catb/catx/' from dual
union all select '/usr/local/' from dual
union all select '/usr/local/lib/liba/' from dual
union all select '/usr/local/lib/libx/' from dual
union all select '/var/cache/' from dual
union all select '/var/cache/'||'xyz'||rownum||'/' from dual
connect by level <= 1e6;
```

Для указанных данных результатом будет

```
PATH
------------------------------------------------------
/tmp/cat/
/tmp/cata/
/tmp/catb/
/usr/local/
/var/cache/
```

Решение

Решение в лоб – это self join по like с последующей фильтрацией. Главный недостаток, что метод соединения может быть только nested loops, учитывая предикат соединения. То есть для каждой строк будет выполняться проход по всем строкам в соединяемом наборе. Это решение

может быть улучшено, если использовать not exists – в таком случае соединение будет выполняться до первого найденного совпадения.

```
select t_path.path
  from t_path
  left join t_path t_top
    on t_path.path like t_top.path || '%_/'
 where t_top.path is null;

PATH
------------------------------------------------------
/tmp/cat/
/tmp/cata/
/tmp/catb/
/usr/local/
/var/cache/

Elapsed: 00:00:34.54
select path
  from t_path
 where not exists (select null
          from t_path t_top
         where t_path.path like t_top.path || '%_/');

PATH
------------------------------------------------------
/tmp/cat/
/tmp/cata/
/tmp/catb/
/usr/local/
/var/cache/

Elapsed: 00:00:09.63
```

При использовании подзапроса время уменьшилось более чем в три раза. Если в подзапросе указать дополнительный фильтр «`and where rownum = 1`», то время его выполнения не изменится учитывая механику работы фильтра.

Очевидно, основные затраты уходят на соединение и лучше бы от него избавиться. В решении ниже отсутствует self join, но используется lateral view для получения подпутей для каждого пути.

```
with t0 as
 (select path, substr(path, 1, instr(path, '/', 1, l.id + 1)) token
    from t_path,
         lateral (select rownum id
                    from dual
                 connect by level < length(path) - length(replace(path,
'/'))) l),
t1 as (select t0.*, count(*) over(partition by path) cnt from t0),
t2 as (select t1.*, min(cnt) over(partition by token) m from t1)
select path from t2 group by path having min(cnt) = min(m) order by path;
```

```
PATH
----------------------------------------------------
/tmp/cat/
/tmp/cata/
/tmp/catb/
/usr/local/
/var/cache/

      Elapsed: 00:00:22.78
```

Время выполнения между первым и вторым вариантом.

Если знать максимальную глубину вложенности, то можно реализовать следующий подход: для каждого подпути смотрим есть ли строки для которых это конечный путь. Если такие строки есть, а в текущей строке есть еще вложенность, то текущая строка в результат не попадает.

```
select path
  from (select t1.*,
               min(nvl2(p2, 1, 0)) over(partition by p1) m2,
               min(nvl2(p3, 1, 0)) over(partition by p1, p2) m3,
               min(nvl2(p4, 1, 0)) over(partition by p1, p2, p3) m4,
               min(nvl2(p5, 1, 0)) over(partition by p1, p2, p3, p4) m5
          from (select path,
                       substr(path, i1, i2 - i1) p1,
                       substr(path, i2, i3 - i2) p2,
                       substr(path, i3, i4 - i3) p3,
                       substr(path, i4, i5 - i4) p4,
                       substr(path, i5, i6 - i5) p5
                  from (select path,
                               instr(path, '/', 1, 1) i1,
                               instr(path, '/', 1, 2) i2,
                               instr(path, '/', 1, 3) i3,
                               instr(path, '/', 1, 4) i4,
                               instr(path, '/', 1, 5) i5,
                               instr(path, '/', 1, 6) i6
                          from t_path) t0) t1)
 where not (m2 = 0 and p2 is not null or m3 = 0 and p3 is not null or
            m4 = 0 and p4 is not null or m5 = 0 and p5 is not null);

PATH
----------------------------------------------------
/tmp/cat/
/tmp/cata/
/tmp/catb/
/usr/local/
/var/cache/

Elapsed: 00:00:13.68
```

Несмотря на сделанные допущения, данный вариант проигрывает запросу с not exists и основные ресурсы уходят на аналитическую сортировку. Важно отметить, что для начала всех тестов было выставлено максимальное значение «sort_area_size» и все запросы были полностью выполнены в памяти.

Последний вариант решения будет с использованием pattern matching.

