Integer Generators

Counting is fundamental to our culture, and we have learned it in the first chapter. Some SQL queries, however, require a little bit more sophistication than just counting rows. Sometimes there is no obvious candidate relation to count, while we still might want to generate an arbitrary number of rows and count them. Formally, what we need is the `Integers` relation which contains nothing more than a single numeric column with a list of positive integer numbers.

Surprisingly, there is no built-in `Integers` relation in SQL[1]. In the first half of this chapter, we'll learn about a dozen ways to cook it. In the second part we glimpse into numerous applications.

Integers Relation

As we have already mentioned, commercial databases don't come already equipped with the `Integers` relation. It is fairly easy to manufacture it, however. The solution differs between various vendors, so that we have to branch the story here.

Recursive With

DB2 has, arguably, the most satisfactory answer with *recursive SQL*

```
with Integers (num) as
( values(1)
  union all
  select num+1 from Integers
  where num < 100
)
select * from Integers
```

The execution flow literally follows the formal description. The `Integers` relation is defined via itself, this is why the adjective

[1] PostgreSQL user would probably smile and point out that there actually is a function `generate_series` that produces such a relation.

"recursive". Unlike other relational operators, which determine the result instantaneously and by purely logical means, the recursive definition works in steps. We start with the empty `Integers` relation, and add the first tuple `(num=1)`. Then, the recursive part

```
…
  select num+1 from Integers
  where num < 100
…
```

is ignited, and starts producing additional tuples. The recursion stops as soon as the recursive part is unable to generate more tuples.

The only imperfection is the hardcoded constant in the `where` condition. Did we really define the `Integers` relation, or just the `IntegersThatAreLessThan100`? Therefore, it is very tempting to move the whole condition to the outer query block where it fits more naturally

```
with Integers (num) as
( values(1)
  union all
  select num+1 from Integers
)
select * from Integers
where num < 100
```

Why this is such a good idea, and why it doesn't work will be a recurring theme in this chapter.

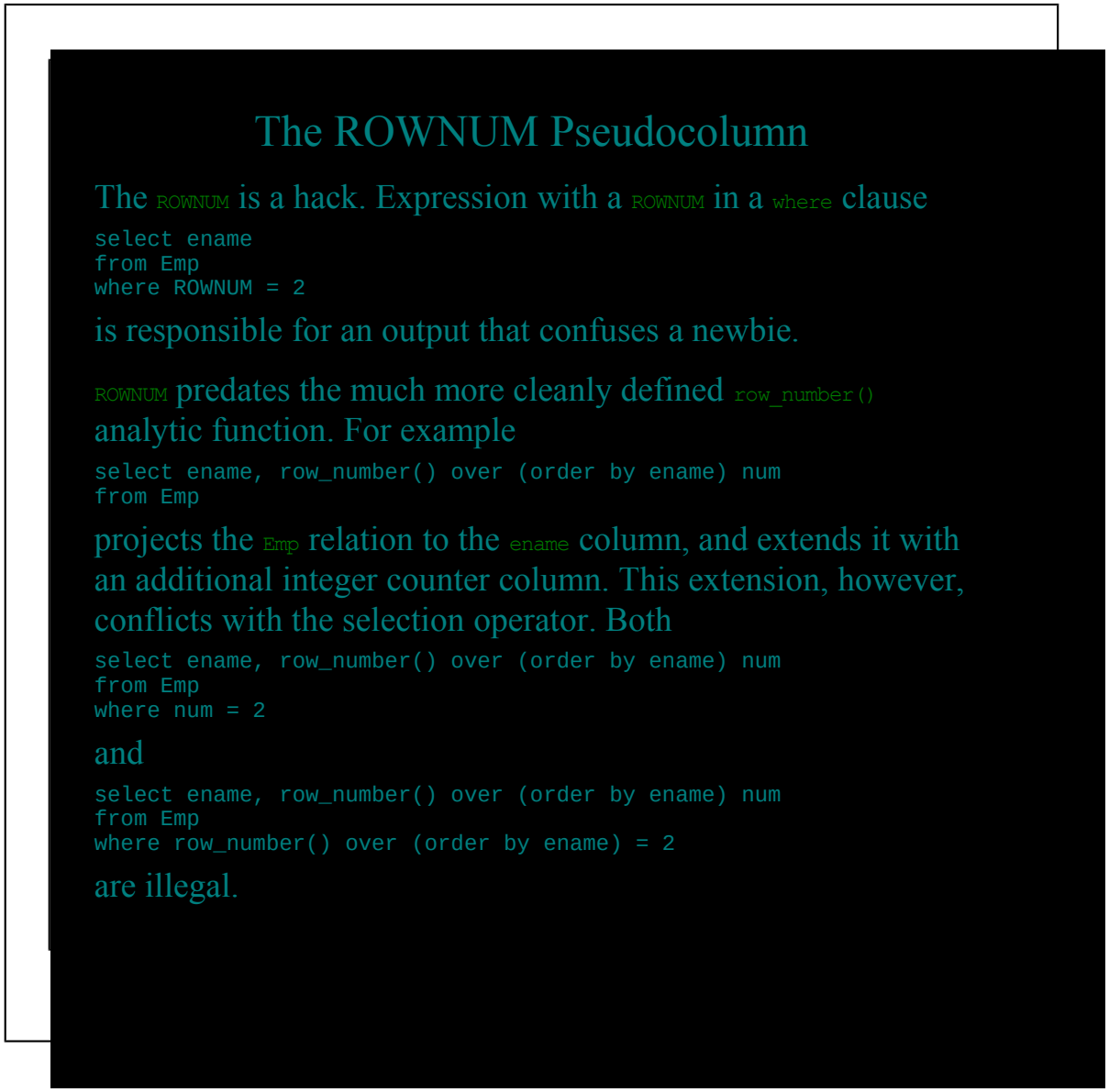
DB2 has enjoyed the recursive SQL feature for quite a while. This solution applies to Microsoft SQL Server circa 2005 as well. A reader interested in prior state of the art is advised to lookup Usenet Groups history on Google, since the question "What is the analog of *rownum* in MS Server?" was asked [and, perhaps, is still being asked] about every week on Microsoft SQL Server programming forum.

