Converting RPG to Set Based SQL (Part 1: Input Operations)

Document version: 1.1

Author: Michael R. Jones

Date: 09/27/2015

Contents

Introduction

This document is intended to assist current or former RPG language programmers to use SQL queries more effectively. This is a learn-by-example document, containing commonly used RPG language input operation code (op code) combinations, and their SQL query code equivalents, using **set** based SQL.

This document does not attempt to provide a full discussion of the reasons why you should use SQL for your database input and output (I/O) needs, versus using RPG. Parts of this document help make that argument, but no attempt has been made to fully explore the pros and cons of replacing RPG I/O with SQL. Instead, this document assumes you've either made the decision to do so, or you've learned enough about SQL to be considering making that switch. The examples here should be very effective at teaching you just how easy it is to convert RPG I/O to SQL, and, once your SQL skills have reached a reasonable level, how much easier SQL is to develop, read, and maintain.

Prior to studying the code examples provided here, I recommend you already have a basic understanding of SQL SELECT queries, in particular the SELECT, FROM, JOIN, WHERE, and ORDER BY clauses. There are numerous, free, online, training websites available to you to gain an introductory understanding of SQL SELECT queries. Basic SQL does vary from one database to another, but not dramatically. It is desirable but not necessary to use an online SQL training website specifically targeted to your particular database in use, in order to learn the basics of SQL queries.

RPG I/O op codes perform I/O one row at a time. While SQL can be structured to perform I/O one row at a time, it should NOT be used in that manner in nearly all cases. SQL really shines when you structure your query requests to produce full SETS of rows from a single SQL query request. **I can't stress enough the importance and benefits of mastering set based thinking, design, and coding when using SQL.** The benefits of SQL performing one row at a time I/O are significant but minimal. On the other hand, the benefits of using SQL to process entire result sets at a time are huge (greatly improved performance, simplicity, a greatly reduced volume of code, easier code maintenance, and reduced chance for defects). All the key SQL statements support processing entire result sets at a time: SELECT, INSERT, UPDATE, MERGE, CREATE TABLE (populated using SELECT).

The examples in this document provide many of the foundational SQL building blocks you need to know to begin the process of learning how to request complex sets of data from the database. If you master the examples provided here, you are well on your way to building any result set of rows using SQL. This book focuses on constructing SQL result sets corresponding to RPG examples. It does not cover, in sufficient detail, how to take those result sets and process them as a result set. In other words, it does not, to a significant degree, show you how to take a SELECT query set of rows and feed them to an INSERT, or an UPDATE, or a MERGE statement. That will be covered in a separate book (part 2 in this series on set based SQL). The focus of this book, constructing SQL result sets, is the logical first step in mastering set based SQL processing.

Assumptions / notes for all code examples

All RPG and SQL examples:

1) TABLE_A has a **UNIQUE** key of KEY_COLUMN.
2) A **one-to-many** TABLE_A row to TABLE_B rows relationship is assumed in all examples.
3) Process all TABLE_A rows.

All RPG examples:

1) TABLE_A rows are read into a qualified data structure named "A".
2) TABLE_B rows are read into a qualified data structure named "B".
3) Columns are referenced like this: A.KEY_COLUMN, B.KEY_COLUMN, etc.

All SQL examples:

1) The examples assume use of IBM DB2® on IBM i®, running a version of the IBM i® operating system, formerly known as i5/OS™ and OS/400®, that supports LATERAL joins. LATERAL joins go back to at least OS version V5R4M0, and possibly earlier.

2) In some respects, the query abilities of DB2® on other platforms tends to be ahead of DB2® on IBM i®. The SQL in this document will likely be valid syntax on relatively recent versions of DB2® on other platforms, but I've not attempted to validate that is the case.
3) The SQL examples build the full result set of desired rows, with data joined together from multiple tables.
4) Except in rare cases, you should be using SQL to join together all your data from multiple tables in a single SQL statement, as much as possible, as opposed to issuing one SQL statement per table and piecing the data together in RPG memory. Instead, use the SQL engine to stitch all your data together. In nearly all cases, using the SQL engine to stitch all your data together will result in better performance and greater overall simplicity. There are times when improved performance results by splitting a single SQL query into two or more queries, but those cases are very uncommon, and typically involve very complex queries.
5) In the examples that follow, the code to fetch and process the SQL result set of rows is intentionally not shown. Excluding the fetch code was done for these reasons:
    a) To keep examples short.
    b) No assumption is made you'll be consuming your SQL result sets with RPG.
    c) RPG code to fetch data from an SQL result set is very different than, for example, Oracle® Java® code to perform the same fetch.
    d) My recommendation is to use languages other than RPG that are more capable at handling today's more demanding functional requirements.
    e) Many applications, in particular batch job applications, in order to maximize performance, should be structured so they do NOT use single row fetches from a cursor inside a loop. Instead, batch jobs should be structured to process a full result set at a time for each SQL statement issued. I've been migrating very large, very complex databases of ~1700 tables from a largely denormalized database to a largely normalized one, without using a single cursor or a single fetch statement. If I can do that, with sufficient result set design and coding skills in your skillset, you certainly can structure your batch jobs to not use cursors or single row fetch statements. A full discussion of the various techniques available to do that is beyond the scope of this book, but this book should help you make great progress towards being able to do just that.
6) If you do not know how to embed SQL in RPG, or how to fetch SQL result set rows in RPG, there are numerous training resources available to provide that information. That training is not provided in this book.
7) Many of the examples use "select *", in order to simplify the SQL examples, but you should NOT use "select *" in production programs, except in very rare cases. One exception is using "select *" to access a temporary work table, which is a reasonable thing to do.
8) Like any programming language, SQL code can be structured in many different ways to accomplish the same thing. Do not assume the SQL examples in this book are the best for your situation. The examples are structured to balance flexibility and simplicity, and to illustrate concepts. They are not necessarily the best design for all scenarios. However, I believe the coding patterns used in this book are easy to learn, are very flexible, can be easily extended to build very complex result sets, and tend to perform very well when extended.
9) Real world code paralleling these examples would likely use similar ORDER BY clauses, so these examples include them. Some real world examples will not require an ORDER BY clause to order the rows of the final result set, but most will.
10) The ORDER BY clause in the LATERAL JOINs in these examples, are required in order to produce the same results as the RPG. Note: ORDER BY clauses inside a LATERAL JOIN do not guarantee the order of the rows delivered in the final result set, hence some of these examples have an ORDER BY inside a LATERAL JOIN, and a second ORDER BY clause to guarantee the order of rows in the final result set.