```
select *
from t_path
match_recognize
(
  order by path
  measures
    first(path) path
  one row per match
  pattern(x+)
  define
    x as path like first(path) || '%'
) mr;

PATH
------------------------------------------------------
/tmp/cat/
/tmp/cata/
/tmp/catb/
/usr/local/
/var/cache/

Elapsed: 00:00:00.89
```

Время выполнения меньше секунды! Алгоритм работы предельно прост: для каждого совпадения мы проверяем является ли первый путь частью текущего пути. Если условие не выполняется, то начинается «новая группа».

Поскольку pattern matching не доступно до 12c, то для этих версий оптимальным решением было бы применение PL/SQL. Иными словами – эта задача еще один отличный пример для предыдущей главы, когда целесообразно реализовать проход по курсору с построчной обработкой вместо PL/SQL подхода.

Resemblance group

Объединить значения в группу схожести (промаркировать) по следующему алгоритму: начинаем со строки с минимальным id и двигаясь по порядку в сторону его увеличения добавляем каждую новую строку в группу, если в группе есть элемент, значение которого отличается от текущего не более чем на единицу.

Для следующих данных

```
create table t_resemblance(id, value) as
(select 1, 1 from dual
union all select 2, 2 from dual
union all select 3, 2.5 from dual
union all select 4, 3.4 from dual
union all select 5, 0.4 from dual
union all select 6, 5 from dual
union all select 7, -0.5 from dual
union all select 8, -2 from dual
union all select 9, -1 from dual
union all select 10, 3 from dual
union all select 11, 4 from dual
union all select 12, 5 from dual);
```

В результате ожидаются все строки кроме шестой и восьмой. Двенадцатая строка тоже имеет значение 5, как и шестая, но она попала в результат, так как на момент ее обхода в группе уже была строка, которая отличалась от нее не более чем на единицу.

Решение

По закону транзитивности, для каждого элемента в группе найдется тот, который отличается от него не более чем на единицу, поэтому при добавлении нового элемента достаточно его сравнивать с минимальным и максимальным.

На языке SQL задача решается с помощью итеративной модели.

```
select *
from t_resemblance
model
dimension by (id)
measures(value, 0 mi, 0 ma, 0 flag)
rules iterate (1e9) until value[iteration_number + 2] is null
(
  mi[iteration_number + 1] = case when nvl(abs(mi[iteration_number] - value[iteration_number + 1]), 0) <=1
                                  then least(nvl(mi[iteration_number], 1e10), value[iteration_number + 1])
                                  else mi[iteration_number]
                             end,
  ma[iteration_number + 1] = case when nvl(abs(ma[iteration_number] - value[iteration_number + 1]), 0) <=1
                                  then greatest(nvl(ma[iteration_number], -1e10), value[iteration_number + 1])
                                  else ma[iteration_number]
                             end,
  flag[iteration number + 1] = case when value[iteration number + 1]
                                         between mi[iteration_number + 1] and ma[iteration_number + 1]
                                    then 1
                               end
);
```

ID	VALUE	MI	MA	FLAG
1	1	1	1	1
2	2	1	2	1
3	2.5	1	2.5	1
4	3.4	1	3.4	1
5	.4	.4	3.4	1
6	5	.4	3.4	
7	-.5	-.5	3.4	1
8	-2	-.5	3.4	
9	-1	-1	3.4	1
10	3	-1	3.4	1
11	4	-1	4	1
12	5	-1	5	1

12 rows selected.

Алгоритм выполняет итеративный проход по набору данных на каждой итерации меняя значения мер только текущей строки. Выполняется сравнение текущего элемента с предыдущим минимум и максимумом и если разница не более единицу, то соответствующая граница обновляется. После обновления границ проставляется значение флажка.

Итерации здесь необходимы поскольку необходимо менять несколько мер для каждой строки. Как было описано в главе про model, без итераций выполняется расчет мер по столбцам. То есть сначала вычисляются все значения меры для первого правила, потом для второго и так далее.

Учитывая специфику алгоритма можно попытаться решить задачу с помощью pattern matching, но при таком подходе мы сталкиваемся с некоторыми ограничениями, которые делают невозможным реализацию требуемого алгоритма

1) При использовании агрегатной функций, она применяется ко всем элементам группы, включая текущий. То есть нельзя сравнить текущий элемент с минимальным среди уже совпавших.
2) В правилах нельзя использовать меры и тем более значения мер для предыдущих строк. Это объясняется тем, что меры вычисляются после того как совпадение найдено.

Можно реализовать алгоритм с просмотром фиксированного числа предыдущих элементов в текущей группе, но это нельзя рассматривать как полноценное решение.