Big Table

There is about dozen of different ways to generate integers in Oracle. Moreover, every new release increases this number. The oldest and the least sophisticated method just hangs on the `rownum` column to some big table; the `all_objects` dictionary view being the most popular choice:

```
select rownum from all_objects where rownum < 100
```

The `rownum` is a pseudo column that assigns incremental integer values to the rows in the result set. This column name is Oracle proprietary syntax, hence the question from users of the other RDBMS platforms highlighted in the previous section.



The `all_objects` view is fairly complicated, though. From a performance perspective, the `sys.obj$` dictionary table is much better choice.

Table Function

Either way, the previous solution looks ridiculous to anybody with little programming background. Integers can be created on the fly

so easily, why does one have to refer to some stored relation, let alone a view over several tables? The idea of relations which can be manufactured with code as opposed to stored relations leads to the concept of *Table Function*. A table function is a piece of procedural code that can produce a result which can be understood by the SQL engine -- that is, a relation! Table function can have a parameter, so that the output relation depends on it. For all practical purposes it looks like a *Parameterized View*, and it is even called this way in the SQL Server world. The `Integers` view is naturally parameterized with an upper bound value.

The Table Function concept evolved further to embrace the idea of *pipelining*. From logical perspective, pipelining doesn't change anything: the table function output is still a relation. The function, however, doesn't require materializing the whole relation at once; it supplies rows one by one along with consumer's demand. This implementation difference, however, have a significant implication in our case. The `integers` relation is infinite! By no means can one hope materializing it in all entirety, hence the upper bound parameter in non-pipelined version mentioned in the previous paragraph.

The size of the pipelined table function output, however, doesn't necessarily have to be controlled by purely logical means. A row's producer (table function) is in intricate relationship with a row's consumer (the enclosing SQL query), and there is a way for the consumer to tell the producer to stop.

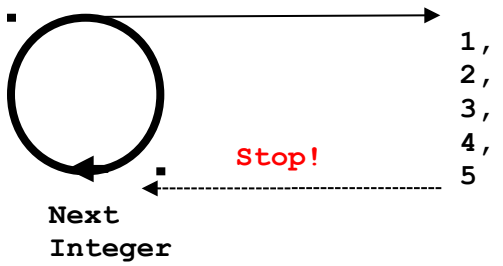


Figure 2.1: A producer generates integers in an infinite loop. As soon as an integer is built, it is shipped to a consumer. After digesting 5 integers the consumer decides that it had enough.