SETLL / READ examples (#01 to #04)

Assumptions / notes:

1) TABLE_B has a **UNIQUE** key of KEY_COLUMN, DATE_COLUMN

Example #01:

- Process ALL rows from TABLE_B per TABLE_A row with matching key and date column ON or AFTER a specified date (2015-01-01).
- Process TABLE_B rows in ascending date order.
- Matching key row required in TABLE_B: If no matching key TABLE_B row, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN : %date( '2015-01-01' ) ) TABLE_B;
   read TABLE_B;
   dow not %eof( TABLE_B )
   and A.KEY_COLUMN = B.KEY_COLUMN;
      // process TABLE_A, TABLE_B row combination here
      read TABLE_B;
   enddo;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
inner join TABLE_B  B
      on   B.KEY_COLUMN    = A.KEY_COLUMN           --matching key rows only
      and  B.DATE_COLUMN >= date( '2015-01-01' )  --satisfy date filter
order by   A.KEY_COLUMN, B.DATE_COLUMN
```

Notes:

- INNER JOIN requires a true condition, in this case, rows with a matching key between the tables.
- Given the keys on TABLE_A and TABLE_B, the ORDER BY clause is necessary to build the SQL result set of rows in the same order as the RPG code would process them.
- Most applications will need the rows processed in a specific order. In cases where your application truly does not require the rows in a specific order, do not use an ORDER BY clause, and in nearly all cases, you will see a reduced runtime.
- SQL is a declarative language, meaning you specify the desired end result, but SQL determines how to accomplish the result. Do not assume that SQL will return the rows in your desired order without an ORDER BY clause. If you need the rows in a specific order, always specify an ORDER BY clause to guarantee the rows are in the result set in that order.
- Consistent with being a declarative language, SQL determines, for you, which indexes to use when processing a query. Always reference physical table / file names in your SQL. Never reference logical file names. SQL will figure out the indexes to use to maximize performance.
- This is an extremely common SQL coding pattern. Notice how much simpler the SQL is than the RPG.
- You should also take note that the SQL has a reduced chance for defects than the RPG, because RPG has to deal with looping and end of file, whereas SQL does not. This reduced chance for defects applies to all the examples.
- Granted, if you embed the SQL in an RPG program and perform a FETCH loop through the rows, you're still dealing with loop code. However, there are many types of applications where you will not need, and should not use (for performance reasons), fetch loops using cursors. An example is performing "`insert into TABLE_A select * from TABLE_C`" in SQL (no loop used and not desired for performance reasons). INSERT INTO followed by a SELECT will insert an entire result set of rows into the target table using a single statement.

Example #02:

- Process up to ONE matching key row from TABLE_B per TABLE_A row, with matching key and date column ON or AFTER a specified date (2015-01-01). The TABLE_B row chosen for each TABLE_A row is the row with the earliest DATE_COLUMN value on or after the specified date.
- Process TABLE_B rows in ascending date order.
- Matching key row required in TABLE_B: if no matching key TABLE_B row, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN : %date( '2015-01-01' ) ) TABLE_B;
   read TABLE_B;
   if not %eof( TABLE_B )
   and A.KEY_COLUMN = B.KEY_COLUMN;
      // process TABLE_A, TABLE_B row combination here
   endif;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
cross join lateral (
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN    = A.KEY_COLUMN          --matching key rows only
     and   B.DATE_COLUMN >= date( '2015-01-01' )  --satisfy date filter
  order by B.DATE_COLUMN   --deliver row with earliest DATE_COLUMN value
  fetch    first row only  --select one row per matching TABLE_A key
) as B
order by   A.KEY_COLUMN
```

Notes:

- A subquery is a query within a query.
- In this example, the RPG picks only one TABLE_B row per TABLE_A row. It picks the TABLE_B row with the earliest date on or after date 2015-01-01. The LATERAL join used here simulates that one row choice.
- LATERAL joins permit joining to a subquery, where the subquery references columns added to the result set earlier in the query, through the use of correlation names ("A" in this case). When used with ORDER BY and FETCH clauses inside the sub-query, LATERAL joins greatly simplify the task of picking any number of rows out of many, and, in my opinion, are underutilized by many SQL developers on the DB2® for IBM i® database.
- CROSS JOINs do not use an ON clause. A CROSS JOIN LATERAL without an ON clause is sufficient in this case, because the row matching takes place in the WHERE clause inside the subquery. Many lateral joins where the row selection takes place inside the lateral subquery do not need an ON clause, and in many cases can therefore use a CROSS JOIN.
- CROSS JOIN LATERAL to one matching row in the subquery does NOT increase the number of rows in the result set. All it does is tack on more columns to the set, assuming a row match is found. "Fetch first row only" ensures a max of one row is joined to.
- CROSS JOIN LATERAL to zero matching rows in the subquery discards the row from the result set. In this case, if no matching TABLE_B row is found, the TABLE_A row is discarded, behaving the same way an INNER JOIN does upon no row match.
- Since only one TABLE_B row is selected per matching key TABLE_A row, there is no need to use "order by A.KEY_COLUMN, B.DATE_COLUMN".