```
select *
from t_resemblance
match_recognize
(
  order by id
  measures
    match_number() match,
    classifier() cls,
    min(value) mi,
    max(value) ma,
    count(*) cnt
  all rows per match
  pattern((x|dummy)+)
  define
    x as x.value between least(decode(count(*), 1, x.value, prev(x.value,1)),
                               nvl(prev(x.value,1),1e10),nvl(prev(x.value,2),1e10),
                               nvl(prev(x.value,3),1e10),nvl(prev(x.value,4),1e10),
                               nvl(prev(x.value,5),1e10)) - 1
                 and greatest(decode(count(*), 1, x.value, prev(x.value,1)),
                               nvl(prev(x.value,1),1e10),nvl(prev(x.value,2),1e10),
                               nvl(prev(x.value,3),1e10),nvl(prev(x.value,4),1e10),
                               nvl(prev(x.value,5),1e10)) + 1
                  or count(*) = 1
) mr;
```

ID	MATCH	CLS	MI	MA	CNT	VALUE
1	1	X	1	1	1	1
2	1	X	1	2	2	2
3	1	X	1	2.5	3	2.5
4	1	X	1	3.4	4	3.4
5	1	X	.4	3.4	5	.4
6	1	DUMMY	.4	5	6	5
7	1	X	-.5	5	7	-.5
8	1	DUMMY	-2	5	8	-2
9	1	X	-2	5	9	-1
10	1	X	-2	5	10	3
11	1	X	-2	5	11	4
12	1	X	-2	5	12	5

```
12 rows selected.
```

Для обозначения текущих границ для каждой строки были использованы просто «`min(value)/max(value)`», но аналогичные выражения не могут быть использованы в секции определений.

Стоит обратить внимание, что в секции определений можно использовать подзапросы, но они не могут быть коррелированными («`ORA-62510: Correlated subqueries are not allowed in MATCH_RECOGNIZE clause`»), то есть эта возможность тоже не помогает решить данную задачу.

Еще одним SQL решением может быть подход с использование recursive subquery factoring.

Baskets

Имеются корзины с определенной вместимостью, их требуется наполнить элементами, так чтоб их сумма не превышала вместимость корзины. Корзины обходятся в порядке увеличения идентификатора, а элементы выбираются в порядке возрастания приоритета, при этом если элемент был уже выбран для другой корзины, то он не рассматривается.

Для исходных данных ниже