Let's write such table function producing infinite list of integers and see what happens:

```
CREATE TYPE IntSet AS TABLE OF Integer;

CREATE or replace FUNCTION Integers
  RETURN  IntSet PIPELINED IS
BEGIN
    loop
       PIPE ROW(0);
    end loop;
END;
/
```

Each table function is required to define the shape of the output it produces as an object type. We declared `IntSet` as a list of integers. The function implementation is unbelievably simple: the flow of control enters infinite loop, where it creates a new output row during each iteration. Since the function is capable producing a list of 0s only, it is the calling SQL query's responsibility to have a pseudo column expression that assigns integers incrementally. Alternatively, we could have a slightly more verbose `Integers` table function with a counter variable incremented during each loop iteration which pipes that integer to the output.

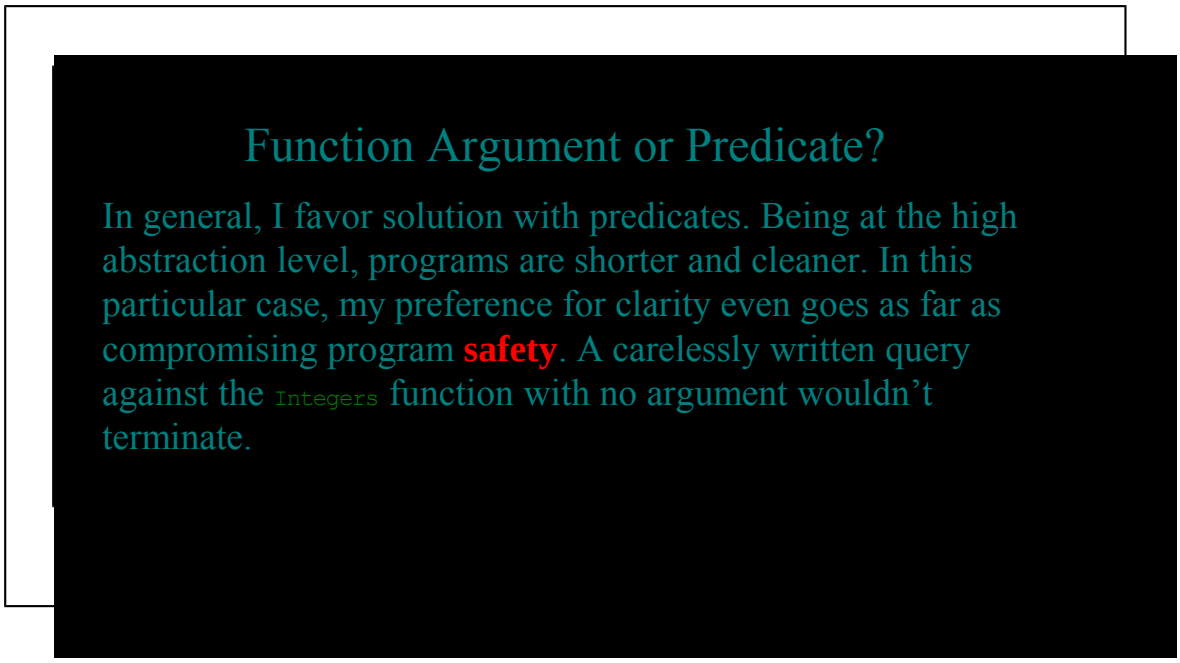
How do we use the `Integers` table function? Just

```
select rownum from Table(Integers)
```

would do, although we have to be careful. A typical client programmatic interface allows opening a cursor, and fetching the result set row by row. Normally, the client would stop when it exhausts the result set. In this case, the result set is infinite, so that the client has to decide by itself when to stop. Of course, the decision when to stop can be moved to server side, and made explicit

```
select rownum from Table(Integers)
where rownum < 1000
```

When designing a pipelined Integers function we neglected a rather popular alternative. Many would find it natural for the Integers function to have an argument which specifies an upper limit in the range of generated integers. For example, Integers(5)



returns the list 1,2,3,4,5. Then, the last could be reformulated in terms of this new function without the predicate

```
select rownum from Table(Integers(1000))
```

Which style is better?

Suppose somebody unfamiliar with the Integers function implementation asks:

What is the maximum number in the list of integers produced by this query?

Predicate `rownum <= 1000` makes the answer to the question immediate, while with the function parameter it might be `1000`, `999`, or even `21000` – it is impossible to tell for sure without examining the `Integers` function implementation.

The case when one would need raw or only slightly cooked list of integers is relatively simple. If the `integers` relation is a part of more complex query, it would require a more elaborate analysis to decide whether the query would terminate without explicit server-side stop condition.