Example #03:

- Process up to TWO matching key rows from TABLE_B per TABLE_A row, with matching key and date column value ON or AFTER a specified date (2015-01-01). The TABLE_B rows chosen for each TABLE_A row are the TWO rows with the earliest DATE_COLUMN values on or after the specified date.
- Process TABLE_B rows in ascending date order.
- Matching key row is required in TABLE_B: if no matching key TABLE_B row is present, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN : %date( '2015-01-01' ) ) TABLE_B;
   for x = 1 to 2;
      read TABLE_B;
      if %eof( TABLE_B )
      or A.KEY_COLUMN <> B.KEY_COLUMN;
         leave;
      endif;
      // process TABLE_A, TABLE_B row combination here
   endfor;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
cross join lateral (
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN    = A.KEY_COLUMN          --matching key rows only
      and  B.DATE_COLUMN >= date( '2015-01-01' )  --satisfy date filter
  order by B.DATE_COLUMN      --deliver rows with earliest DATE_COLUMN values
  fetch    first 2 rows only  --corresponds to "for x = 1 to 2"
) as B
order by   A.KEY_COLUMN, B.DATE_COLUMN
```

Notes:

- This is the same as example #2, except up to two TABLE_B rows can be joined to each TABLE_A row.
- A CROSS JOIN LATERAL of one TABLE_A row to only one matching TABLE_B row will result in one row in the result set for the matching TABLE_A key.
- A CROSS JOIN LATERAL of one TABLE_A row to two matching TABLE_B rows will result in two rows in the result set for the matching TABLE_A key.
- CROSS JOIN LATERAL to zero matching rows in the subquery discards the TABLE_A row from the result set.
- If TABLE_B has three or more matching rows, only the first two with the earliest DATE_COLUMN values on or after the specified date will be included in the result set.
- Because the result set can contain more than one TABLE_B row per TABLE_A row, the ORDER BY clause also contains B.DATE_COLUMN, to guarantee the rows are delivered in the same order as RPG would process them (based on the keys assumed for the tables).

Example #04:

- Process up to THREE matching key rows from TABLE_B per TABLE_A row, with matching key and date column value ON or AFTER a specified date (2015-01-01). The TABLE_B rows chosen for each TABLE_A row are the THREE rows with the earliest DATE_COLUMN values on or after the specified date.
- Process TABLE_B rows in ascending date order.
- This time, a matching key row is NOT required in TABLE_B in order to process the TABLE_A row, which differs from the previous three examples. If no matching key row in TABLE_B, the TABLE_A row is NOT discarded from result set, and is processed. Because we want to process all TABLE_A rows, even if no TABLE_B row match, CROSS JOIN is replaced with LEFT JOIN in this example.

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN : %date( '2015-01-01' ) ) TABLE_B;
   read TABLE_B;
   if not %eof( TABLE_B )
   and A.KEY_COLUMN = B.KEY_COLUMN;
      for x = 1 to 3;
         // process TABLE_A, TABLE_B row combination here
         if x < 3;  // prevent unnecessary 4th read
            read TABLE_B;
            if %eof( TABLE_B )
            or A.KEY_COLUMN <> B.KEY_COLUMN;
               leave;
            endif;
         endif;
      endfor;
   else;
      // process TABLE_A row, by itself here, since no TABLE_B row match was found.
   endif;
enddo;
```

SQL:

```
select     A.*
          ,coalesce( B.DATE_COLUMN, date( '0001-01-01' ) )  as B_DATE_COLUMN
          ,coalesce( B.COLUMN_1, 0 )    as B_COLUMN_1
          ,coalesce( B.COLUMN_2, '' )   as B_COLUMN_2
          --etc. for additional TABLE_B columns, as needed
from       TABLE_A  A
left join lateral (  --optional row match left join delivers all TABLE_A rows
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN    = A.KEY_COLUMN          --matching key rows only
      and  B.DATE_COLUMN >= date( '2015-01-01' )  --satisfy date filter
  order by B.DATE_COLUMN      --deliver rows with earliest DATE_COLUMN values
  fetch    first 3 rows only  --corresponds to "for x = 1 to 3"
) as B on 1 = 1
order by   A.KEY_COLUMN, B.DATE_COLUMN
```

Notes:

- The COALESCE function returns the first non-null argument passed to it, or null if all arguments are null.
- Do not use COALESCE if you prefer or need to receive NULL indicators when a left join row is not found. However, keep in mind that, in some languages, RPG in particular, doing so requires extra code to handle NULL indicators.
- COALESCE is an enhanced version of the IFNULL function. COALESCE supports more than two arguments, whereas IFNULL only supports two.
- LEFT JOIN LATERAL is used to satisfy the requirement to process a TABLE_A row when no matching key TABLE_B row is present. LEFT JOINs (LATERAL or not), will never decrease the number of result set rows, but are capable of increasing the number of rows in the result set (when joining to more than one row). LEFT JOIN is the same as LEFT OUTER JOIN.
- If you prefer, you can use "on B.KEY_COLUMN is not null" instead of "on 1 = 1" to produce the same result.
- Technically, the ON clause of a JOIN is simply looking for a true or false condition. When true, the join is satisfied, when false, it is not. In this case, because the row matching is performed in the WHERE clause of the subquery, and a LEFT JOIN is in use which requires an ON clause, we only need an ON clause that always evaluates to true (e.g. 1 = 1). In this case, the ON clause does not perform row matching like it does in a typical join.

SETLL / READE examples (#05 to #08)

Assumptions / notes:

2) TABLE_B has a **UNIQUE** key of KEY_COLUMN, SEQUENCE_#

Example #05:

- Process ALL rows from TABLE_B per TABLE_A row with matching key.
- Process TABLE_B rows in ascending SEQUENCE_# order.
- Matching key row required in TABLE_B: if no matching key TABLE_B row, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN ) TABLE_B;
   dou %eof( TABLE_B );
      reade ( A.KEY_COLUMN ) TABLE_B;
      if not %eof( TABLE_B );
         // process TABLE_A, TABLE_B row combination here
      endif;
   enddo;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
inner join TABLE_B  B  --inner join discards TABLE_A row if no TABLE_B match
      on   B.KEY_COLUMN = A.KEY_COLUMN  --matching key rows only
order by   A.KEY_COLUMN, B.SEQUENCE_#
```

Notes:

- Both the RPG and SQL examples here represent commonly used coding patterns.
- Because this SQL is requesting an entire result set, in one request, containing all TABLE_A rows, joined to all matching TABLE_B rows, performance should be better than the RPG equivalent. Typically, the more rows in the result set, the more SQL will outperform RPG. This is because you're requesting the result set using a single SQL statement, versus requesting one row at a time from the database in RPG. There is a very significant amount of overhead in each input / output request, and performance generally improves significantly if you reduce the number of input / output requests passed to the database. In general, the more you reduce the number of I/O requests executed, the better the performance.
- Given the keys on TABLE_A and TABLE_B, the ORDER BY clause is necessary to deliver the rows in the SQL result set in the same order as the RPG code would process them.

Example #06:

- Process only ONE matching key row from TABLE_B per TABLE_A row. The TABLE_B row chosen for each TABLE_A row is the row with the smallest SEQUENCE_# value.
- Matching key row required in TABLE_B: if no matching key TABLE_B row, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN ) TABLE_B;
   if %equal( TABLE_B );
      reade ( A.KEY_COLUMN ) TABLE_B;
      if not %eof( TABLE_B );
         // process TABLE_A, TABLE_B row combination here
      endif;
   endif;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
cross join lateral (
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN = A.KEY_COLUMN  --matching key rows only
  order by B.SEQUENCE_#    --deliver the TABLE_B row with lowest SEQUENCE_#
  fetch    first row only  --select one TABLE_B row per TABLE_A row
) as B
order by   A.KEY_COLUMN
```

Notes:

- CROSS JOIN LATERAL to one matching row in the subquery does NOT increase the number of rows in the result set. All it does is tack on more columns to the set, assuming a row match is found. "Fetch first row only" ensures a max of one row is joined to.
- CROSS JOIN LATERAL to zero matching rows in the subquery discards the row from the result set. In this case, if no matching TABLE_B row is found, the TABLE_A row is discarded from the result set.
- Since only one TABLE_B row is selected per matching key TABLE_A row, there is no need to use "order by A.KEY_COLUMN, B.SEQUENCE".

Example #07:

- Process up to TWO matching key rows from TABLE_B per TABLE_A row. The TABLE_B rows chosen are the TWO rows with the smallest SEQUENCE_# values.
- Process TABLE_B rows in ascending SEQUENCE_# order.
- Matching key row is required in TABLE_B: if no matching key TABLE_B row is present, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN ) TABLE_B;
   if %equal( TABLE_B );
      for x = 1 to 2;
         reade ( A.KEY_COLUMN ) TABLE_B;
         if %eof( TABLE_B );
            leave;
         endif;
         // process TABLE_A, TABLE_B row combination here
         endif;
      endfor;
   endif;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
cross join lateral (
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN = A.KEY_COLUMN  --matching key rows only
  order by B.SEQUENCE_#       --deliver rows with lowest SEQUENCE_#
  fetch    first 2 rows only  --corresponds to "for x = 1 to 2"
) as B
order by   A.KEY_COLUMN, B.SEQUENCE_#
```

Notes:

- This is the same as example #6, except up to two TABLE_B rows can be joined to each TABLE_A row.
- A CROSS JOIN LATERAL of one TABLE_A row to one matching TABLE_B row will result in one row in the result set for the matching TABLE_A key.
- A CROSS JOIN LATERAL of one TABLE_A row to two matching TABLE_B rows will result in two rows in the result set for the matching TABLE_A key.
- CROSS JOIN LATERAL to zero matching rows in the subquery discards the TABLE_A row from the result set.
- If TABLE_B has three or more matching rows, only the first two with the lowest SEQUENCE_# will be included in the result set.
- Because the result set can contain more than one TABLE_B row per TABLE_A row, the ORDER BY clause also contains B.SEQUENCE_#, to guarantee the rows are delivered in the same order as RPG would process them.

Example #08:

- Process up to THREE matching key rows from TABLE_B per TABLE_A row.
- Process TABLE_B rows in ascending SEQUENCE_# order.
- This time, a matching key row is NOT required in TABLE_B in order to process the TABLE_A row, which differs from the previous three examples. If no matching key row in TABLE_B, the TABLE_A row is NOT discarded from result set, and is processed.
- For each TABLE_A row, up to THREE rows from TABLE_B are processed (the rows with the smallest SEQUENCE_# values).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN ) TABLE_B;
   if %equal( TABLE_B );
      for x = 1 to 3;
         reade ( A.KEY_COLUMN ) TABLE_B;
         if %eof( TABLE_B );
            leave;
         endif;
         // process TABLE_A, TABLE_B row combination here
         endif;
      endfor;
   else;
      // process TABLE_A row here, by itself, when no TABLE_B row match
   endif;
enddo;
```

SQL:

```
select     A.*
          ,coalesce( B.SEQUENCE_#, 0 ) as B_SEQUENCE_#
          ,coalesce( B.COLUMN_1, 0 )   as B_COLUMN_1  --assume numeric
          ,coalesce( B.COLUMN_2, '' )  as B_COLUMN_2  --assume alpha
         --etc. for additional TABLE_B columns, as needed.
from       TABLE_A  A
left join lateral (  --optional row match left join delivers all TABLE_A rows
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN = A.KEY_COLUMN  --matching key rows only
  order by B.SEQUENCE_#                 --deliver rows with lowest SEQUENCE_#
  fetch    first 3 rows only            --corresponds to "for x = 1 to 3"
) as B on 1 = 1
order by   A.KEY_COLUMN, B.SEQUENCE_#
```

Notes:

- LEFT JOIN LATERAL is used to satisfy the requirement to process a TABLE_A row when no matching key TABLE_B row is present. LEFT JOINs (LATERAL or not), will never decrease the number of result set rows, but are capable of increasing the number of rows (when more than one row matches the join), in the result set. LEFT JOIN is the same as LEFT OUTER JOIN.
- It is common to use COALESCE, or IFNULL, when processing columns pulled into a result set by a LEFT JOIN. If you need or prefer NULL when a TABLE_B row is not found, do not use COALESCE and simply refer to the TABLE_B columns. For some applications, an indication of NULL is desirable, for others, it is not (blanks, zeros, or other default value is sufficient in place of a NULL indication). In RPG, handling NULLs requires adding extra code for NULL indicators.
- If you prefer, you can use "on B.KEY_COLUMN is not null" instead of "on 1 = 1" to produce the same result.
- Technically, the ON clause of a JOIN is simply looking for a true or false condition. When true, the join is satisfied, when false, it is not. In this case, because the row matching is performed in the WHERE clause of the subquery, and a LEFT JOIN is in use which requires an ON clause, we only need an ON clause that always evaluates to true (e.g. 1 = 1). In this case, the ON clause does not perform row matching like it does in a typical join.