```
with
  baskets as (
    select 100 as basket_id, 500000 as basket_amount from dual union all
    select 200 as basket_id, 400000 as basket_amount from dual union all
    select 300 as basket_id, 1000000 as basket_amount from dual
  ),
  inventory as (
    select 1000001 as item_id, 50000 as item_amount from dual union all
    select 1000002 as item_id, 15000 as item_amount from dual union all
    select 1000003 as item_id, 250000 as item_amount from dual union all
    select 1000004 as item_id, 350000 as item_amount from dual union all
    select 1000005 as item_id, 45000 as item_amount from dual union all
    select 1000006 as item_id, 100500 as item_amount from dual union all
    select 1000007 as item_id, 200500 as item_amount from dual union all
    select 1000008 as item_id, 30050 as item_amount from dual union all
    select 1000009 as item_id, 400500 as item_amount from dual union all
    select 1000010 as item_id, 750000 as item_amount from dual
  ),
  eligibility as (
    select 100 as basket_id, 1000003 as item_id, 1 as priority_level from dual union all
    select 100 as basket_id, 1000004 as item_id, 2 as priority_level from dual union all
    select 100 as basket_id, 1000002 as item_id, 3 as priority_level from dual union all
    select 100 as basket_id, 1000005 as item_id, 4 as priority_level from dual union all
    select 200 as basket_id, 1000004 as item_id, 1 as priority_level from dual union all
    select 200 as basket_id, 1000003 as item_id, 2 as priority_level from dual union all
    select 200 as basket_id, 1000001 as item_id, 3 as priority_level from dual union all
    select 200 as basket_id, 1000005 as item_id, 4 as priority_level from dual union all
    select 200 as basket_id, 1000007 as item_id, 5 as priority_level from dual union all
    select 200 as basket_id, 1000006 as item_id, 6 as priority_level from dual union all
    select 300 as basket_id, 1000002 as item_id, 1 as priority_level from dual union all
    select 300 as basket_id, 1000009 as item_id, 2 as priority_level from dual union all
    select 300 as basket_id, 1000010 as item_id, 3 as priority_level from dual union all
    select 300 as basket_id, 1000006 as item_id, 4 as priority_level from dual union all
    select 300 as basket_id, 1000008 as item_id, 5 as priority_level from dual
  )
```

Ожидается следующий результат

BASKET_ID	PRIORITY_LEVEL	ITEM_ID	BASKET_AMOUNT	ITEM_AMOUNT	RESULT
100	1	1000003	500000	250000	250000
100	2	1000004	500000	350000	0
100	3	1000002	500000	15000	265000
100	4	1000005	500000	45000	310000
200	1	1000004	400000	350000	350000
200	2	1000003	400000	250000	0
200	3	1000001	400000	50000	400000
200	4	1000005	400000	45000	0
200	5	1000007	400000	200500	0
200	6	1000006	400000	100500	0
300	1	1000002	1000000	15000	0
300	2	1000009	1000000	400500	400500
300	3	1000010	1000000	750000	0
300	4	1000006	1000000	100500	501000
300	5	1000008	1000000	30050	531050

15 rows selected.

Решение

Основная сложность в задаче то, что надо учитывать уже выбранные элементы. Иначе задача решалась бы элементарно по аналогии с подходом для «Zeckendorf representation». С другой стороны, именно эта деталь делает невозможным решение с помощью pattern matching.

Задача может быть решена с помощью model следующим образом:

```
select *
from t
model
dimension by (basket_id, priority_level, item_id)
measures (basket_amount, item_amount, 0 result)
rules
(
  result[any, any, any] order by basket_id, priority_level, item_id =
  case when max(result)[any, any, cv(item_id)] = 0 and
            nvl(max(result)[cv(basket_id),priority_level < cv(priority_level),any],0) +
            item_amount[cv(basket_id),cv(priority_level),cv(item_id)]
            <= max(basket_amount)[cv(basket_id),cv(priority_level),any]
       then nvl(max(result)[cv(basket_id),priority_level < cv(priority_level),any],0) +
            item_amount[cv(basket_id),cv(priority_level),cv(item_id)]
       else 0
  end
)
order by 1, 2;
```

Для уникальной адресации достаточно комбинации (basket_id, priority_level), но item_id было добавлено в измерения для возможности определить был ли использован текущий элемент. Вся логика реализована в одном компактном правиле, но в нем используется несколько разных агрегатов с разной адресацией.

Можно обратить внимание что для решения может быть применен тот же подход, что и для предыдущей задачи: проходим весь набор данных в определенном порядке и вычисляем для текущей строки значение нескольких мер. Это может быть реализовано либо с помощью итеративной модели либо с помощью recursive subquery factoring. Последнее может выглядеть так:

```
with
t0 as (select t.*, row_number() over (order by basket_id, priority_level) rn from t),
rec(basket_id, item_id, basket_amount, item_amount, priority_level, rn, total, is_used, str) as
(select t.basket_id, t.item_id, t.basket_amount, t.item_amount, t.priority_level, t.rn,
        case when t.item_amount <= t.basket_amount then t.item_amount else 0 end,
        case when t.item_amount <= t.basket_amount then 1 end,
        cast(case when t.item_amount <= t.basket_amount then ',' || t.item_id end as
varchar2(4000))
 from t0 t where rn = 1
 union all
 select t.basket_id, t.item_id, t.basket_amount, t.item_amount, t.priority_level, t.rn,
        case when decode(t.basket_id, r.basket_id, r.total, 0) + t.item_amount <= t.basket_amount
              and instr(r.str, t.item_id) = 0
             then decode(t.basket_id, r.basket_id, r.total, 0) + t.item_amount
             else decode(t.basket_id, r.basket_id, r.total, 0) end,
        case when decode(t.basket_id, r.basket_id, r.total, 0) + t.item_amount <= t.basket_amount
              and instr(r.str, t.item_id) = 0
             then 1 end,
        case when decode(t.basket_id, r.basket_id, r.total, 0) + t.item_amount <= t.basket_amount
              and instr(r.str, t.item_id) = 0
             then r.str || ',' || t.item_id
             else r.str end
 from t0 t
 join rec r
 on t.rn = r.rn + 1)
select * from rec;
```

Для определения, был ли уже использован элемент, использована строка с конкатенацией всех использованных элементов. Для того, чтоб обойти ограничение на длину строки, эта же логика может быть реализована с помощью коллекций.

Не смотря на лаконичность решения с model, подобный подход жизнеспособен на достаточно малых объемах данных – порядка тысяч строк. В противном случае лучший вариант – это PL/SQL решение с применением ассоциативного массива элементов, указывающего был ли использован каждый конкретный элемент.

Longest common subsequence

Классическая задача поиска наибольшей увеличивающейся подпоследовательности состоит в нахождении наиболее длинной возрастающей подпоследовательности в данной последовательности элементов. Стоит отметить что элементы подпоследовательности не обязательно должны идти подряд в исходной последовательности.

Например, для последовательности

14, 15, 9, 11, 16, 12, 13

Ожидается результат

9, 11, 12, 13

Для простоты будем искать просто длину наибольшей увеличивающейся подпоследовательности, без вывода самой последовательности. Для данных выше ответ – 4.

Решение

При решении этой задачи начнем с рассмотрения PL/SQL подходов в отличие от всех предыдущих задач этой главы. Решение может быть реализовано достаточно эффективно методами динамического программирования.

Итак представим, что необходимо вычислить длину последовательности для текущего элемента, при этом задача решена для всех предыдущих элементов. Тогда ответом для текущего элемента будет максимум из длин последовательностей для предыдущих элементов, которые меньше него. Возможно, звучит довольно запутанно, но алгоритм выглядит достаточно тривиально.

Воспользуемся временной таблицей tmp из раздела «Задачи комбинаторного характера» и типом numbers из раздела «Unnesting collections».