Cube

One more approach leverages the `cube` operator

```
select rownum from (
    select 0
    from dual
    group by cube(1,1,1,1,1)
)
```

This solution had drawn the following eloquent comment from a reader named Mikito Harakiri:

> Most posters here[2] seem to have trouble seeing the difference between finite and infinite.
>
> OK, exponentiate the number, it is still produces a finite number (of rows). One day your program with this "cool" `cube` solution would break just because you have been lazy enough to code a trivial pipelined function. Essentially, `cube` is as bad as selecting from `obj$` (OK, `col$` is probably bigger). Well, except that you take your fellow programmer's time, who has to understand your code. What is the purpose of this `group by cube`, in general, and why there are 5 parameters (and, say, not 6), in particular.
> Table function method

```
select rownum from Table(Integers()) where rownum < 50
```

> is much cleaner and robust.

Hierarchical Query

I have mentioned already DB2 integer generator leveraging recursive SQL. Oracle doesn't have recursive SQL capabilities (at the time of this writing), so that users have to use non-standard hierarchical query extension. A contrast between the Oracle and DB2 solutions is often enlightening. One fundamental difference between these two platforms is that Oracle seems to be able to detect loops, while DB2 doesn't make such a claim. Detecting loops, in general, is undecideable, which is the basis for DB2's position. Does Oracle's loop detection work because hierarchical extension has narrowed query expression capabilities compared to recursive SQL? Can we challenge it?

Consider a typical hierarchical query

```
select ename from emp
connect by empno = prior mgr
start with empno = 1000
```

First, Oracle finds all the nodes in the graph satisfying the `start with` condition. Then, for each batch of nodes found on a previous step, it finds a batch of extra nodes that satisfy `the connect` by condition. Any time the new node collides with the nodes that have been already discovered, it signals the `connect by` loop error.

[2] Ask Tom forum, the thread "*how to display selective record twice in the query?*"

How does it identify the nodes? Should it compare all the relation attributes, or only those in the `select` clause, or choose some other ones? It is easy to see that the attributes in the `select` clause shouldn't matter. Indeed, adding a `rownum` pseudo column would artificially make the next node appear to be always different from its predecessors. The loop, however, is a property of the graph. Graph either has a cycle or not, no matter what node labels there may be. Therefore, the only columns which should be relevant for loop detection are the ones in the predicate with the `prior` prefix.

What if we write hierarchical query without the `prior`? This experiment reveals a remarkably succinct integer generator

```
select rownum from dual
connect by 0=0
```

As an alternative to the `rownum` pseudo column, we could use `level`

```
select level from dual
connect by 0=0
```

Both queries produce infinite result set. It is pipelined, however, so that the execution can be stopped either on the client, or explicitly on the server

```
select level from dual
connect by level < 100
```

Now that we have ample supply of integer generators, we can proceed with applications.

String Decomposition

Given a relation `A` with a single column `LST`

LST
1,3,7

the query should output a relation with tuples containing individual substrings that are "separated by commas" in the original strings

PREF
1
3
7

The clause "separated by commas" is decorated with quotation marks on purpose. While the value 3 is indeed enclosed with

commas on both sides, 1 and 7 are delimited from one side only. If we have even a remote hope to convert our informal requirement into the formal language, the first thing to do is fixing this nuisance. It is quite easy

```
select ','||LST||',' LST from A
```

LST
,1,3,7,

Now, **each** value is enclosed from both sides. The answer is a substring between `n`-th and `n+1`-th delimiter position. Getting a little bit ahead of myself, I'll mention that a very elegant recursive SQL solution awaits us later. We continue with one of Oracle integer number generators, therefore

```
with B as (select ','||LST||',' LST from A)
select substr(LST,instr(LST,',',1,num)+1,
                  instr(LST,',',1,num+1)-instr(LST,',',1,num)-1) PREF
from B,(
  select rownum num from dual connect by rownum < 10
)
```

PREF
1
3
7

Now that we have some solution, it is instructive to reflect back and notice that without the additional commas in the beginning and at the end of the string our solution would be quite messy. We would have to make separate cases for the values that are delimited from one side only. In a word, **rigorous** means **simple**!

Why does the query insist on producing exactly 9 integers? Well, 9 is a "reasonably big" number. Any number greater or equal to `LENGTH(LST)` would do. If we have more than one row in the original relation `A`, then we could simply take a maximum of the string lengths.

What about those rows with `NULL`s? Easy: anything unwanted could be filtered out with a proper predicate.