SETGT / READE examples (#9 to #12)

Assumptions / notes:

1) TABLE_B has a **UNIQUE** key of KEY_COLUMN, DATE_COLUMN

Example #09:

- Process ALL rows from TABLE_B per TABLE_A row with matching key and a date column value AFTER a specified date (2014-12-31).
- Process TABLE_B rows in ascending date order.
- Matching key row required in TABLE_B: if no matching key TABLE_B row, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setgt ( A.KEY_COLUMN : %date( '2014-12-31' ) ) TABLE_B;
   dou %eof( TABLE_B );
      reade ( A.KEY_COLUMN ) TABLE_B;
      if not %eof( TABLE_B );
         // process TABLE_A, TABLE_B row combination here
      endif;
   enddo;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
inner join TABLE_B  B
      on   B.KEY_COLUMN  = A.KEY_COLUMN          --matching key rows only
      and  B.DATE_COLUMN > date( '2014-12-31' ) --satisfy date filter
order by   A.KEY_COLUMN, B.DATE_COLUMN
```

Notes:

- All matching key rows between the tables are delivered, but only when TABLE_B.DATE_COLUMN is after the specified date.
- For a given TABLE_A key, if TABLE_B does not contain a matching key row DATE_COLUMN after the specified date, the TABLE_A row is discarded from the result set, which is standard INNER JOIN behavior for "no row match". INNER JOIN requires a true condition, in this case, rows with a matching key between the tables.
- Because multiple TABLE_B rows are possible per matching TABLE_A key, an ORDER BY clause containing DATE_COLUMN is used, to ensure the result set is delivered with the rows in the same order as RPG would process them.

Example #10:

- Process up to ONE matching key row from TABLE_B per TABLE_A row, with a date column value AFTER a specified date (2014-12-31). The TABLE_B row for each matching TABLE_A row is the row with the smallest DATE_COLUMN value after the specified date.
- Process TABLE_B rows in ascending date order.
- Matching key row required in TABLE_B: if no matching key TABLE_B row, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setgt ( A.KEY_COLUMN : %date( '2014-12-31' ) ) TABLE_B;
   reade ( A.KEY_COLUMN ) TABLE_B;
   if not %eof( TABLE_B );
      // process TABLE_A, TABLE_B row combination here
   endif;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
cross join lateral (
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN  = A.KEY_COLUMN          --matching key rows only
      and  B.DATE_COLUMN > date( '2014-12-31' )  --satisfy date filter
  order by B.DATE_COLUMN   --deliver row with earliest DATE_COLUMN value
  fetch    first row only  --only one TABLE_B row per TABLE_A row
) as B
order by   A.KEY_COLUMN
```

Notes:

- CROSS JOIN LATERAL to one matching key row does NOT increase the number of rows in the result set. All it does is tack on more columns to the set, assuming a row match is found. "Fetch first row only" ensures a max of one row is joined to.
- CROSS JOIN LATERAL to zero rows (no matching row), discards the TABLE_A row from the result set, just like an INNER JOIN does if the join ON clause evaluates to no match (false).
- In this example, use of "fetch first row only" produces at most one TABLE_B row per TABLE_A row.
- If you prefer, you can use an INNER JOIN LATERAL with either "on 1 = 1" or "on B.KEY_COLUMN is not null", to produce the same results. Essentially, "on B.KEY_COLUMN is not null" means a row matching the subquery WHERE clause for TABLE_B row must be found.
- Because only one TABLE_B matching row per TABLE_A row is delivered, DATE_COLUMN is not necessary in the ORDER BY clause.

Example #11:

- Process up to TWO matching key rows from TABLE_B per TABLE_A row, with a date column value AFTER a specified date (2014-12-31). The TABLE_B rows chosen for each matching TABLE_A row are the TWO rows with the smallest DATE_COLUMN values AFTER date 2014-12-31.
- Process TABLE_B rows in ascending date order.
- Matching key row is required in TABLE_B: if no matching key TABLE_B row is present, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setgt ( A.KEY_COLUMN : %date( '2014-12-31' ) ) TABLE_B;
   for x = 1 to 2;
      reade ( A.KEY_COLUMN ) TABLE_B;
      if %eof( TABLE_B );
         leave;
      endif;
      // process TABLE_A, TABLE_B row combination here
   endfor;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
cross join lateral (
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN  = A.KEY_COLUMN          --matching key rows only
     and   B.DATE_COLUMN > date( '2014-12-31' )  --satisfy date filter
  order by B.DATE_COLUMN       --deliver rows with earliest DATE_COLUMN values
  fetch    first 2 rows only   --corresponds to "for x = 1 to 2"
) as B
order by   A.KEY_COLUMN, B.DATE_COLUMN
```

Notes:

- CROSS JOIN LATERAL to one matching key row does NOT increase the number of rows in the result set. All it does is tack on more columns to the set, assuming a row match is found.
- CROSS JOIN LATERAL to zero rows (no matching row), discards the TABLE_A row from the result set, just like an INNER JOIN does if the join ON clause evaluates to no match (false).
- In this example, use of "fetch first 2 row only" produces up to two TABLE_B rows per TABLE_A row, selecting, at most, the two TABLE_B matching key rows with the lowest DATE_COLUMN values after the specified date.
- Because multiple TABLE_B rows are possible per matching TABLE_A key, an ORDER BY clause containing DATE_COLUMN is used, to ensure the result set is delivered with the rows in the same order as RPG would process them.

Example #12:

- Process up to THREE matching key rows from TABLE_B per TABLE_A row, with a date column value AFTER a specified date (2014-12-31). The TABLE_B rows chosen for each matching TABLE_A row are the THREE rows with the smallest DATE_COLUMN values AFTER date 2014-12-31.
- Process TABLE_B rows in ascending date order.
- A matching key row is NOT required in TABLE_B in order to process the TABLE_A row. If no matching key row in TABLE_B, the TABLE_A row is NOT discarded from result set, and is processed.

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setgt ( A.KEY_COLUMN : %date( '2014-12-31' ) ) TABLE_B;
   reade ( A.KEY_COLUMN ) TABLE_B;
   if not %eof( TABLE_B );
      for x = 1 to 3;
         // process TABLE_A, TABLE_B row combination here
         if x < 3;  // prevent unnecessary 4th read
            reade ( A.KEY_COLUMN ) TABLE_B;
            if %eof( TABLE_B );
               leave;
            endif;
         endif;
      endfor;
   else;
       // process TABLE_A row, by itself here, since no TABLE_B match was found.
   endif;
enddo;
```