```
declare
  t numbers := numbers(14, 15, 9, 11, 16, 12, 13);
begin
  delete from tmp;

  for i in 1 .. t.count loop

    insert into tmp
      (lvl, x, num)
    values
      ((select nvl(max(lvl), 0) + 1 from tmp where num < t(i)), i, t(i));

  end loop;

end;
/

PL/SQL procedure successfully completed.

select * from tmp;

       LVL          X        NUM
---------- ---------- ----------
         1          1         14
         2          2         15
         1          3          9
         2          4         11
         3          5         16
         3          6         12
         4          7         13

7 rows selected.
```

Чтоб получить итоговый максимум надо либо выполнить проход по tmp либо воспользоваться вспомогательной переменной в PL/SQL. Поскольку для каждого элемента выполняется полный проход по всем предыдущим, то сложность алгоритма $O(n^2)$. Как раз эта необходимость прохода по всем обработанным элементам делает нецелесообразным реализацию с помощью recursive subquery factoring – детали, опять же, в разделе про «Задачи комбинаторного характера».

Возвращаясь к PL/SQL алгоритм может быть улучшен, если воспользоваться вспомогательным массивом, который будет содержать в котором будет храниться наибольшую возрастающую подпоследовательность и обновляться на каждом шаге. Обновление реализовано по следующему принципу: выполняется двоичный поиск наибольшего элемента меньше текущего и следующему за ним присваивается значение текущего. Поскольку сложность двоичного поиска $O(\log_2 n)$, то общая сложность алгоритма $O(n * \log_2 n)$.

```
declare

  x    numbers := numbers(14, 15, 9, 11, 16, 12, 13);
  m    numbers := numbers();
  l    int;
  newl int;
  v    varchar2(4000);
  -- index of the greatest element lower than p in array M
  function f(p in number) return int as
    lo  int;
    hi  int;
    mid int;
  begin
    lo := 1;
    hi := l;
    while lo <= hi loop
      mid := ceil((lo + hi) / 2);
      if x(m(mid)) < p then lo := mid + 1; else hi := mid - 1; end if;
    end loop;
    return lo;
  end;
begin
  m.extend(x.count);

  l := 0;
  for i in 1 .. x.count loop
    newl := f(x(i));
    m(newl) := i;

    if newl > l then l := newl; end if;

    v := '';
    for j in 1 .. l loop
      v := v || ' ' || x(m(j));
    end loop;
    dbms_output.put_line(i || ' ' || v);
  end loop;
end;
/
1  14
2  14 15
3  9 15
4  9 11
5  9 11 16
6  9 11 12
7  9 11 12 13

PL/SQL procedure successfully completed.
```

На SQL алгоритм может быть реализован следующим образом:

```
with t(id, value) as
(select rownum, column_value from table(numbers(14, 15, 9, 11, 16, 12, 13)))
select *
  from t
model
dimension by (id, value)
measures(0 l)
(l[any, any] order by id = nvl(max(l)[id < cv(id),value < cv(value)],0) + 1)
order by 1;
```

Подход аналогичен первому алгоритму на PL/SQL, а вот аналогию второму алгоритму реализовать средствами SQL не представляется возможным. Применяя model, мы работаем с набором данных, состоящим из строк и столбцов (пусть и рассматриваем его как многомерный массив) и использование каких-либо дополнительных структур данных, для оптимизации работы с основным набором, невозможно.

Quine

Последняя задача более для развлекательных целей чем для демонстрации возможностей Oracle SQL. Quine – это программа, которая выдает точную копию своего текст. В интернете можно найти огромное число решений для различных языков программирования и я не буду останавливаться на решении для PL/SQL – подход во многом совпадает с решением для языка Pascal, например. Несколько интересней рассмотреть SQL подходы.

Решение

Одним из требований является, что программы (в нашем случае запрос) не должны обращаться к внешним источникам для чтения своего текста. То есть следующий запрос нельзя рассматривать как полноценное решение.

```
set pagesize 0 linesize 90

select sql_text||';'from v$sqlarea join v$session using(sql_id)where sid=userenv('sid');
select sql_text||';'from v$sqlarea join v$session using(sql_id)where sid=userenv('sid');
```

А вот следующие два решения удовлетворяют всем требованиям.