Admittedly, the last two answers taste like kludges. The solution feels even less satisfying if we compare it with the recursive SQL solution, which I promised before

```
with decomposed( pref, post ) as (
  select  '',  lst from A
  union
  select substr(lst, 1, instr(lst,',')),
         substr(lst, instr(lst,',')+1, length(lst)) from decomposed
) select pref from decomposed where pref <> ''
```

The execution steps can be easily visualized

Execution Step	Relation	New Tuples
0	Pref: ; Post: 1,3,7	Pref: 1; Post: 3,7
1	Pref: ; Post: 1,3,7 Pref: 1; Post: 3,7	Pref: 3; Post: 7
2	Pref: ; Post: 1,3,7 Pref: 1; Post: 3,7 Pref: 3; Post: 7	Pref: 7; Post:
3	Pref: ; Post: 1,3,7 Pref: 1; Post: 3,7 Pref: 3; Post: 7 Pref: 7; Post:	Pref; Post

Interestingly, we are no longer required to decorate strings with starting and trailing delimiter.

If your RDBMS of choice supports table functions, you might argue that the task of string parsing could be delegated to some

procedural code. Here is implementation in Oracle posted on the OTN SQL& PL/SQL forum by Scott Swank:

```
CREATE TYPE Strings AS TABLE OF varchar2(100);

FUNCTION parse (
    text        IN   VARCHAR2,
    delimiter   IN   VARCHAR2 DEFAULT ',')
    RETURN Strings
  IS
    delim_len   CONSTANT PLS_INTEGER      := LENGTH (delimiter);
    tokens               Strings   := Strings ();
    text_to_split        VARCHAR2 (32767);
    delim_pos            PLS_INTEGER;
  BEGIN
    tokens.DELETE;
    text_to_split := text;

    WHILE text_to_split IS NOT NULL
    LOOP
      delim_pos := INSTR (text_to_split, delimiter);

      IF delim_pos > 0
      THEN
        tokens.EXTEND;
        tokens (tokens.LAST) :=
                  SUBSTR (text_to_split, 1, delim_pos - 1);
        text_to_split :=
                  SUBSTR (text_to_split, delim_pos + delim_len);
      ELSE
        tokens.EXTEND;
        tokens (tokens.LAST) := text_to_split;
        text_to_split := NULL;
      END IF;
    END LOOP;

    RETURN tokens;
END parse;
```

Even though the implementation looks complicated, all that is visible from SQL query is just a function call

```
select column_value
from table(parse('1,2,3,4'));
```

It might not seem obvious how to call this table function in a context where the input data is a set of strings. We can't put a `table` expression inside some subquery within the `select` or `where` clause, since it returns multiple rows[3]. The only solution is keeping it inside the `where` clause, although the function parameter has to be correlated with the source table

```
create table sources (
   num integer,
   src varchar2(1000)
);

insert into sources values(1, '1.2.3,4.5,6,7');
insert into sources values(2, '8,9');

select
```

[3] Unless we expand our scope to a **nested** tables.

```
  num,
  t.*
from sources, table(parse(src)) t;
```

This syntax is reminiscent of ANSI SQL **lateral views**, where the second table expression is allowed to refer to the first one. From a theoretical perspective an expression with lateral view is an asymmetric join with join theta condition pushed inside the second table.

This solution scales up nicely. For example we can parse comma separated list, first, and then parse the result once more

```
select
  num,
  t1.column_value a,
  t2.column_value b
from sources t, table(parse(src)) t1,
table(parse(column_value,'.')) t2;
```

NUM	A	B
1	1.2.3	1
1	1.2.3	2
1	1.2.3	3
1	4.5	4
1	4.5	5
1	6	6
1	7	7
2	8	8
2	9	9

Enumerating Pairs

Enumerating pairs seems easy. Just build a Cartesian product of the two `integers` relations

```
select i1.num x, i2.num y
from Integers i1, Integers i2
```

With a finite `integers` relation (bounded by an arbitrary upper limit) there shouldn't be any surprises. What if it is infinite, can we drop the limit predicate on the server side and leverage the pipelining idea?

Pipelined Operators

When executing a SQL statement, the RDBMS engine represents it internally as a tree of *operators*. Each operator performs a relatively simple task; the complexity lies in the way the operators are combined together.

Each operator works as a black box. It consumes a relation as an input, massages it, and outputs the result to the other operator. If the operator is able to start outputting rows as soon as it consumed one or several rows, it is called *pipelined*. Pipelined operators don't block the execution flow; whereas *blocking* operators have to consume the whole input relation before outputting even a single row.

At first, the answer seems to depend on a join method employed by your SQL execution engine for this particular query. Both hash join and sorted merge join materialize their argument relations, therefore the execution plan involving any of those methods is never pipelined. Nested loops is typically a pipelined method. It iterates via the outer relation, and for each row finds matching rows from the inner relation. In case of the Cartesian product of Integer relations, however, all the rows from the inner relation match and there is an infinite number of them! Therefore, the execution would be stuck scanning the inner table, and would never be able to get past the first row in the outer table. In other words, the nested loop join fails to deliver pipelined set of integer pairs as well.

Let's approach the problem from another angle. When a mathematician says "enumerating" she really means a [one-to-one[4]] mapping some set of objects into the integers. Figure 2.2 shows a nearly ubiquitous mapping of integer pairs into integers.

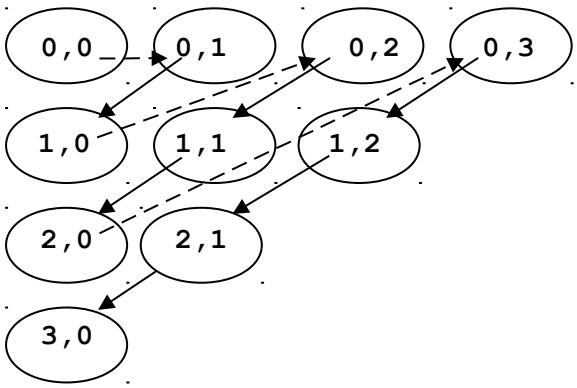


Figure 2.2: Enumerating all the pairs of integers. Each pair is assigned an integer which is the length of path to the origin. The origin (0,0) is mapped into 0, (0,1) is mapped into 1, and so on.

Without further ado, here are the formulas for the `(x,y) -> n` mapping

$$n := \frac{1}{2}(x + y)(x + y + 1) + y$$

and the reverse `n -> (x,y)` mapping

$$XplusY := \text{floor}\left(\frac{\sqrt{8n + 1} - 1}{2}\right)$$

$$x := n - \frac{(XplusY + 1)\,XplusY}{2}$$

$$y := XplusY - x$$

Proving them is left as an exercise for curious reader. Translating the above formulas into SQL is straightforward

```
select n,n-(xplusy+1)*xplusy/2 x,
  xplusy-n+(xplusy+1)*xplusy/2 y
from (
  select FLOOR((SQRT(8*(rownum-1)+1)-1)/2) xplusy,
  rownum-1 n from dual connect by 1=1
)
```

N	X	Y

[4] bijective

0	0	0
1	0	1
2	1	0
3	0	2
4	1	1
5	2	0
6	0	3
7	1	2
8	2	1
9	3	0
...	...	...

Even though the implementation side varies from vendor to vendor, it is instructive to see the execution statistics

OPERATION	OUTPUT ROWS
VIEW	10
COUNT	10
CONNECT BY	10
FAST DUAL	1

Ignoring fancy operator names in the execution statistics above, we see that the very first operator produces one row. This row performs a basis for `connect by` iteration that starts producing rows one-by-one. Each row is pipelined through two more levels of processing to the top. The client receives each row[5] and, after getting the tenth row, loses any interest continuing further. The execution statistics is a snapshot at that moment.

Admittedly, using the above pipelined integer pairs implementation inside other queries as inner view or subquery would only raise eyebrows. Those square root expressions are better be hidden behind a named view

```
create view IntegerPairs as
select n,n-(xplusy+1)*xplusy/2 x,
  xplusy-n+(xplusy+1)*xplusy/2 y
from (
  select FLOOR((SQRT(8*(rownum-1)+1)-1)/2) xplusy,
  rownum-1 n from dual connect by 1=1
)
```

With this pipelining idea that we push forward in integer pairs implementation did we achieve anything at all? Consider a query

Find positive integers X and Y satisfying both `x + y = 10` and `x - y = 2` equations

[5] In reality, the SQL programming interface buffers the result set, and transmit rows in batches of certain size.

With `IntegerPairs` view the solution is immediate

```
select x,y from IntegerPairs
where x+y=10 and x-y=2
```

Would it actually work? A pair `x=6 and y=5` is the unique solution of the system of equations, but would the execution really stop after we have it in the result set? Even if the client is not interested in more than one row from the result set, it has no way to communicate this to the server. If the result set is buffered, then the server would continue producing more pairs, and never stop.

Ad-Hoc Constants

If pipelining doesn't really work for constraint problems with two integer variables, why mention it at all? In an alternative approach we would just write

```
select x,y from (
   select num x from Integers where num < 234
), (
   select num y from Integers where num < 567
) where x+y=10 and x-y=2
```

which produces correct answer without any risk of getting stuck in the infinite loop. Much left is to be desired for the code clarity, however. Those "safety" constants are eyesore. They have to be estimated in advance. The values that work for one problem, might not work for next problem. A cartoon description of pipelining would define it as a programming style that avoids magic constants.