```
select substr(rpad(1,125,'||chr(39)),26)from
dual;select substr(rpad(1,125,'||chr(39)),26)from
dual;
select substr(rpad(1,125,'||chr(39)),26)from
dual;select substr(rpad(1,125,'||chr(39)),26)from
dual;
```

```
select
replace('@''[@'||chr(93)||''')from dual;','@',q'[select
replace('@''[@'||chr(93)||''')from dual;','@',q]')from dual;
select
replace('@''[@'||chr(93)||''')from dual;','@',q'[select
replace('@''[@'||chr(93)||''')from dual;','@',q]')from dual;
```

Возможно у Вас будут идеи для решения покороче?

Резюме

На выбранных задачах было продемонстрировано применение специфических конструкций языка Oracle SQL, которые значительно расширяют возможности SQL описанные в стандарте. Особенно стоит выделить pattern matching, который позволяет очень эффективно решать задачи, которые ранее потребовали процедурного подхода. Некоторые специфические задачи могут быть решены с помощью recursive subquery factoring или model сопоставимо по производительности с PL/SQL. Оба подхода не требуют создания коллекций объектного типа для результата в отличие от процедурного подхода, кроме того в плюсы model можно отнести лаконичность записи и хорошие возможности по распараллеливанию при указании partitioning, а к плюсам recursive subquery factoring – работу в рамках SQL движка и отсутствие необходимости переключения контекста при вызове рекурсивного члена. Если же требуется выполнить достаточно сложный алгоритм, требующий вспомогательных структур данных, то в таком случае наиболее подходящий инструмент – PL/SQL, но все же, прибегая к процедурному расширению языка для работы с данными важно понимать причины в каждом конкретном случае. Последнее что стоит отметить, алгоритмы, требующие сложных вычислений, вероятно имеет смысл реализовывать с помощью внешних библиотек на C. Часто многое решают детали, поэтому имеет смысл сравнивать разные подходы учитывая конкретную специфику.

Summary

Ниже перечислим задачи, решенные во второй части книги и применяемые механизмы Oracle SQL.

#	Quiz	CB	AF	RW	M	PM	PL
1	Converting into decimal	+					+
2	Connected components	+					+
3	Ordering dependencies	+		+			
4	Percentile with shift		+				
5	N consequent 1s		+		+	+	
6	Next value		+		+	+	
7	Next branch		+		+	+	
8	Random subset			+	+		+
9	Covering ranges	+	+			+	
10	Zeckendorf representation	+		+		+	
11	Top paths		+			+	
12	Resemblance group				+	+	
13	Baskets			+	+		
14	Longest common subsequence				+		+
15	Quine						

Использованные сокращения
AF: analytical functions
CB: connect by
RW: recursive with
M: model
PM: pattern matching
PL: PL/SQL

Links

1. A LOOK UNDER THE HOOD OF CBO: THE 10053 EVENT
   http://www.centrexcc.com/A%20Look%20under%20the%20Hood%20of%20CBO%20-%20the%2010053%20Event.pdf
2. Closing The Query Processing Loop in Oracle 11g
   http://www.vldb.org/pvldb/1/1454178.pdf
3. The Oracle Optimizer Explain the Explain Plan
   http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-explain-the-explain-plan-052011-393674.pdf
4. Query Optimization in Oracle Database10g Release 2
   http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-general-query-optimization-10gr-130948.pdf
5. SQL Sucks
   http://www.nocoug.org/download/2006-08/SQL_Sucks_NoCOUG_Journal_Article_Part_2.pdf
6. Explaining the EXPLAIN PLAN
   https://nocoug.files.wordpress.com/2014/08/nocoug_journal_201408.pdf
7. Universality in Elementary Cellular Automata
   http://www.complex-systems.com/pdf/15-1-1.pdf
8. Absolutely Typical - The whole story on Types and how they power PL/SQL Interoperability
   https://technology.amis.nl/wp-content/uploads/images/AbsolutelyTypical_UKOUG2011_jellema.zip
9. Doing SQL from PL/SQL: Best and Worst Practices
   http://www.oracle.com/technetwork/database/features/plsql/overview/doing-sql-from-plsql-129775.pdf