Enumerating Sets of Integers

When enumerating integer sets we map every integer into a set of integers[6]. Representing an integer in binary numbering system provides a natural way to implement such mapping. Formally, let

$$N = a_0 2^0 + a_1 2 + a_2 2^2 + a_3 2^3 + etc$$

where each a_i is either 0 or 1. Then, the sequence $(a_0, a_1, a_2, a_3, \ldots)$ is a characteristic vector of a set: i is an element of the set whenever $a_i = 1$. For example, $10 = 2^1 + 2^3$ maps into $\{1,3\}$. Here is integer set enumeration for up to N=10:

[6] When speaking of bijective mapping between integers and integer sets it is common to assume that the sets are **finite**, which we quietly adopted.

N	Integer Set
1	{0}
2	{1}
3	{0,1}
4	{2}
5	{0,2}
6	{1,2}
7	{0,1,2}
8	{3}
9	{0,3}
10	{1,3}

Unfortunately, we can't produce this output in SQL, at least, not yet. Later, we'll study composite data types and learn how to aggregate multiple values into strings and collections. For now, we build a relation "*i* is an element of set N":

N	*i*
1	0
2	1
3	0
3	1
4	2
5	0
5	2
6	1
6	2
7	0
7	1
7	2
8	3
9	0
9	3
10	1
10	3

The implementation is easy. We take the `IntegerPairs` relation and recruit the `BITAND` and `POWER` SQL functions to filter the unwanted pairs (N, *i*) pairs:

```
select y N, x i from IntegerPairs
where bitand(power(2,x),y)>0
```

As you might remember the idea of pipelining was somewhat discredited in the previous section. Yet it may be recovered for

our purpose. We take an infinite stream of integer pairs, apply a filter with the `bitand(power(2,x),y)>0` predicate, cut a finite segment of the result on the client. With non-pipelined `IntegerPairs` implementation (via Cartesian product of `Integers` relations) the execution would hang in an infinite loop.

Discrete Interval Sampling

Perhaps the second most common problem that utilizes the `Integers` relation is *unpacking* the intervals[7] or, leveraging Signal Processing terminology *Discrete Interval Sampling*.

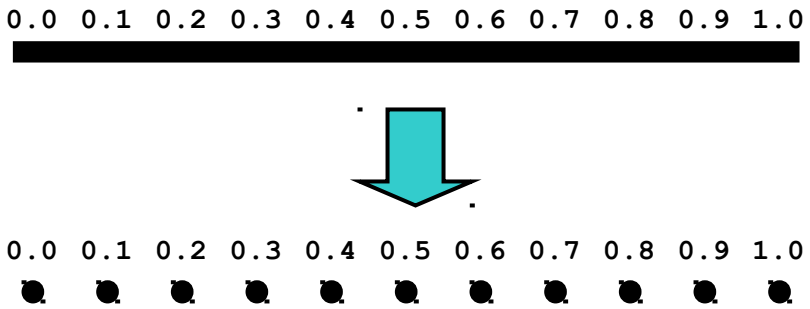


Figure 2.3: Discrete Interval Sampling example. Interval [0,1] is a continuum of points. We select 11 discrete points at the offset 0.1 each.

Reduced to essentials the problem is the following. Given two dates, `'1-jan-2005'` and `'31-dec-2005'`, for example, return the list of the days in between. This is trivial, of course, if we already have the `Dates` relation filled in with the dates:

```
select day from Dates
where day between '1-jan-2005' and '31-dec-2005'
```

If we don't have the `Dates` relation at hand, why don't we just manufacture it? All what is required is support of datetime/interval arithmetic by the RDBMS of your choice. Perhaps, the easiest approach is adding numeric values to the date

[7] C. J. Date, Hugh Darwen, and Nikos A. Lorentzos. Temporal Data and the Relational Model. 2003. Morgan Kaufmann.

represented in the `date` data type[8]. For example, adding 1 to today's `date` produces tomorrow's `date`. Likewise, `to_date('1-jan-2005')+1` is `to_date('2-jan-2005')`, and so on. Therefore,

```
select to_date('1-jan-2005')+num
from Integers
where to_date('1-jan-2005')+num <= to_date('31-dec-2005')
```

will generate a list of days spanning the whole year. This toy query is frequently encountered as a part of more challenging puzzles.

Consider hotel reservation application. The `Bookings` table stores reservations made against a set of rooms

RoomNo	"From"[9]	"To"	Customer
1	2005-05-07	2005-05-09	Jim
2	2005-05-08	2005-05-10	Steve
1	2005-05-11	2005-05-14	Bob

Users typically query what rooms are empty during a certain period. For example, querying all available rooms in the time period from 2005-05-07 to 2005-05-12 should return

RoomNo	Day
2	2005-05-07
1	2005-05-10
2	2005-05-11
2	2005-05-12

We start with two relations: the first containing the list of all the rooms, and the second containing all the dates between 2005-05-07 and 2005-05-12. It is very tempting to define the first relation as

```
select distinct RoomNo from Bookings
```

It is possible, however, that a particular room is always empty, and therefore, has no records in the `Bookings` table. The whole `Bookings` table can be empty! Let's assume that we have the list of all rooms as one more relation -- the `Rooms` table.

We are just one step from the solution. If we generate a Cartesian product of the two relations, we can filter out the unwanted results (again, with scalar subquery!)

```
select RoomNo, Day
from (
```

[8] The ANSI SQL equivalent to oracle's `date` is `timestamp` datatype.

[9] The "From" and "To" are SQL keywords, which is why they are decorated with quotation marks

```
    select to_date('07-May-2005')+num Day
    from Integers
    where to_date('07-May-2005')+num <= to_date('12-May-2005')
), Rooms r
where 0 = ( select count(*) from Bookings b
            where r.RoomNo = b.RoomNo
            and Day between "From" and "To"
);
```

Although the query has written itself effortlessly, much is left to be desired about its performance. A seasoned SQL performance analyst would immediately spot a couple of problems with this solution. Building a Cartesian product, first, and evaluating every resulting tuple with a subquery doesn't sound very promising. But the Cartesian product is unavoidable: if the `Bookings` relation is empty, then the result *must* be the Cartesian product. It is an interesting research question, whether one can reduce the cost associated with subquery evaluation in this example.

An alternative approach to the problem would reformulate the query from a business perspective.

- What is the list of rooms available during a certain interval?
- What is the average ratio of occupied to empty rooms during a certain period?

Both of these queries can be answered without discrete interval sampling.

Summary

- Don't refer to any stored relation when generating integers.
- Recursive `with` integer generator is the cleanest solution.
- Integers provide a basis for more complex structures: integer pairs, sets of integers, etc.
- Integers are essential for iterative methods, e.g. String Decomposition, Discrete Interval Sampling.

Exercises

1. PostgreSQL has the `generate_series` function which has 3(!) arguments. Given `start`, `stop`, and `step` numbers it would

produce a series of integers beginning with `start`, incremented by `step`, with the last element not exceeding the `stop`. We have already discussed some controversial issues related to stop condition. The `start` and `step`, however, are much simpler matter. Prove that they are redundant.

2. In section "String Decomposition" can't we push forward the pipelining idea, and get rid of the stop predicate `rownum < 10` altogether? Try changing it to the `0 = 0` tautology and see what happens. Check up the execution plan, is it is pipelined? Try to influence the optimizer to use different join methods.
3. In the previous exercise, what if the condition filtering out unwanted `NULL`s is added? Could the execution possibly stop?
4. Can any of the integer generators that you studied in this chapter be abstracted into the `Integers` view with infinite number of rows? What do you think is missing for a SQL engine to support such a view?
5. Derive the formulas for mapping integer pairs into integers.
6. Given a set of integers, for example

N
3
5

interpreted as the right interval boundaries, fill in each interval `[0,N]` with integers, e.g.

N	*i*
3	1
3	2
3	3
5	1
5	2
5	3
5	4
5	5

Make sure the query is safe, in other words the auxiliary list of integers is bounded by the maximum integer from the original list.

7. Explain the following obfuscated query:

```
select -1/2+1/2*sqrt(1+8*sum(rownum)) from Emp
```

8. One way to check for balanced parenthesis across a string is to increment and decrement a running sum as each opening or closing parenthesis is encountered[10], eg

```
(a or b) and (c or (d))
11111110000001111112210
```

Write a query that produces the running sum output:

Offset	Parenthesis Level
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	0
9	0
10	0
11	0
12	0
13	0
14	1
15	1
16	1
17	1
18	1
19	1
20	2
21	2
22	1

Hint: apply a combination of techniques from chapters 1 and 2. You need two integer generators: one for the offset, and the second one for conditional summation.

[10] Kendall Willets, private communications