SQL:

```
select     A.*
          ,coalesce( B.DATE_COLUMN, date( '0001-01-01' ) )  as B_DATE_COLUMN
          ,coalesce( B.COLUMN_1, 0 )   as B_COLUMN_1
          ,coalesce( B.COLUMN_2, '' )  as B_COLUMN_2
         --etc. for additional TABLE_B columns, as needed
from       TABLE_A  A
left join lateral (  --optional row match left join delivers all TABLE_A rows
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN  = A.KEY_COLUMN          --matching key rows only
      and  B.DATE_COLUMN > date( '2014-12-31' )  --satisfy date filter
  order by B.DATE_COLUMN      --deliver rows with earliest DATE_COLUMN values
  fetch    first 3 rows only  --corresponds to "for x = 1 to 3"
) as B on 1 = 1
order by   A.KEY_COLUMN, B.DATE_COLUMN
```

Notes:

- LEFT JOIN LATERAL is used to satisfy the requirement to process a TABLE_A row when no matching key TABLE_B row is present. LEFT JOINs (LATERAL or not), will never decrease the number of result set rows, but are capable of increasing the number of rows, in the result set. LEFT JOIN is the same as LEFT OUTER JOIN.
- If you prefer, you can use "on B.KEY_COLUMN is not null" instead of "on 1 = 1" to produce the same result.

SETLL / READP examples (#13 to #16)

Assumptions / notes:

1) TABLE_B has a **UNIQUE** key of KEY_COLUMN, DATE_COLUMN

Example #13:

- Process ALL rows from TABLE_B per TABLE_A row with matching key and date column BEFORE today's date.
- Process TABLE_B rows in descending date order.
- Matching key row required in TABLE_B: If no matching key TABLE_B row, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN : %date() ) TABLE_B;
   readp TABLE_B;
   dow not %eof( TABLE_B )
   and A.KEY_COLUMN = B.KEY_COLUMN;
      // process TABLE_A, TABLE_B row combination here
      readp TABLE_B;
   enddo;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
inner join TABLE_B  B
      on   B.KEY_COLUMN  = A.KEY_COLUMN       --matching key rows only
      and  B.DATE_COLUMN < CURRENT_DATE       --satisfy date filter
order by   A.KEY_COLUMN, B.DATE_COLUMN desc  --desc corresponds to readp
```

Notes:

- There is nothing significantly new conceptually in this example versus the previous examples.
- This example is provided simply because the above RPG coding pattern is a common one, and it is the intent of author to cover the most commonly used RPG input / output coding patterns.
- By now, you should notice two trends in these examples:
  - The variations in the RPG and SQL code are minor across examples, and
  - The SQL code is easier to read. If you do not find that to be the case yet, you should soon.
- To reduce repetition in the example notes, the volume of explanation decreases significantly in the examples that follow.

Example #14:

- Process up to ONE matching key row from TABLE_B per TABLE_A row, with matching key and date column value BEFORE today's date. The TABLE_B row chosen for each matching TABLE_A row is the row with the latest DATE_COLUMN value BEFORE today's date.
- Process TABLE_B rows in descending date order.
- Matching key row required in TABLE_B: if no matching key TABLE_B row, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN : %date() ) TABLE_B;
   readp TABLE_B;
   if not %eof( TABLE_B )
   and A.KEY_COLUMN = B.KEY_COLUMN;
      // process TABLE_A, TABLE_B row combination here
   endif;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
cross join lateral (
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN  = A.KEY_COLUMN  --matching key rows only
      and  B.DATE_COLUMN < CURRENT_DATE  --satisfy date filter
  order by B.DATE_COLUMN desc  --deliver row with latest DATE_COLUMN value
  fetch    first row only      --select one row per matching TABLE_A key
) as B
order by   A.KEY_COLUMN
```

Notes:

- Again, there is nothing significantly new conceptually in this example versus the previous examples.
- This example is provided simply because the above RPG coding pattern is a common one, and it is the intent of author to cover the most commonly used RPG input / output coding patterns (i.e. commonly used RPG input operation code combinations).

Example #15:

- Process up to TWO matching key rows from TABLE_B per TABLE_A row, with matching key and date column value BEFORE today's date. The TABLE_B rows chosen for each matching TABLE_A row are the TWO rows with the latest DATE_COLUMN values BEFORE today's date.
- Process TABLE_B rows in descending date order.
- Matching key row is required in TABLE_B: if no matching key TABLE_B row is present, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN : %date() ) TABLE_B;
   for x = 1 to 2;
      readp TABLE_B;
      if %eof( TABLE_B )
      or A.KEY_COLUMN <> B.KEY_COLUMN;
         leave;
      endif;
      // process TABLE_A, TABLE_B row combination here
   endfor;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
cross join lateral (
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN  = A.KEY_COLUMN       --matching key rows only
      and  B.DATE_COLUMN < CURRENT_DATE       --satisfy date filter
  order by B.DATE_COLUMN desc     --deliver rows with latest DATE_COLUMN values
  fetch    first 2 rows only      --corresponds to "for x = 1 to 2"
) as B
order by   A.KEY_COLUMN, B.DATE_COLUMN desc  --desc corresponds to readp
```

Notes:

- This is a minor variation of previous examples. Refer to previous example explanations if needed.

Example #16:

- Process up to THREE matching key rows from TABLE_B per TABLE_A row, with matching key and date column value BEFORE today's date. The TABLE_B rows chosen for each matching TABLE_A row are the THREE rows with the latest DATE_COLUMN values BEFORE today's date.
- Process TABLE_B rows in descending date order.
- A matching key row is NOT required in TABLE_B in order to process the TABLE_A row. If no matching key row in TABLE_B, the TABLE_A row is NOT discarded from result set, and is processed.

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN : %date() ) TABLE_B;
   readp TABLE_B;
   if not %eof( TABLE_B )
   and A.KEY_COLUMN = B.KEY_COLUMN;
      for x = 1 to 3;
         // process TABLE_A, TABLE_B row combination here
         if x < 3;  // prevent unnecessary 4th read
            readp TABLE_B;
            if %eof( TABLE_B )
            or A.KEY_COLUMN <> B.KEY_COLUMN;
               leave;
            endif;
         endif;
      endfor;
   else;
      // process TABLE_A row, by itself here, since no TABLE_B match was found.
   endif;
enddo;
```

SQL:

```
select     A.*
          ,coalesce( B.DATE_COLUMN, date( '0001-01-01' ) )  as B_DATE_COLUMN
          ,coalesce( B.COLUMN_1, 0 )   as B_COLUMN_1
          ,coalesce( B.COLUMN_2, '' )  as B_COLUMN_2
          --etc. for additional TABLE_B columns, as needed
from       TABLE_A  A
left join lateral (  --optional row match left join delivers all TABLE_A rows
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN  = A.KEY_COLUMN      --matching key rows only
      and  B.DATE_COLUMN < CURRENT_DATE      --satisfy date filter
  order by B.DATE_COLUMN desc     --deliver rows with latest DATE_COLUMN values
  fetch    first 3 rows only      --corresponds to for x = 1 to 3
) as B on 1 = 1
order by   A.KEY_COLUMN, B.DATE_COLUMN desc  --desc corresponds to readp
```

Notes:

- This is a minor variation of previous examples. Refer to previous example explanations if needed.

SETGT / READP examples (#17 to #19)

Assumptions / notes:

1) TABLE_B has a **UNIQUE** key of KEY_COLUMN, DATE_COLUMN

Example #17:

- Process ALL rows from TABLE_B per TABLE_A row with matching key.
- Process TABLE_B rows in descending date order.
- Matching key row required in TABLE_B: if no matching key TABLE_B row, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setgt ( A.KEY_COLUMN ) TABLE_B;
   dou %eof( TABLE_B )
   or A.KEY_COLUMN <> B.KEY_COLUMN;
      readp TABLE_B;
      if not %eof( TABLE_B )
      and A.KEY_COLUMN = B.KEY_COLUMN;
         // process TABLE_A, TABLE_B row combination here
      endif;
   enddo;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
inner join TABLE_B  B
      on   B.KEY_COLUMN = A.KEY_COLUMN       --matching key rows only
order by   A.KEY_COLUMN, B.DATE_COLUMN desc  --desc corresponds to readp
```

Notes:

- This is a minor variation of previous examples. Refer to previous example explanations if needed.

Example #18:

- Process up to TWO matching key rows from TABLE_B per TABLE_A row. The TABLE_B rows chosen for the matching TABLE_A row are the TWO rows with the most recent DATE_COLUMN values.
- Process TABLE_B rows in descending date order.
- Matching key row is required in TABLE_B: if no matching key TABLE_B row is present, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setgt ( A.KEY_COLUMN ) TABLE_B;
   for x = 1 to 2;
      readp TABLE_B;
      if %eof( TABLE_B )
      or A.KEY_COLUMN <> B.KEY_COLUMN;
         leave;
      endif;
      // process TABLE_A, TABLE_B row combination here
   endfor;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
cross join lateral (
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN = A.KEY_COLUMN       --matching key rows only
  order by B.DATE_COLUMN desc   --deliver rows with latest DATE_COLUMN values
  fetch    first 2 rows only    --corresponds to for x = 1 to 2
) as B
order by   A.KEY_COLUMN, B.DATE_COLUMN desc  --desc corresponds to readp
```

Notes:

- This is a minor variation of previous examples. Refer to previous example explanations if needed.

Example #19:

- Process up to THREE matching key rows from TABLE_B per TABLE_A row. The TABLE_B rows chosen for the matching TABLE_A row are the THREE rows with the most recent DATE_COLUMN values.
- Process TABLE_B rows in descending date order.
- A matching key row is NOT required in TABLE_B in order to process the TABLE_A row.
- If no matching key row in TABLE_B, the TABLE_A row is NOT discarded from result set, and is processed.

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN ) TABLE_B;
   if %equal( TABLE_B );
      setgt ( A.KEY_COLUMN ) TABLE_B;
      for x = 1 to 3;
         readp TABLE_B;
         if %eof( TABLE_B )
         or A.KEY_COLUMN <> B.KEY_COLUMN;
           leave;
         endif;
         // process TABLE_A, TABLE_B row combination here
      endfor;
   else;
      // process TABLE_A row here by itself, since no TABLE_B match found.
   endif;
enddo;
```

SQL:

```
select     A.*
          ,coalesce( B.DATE_COLUMN, date( '0001-01-01' ) )  as B_DATE_COLUMN
          ,coalesce( B.COLUMN_1, 0 )   as B_COLUMN_1
          ,coalesce( B.COLUMN_2, '' )  as B_COLUMN_2
          --etc. for additional TABLE_B columns, as needed
from       TABLE_A  A
left join lateral (  --optional row match left join delivers all TABLE_A rows
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN = A.KEY_COLUMN       --matching key rows only
  order by B.DATE_COLUMN desc   --deliver rows with latest DATE_COLUMN values
  fetch    first 3 rows only    --corresponds to for x = 1 to 3
) as B on 1 = 1
order by   A.KEY_COLUMN, B.DATE_COLUMN desc  --desc corresponds to readp
```

Notes:

- This is a minor variation of previous examples. Refer to previous example explanations if needed.

SETLL / READPE examples (#20 to #23)

Assumptions / notes:

1) TABLE_B has a **UNIQUE** key of KEY_COLUMN, DATE_COLUMN

Example #20:

- Process ALL rows from TABLE_B with matching key and date column value BEFORE today's date.
- Process TABLE_B rows in descending date order.
- Matching key row required in TABLE_B: if no matching key TABLE_B row, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN : %date() ) TABLE_B;
   dou %eof( TABLE_B );
      readpe ( A.KEY_COLUMN ) TABLE_B;
      if not %eof( TABLE_B );
         // process TABLE_A, TABLE_B row combination here
      endif;
   enddo;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
inner join TABLE_B  B
      on   B.KEY_COLUMN  = A.KEY_COLUMN       --matching key rows only
      and  B.DATE_COLUMN < CURRENT_DATE       --satisfy date filter
order by   A.KEY_COLUMN, B.DATE_COLUMN desc  --desc corresponds to readpe
```

Notes:

- This is a minor variation of previous examples. Refer to previous example explanations if needed.

Example #21:

- Process up to ONE matching key row from TABLE_B per TABLE_A row, with matching key and date column value BEFORE today's date. The TABLE_B row chosen for the matching TABLE_A row is the row with the latest DATE_COLUMN value BEFORE today's date.
- Process TABLE_B rows in descending date order.
- Matching key row required in TABLE_B: if no matching key TABLE_B row, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN : %date() ) TABLE_B;
   readpe ( A.KEY_COLUMN ) TABLE_B;
   if not %eof( TABLE_B );
      // process TABLE_A, TABLE_B row combination here
   endif;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
cross join lateral (
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN  = A.KEY_COLUMN  --matching key rows only
      and  B.DATE_COLUMN < CURRENT_DATE  --satisfy date filter
  order by B.DATE_COLUMN desc  --deliver row with latest DATE_COLUMN value
  fetch    first row only      --max of 1 TABLE_B row per TABLE_A row
) as B
order by   A.KEY_COLUMN
```

Notes:

- This is a minor variation of previous examples. Refer to previous example explanations if needed.

Example #22:

- Process up to TWO matching key rows from TABLE_B per TABLE_A row, with matching key and date column value BEFORE today's date. The TABLE_B rows chosen for the matching TABLE_A row are the TWO rows with the latest DATE_COLUMN values BEFORE today's date.
- Process TABLE_B rows in descending date order.
- Matching key row is required in TABLE_B: if no matching key TABLE_B row is present, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN : %date() ) TABLE_B;
   for x = 1 to 2;
      readpe ( A.KEY_COLUMN ) TABLE_B;
      if %eof( TABLE_B );
         leave;
      endif;
      // process TABLE_A, TABLE_B row combination here
      endif;
   endfor;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
cross join lateral (
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN  = A.KEY_COLUMN       --matching key rows only
      and  B.DATE_COLUMN < CURRENT_DATE       --satisfy date filter
  order by B.DATE_COLUMN desc   --deliver rows with latest DATE_COLUMN values
  fetch    first 2 rows only    --corresponds to for x = 1 to 2
) as B
order by   A.KEY_COLUMN, B.DATE_COLUMN desc  --desc corresponds to readpe
```

Notes:

- This is a minor variation of previous examples. Refer to previous example explanations if needed.

Example #23:

- Process up to THREE matching key rows from TABLE_B per TABLE_A row, with matching key and date column value BEFORE today's date. The TABLE_B rows chosen for the matching TABLE_A row are the THREE rows with the latest DATE_COLUMN values BEFORE today's date.
- Process TABLE_B rows in descending date order.
- A matching key row is NOT required in TABLE_B in order to process the TABLE_A row.
- If no matching key row in TABLE_B, the TABLE_A row is NOT discarded from result set, and is processed.

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN : %date() ) TABLE_B;
   readpe ( A.KEY_COLUMN ) TABLE_B;
   if not %eof( TABLE_B );
      for x = 1 to 3;
         // process TABLE_A, TABLE_B row combination here
         if x < 3;  // prevent unnecessary 4th read
            readpe ( A.KEY_COLUMN ) TABLE_B;
            if %eof( TABLE_B );
               leave;
            endif;
         endif;
      endfor;
   else;
      // process TABLE_A row here by itself, since no TABLE_B match found.
   endif;
enddo;
```

SQL:

```
select     A.*
          ,coalesce( B.DATE_COLUMN, date( '0001-01-01' ) )  as B_DATE_COLUMN
          ,coalesce( B.COLUMN_1, 0 )    as B_COLUMN_1
          ,coalesce( B.COLUMN_2, '' )   as B_COLUMN_2
          --etc. for additional TABLE_B columns, as needed
from       TABLE_A  A
left join lateral (  --optional row match left join delivers all TABLE_A rows
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN  = A.KEY_COLUMN      --matching key rows only
      and  B.DATE_COLUMN < CURRENT_DATE      --satisfy date filter
  order by B.DATE_COLUMN desc   --deliver rows with latest DATE_COLUMN values
  fetch    first 3 rows only    --corresponds to for x = 1 to 3
) as B on 1 = 1
order by   A.KEY_COLUMN, B.DATE_COLUMN desc  --desc corresponds to readpe
```

Notes:

- This is a minor variation of previous examples. Refer to previous example explanations if needed.

SETGT / READPE examples (#24 to #26)

Assumptions / notes:

1) TABLE_B has a **UNIQUE** key of KEY_COLUMN, DATE_COLUMN

Example #24:

- Process ALL rows from TABLE_B per TABLE_A row with matching key.
- Process TABLE_B rows in descending date order.
- Matching key row required in TABLE_B: if no matching key TABLE_B row, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setgt ( A.KEY_COLUMN ) TABLE_B;
   dou %eof( TABLE_B );
      readpe ( A.KEY_COLUMN ) TABLE_B;
      if not %eof( TABLE_B );
         // process TABLE_A, TABLE_B row combination here
      endif;
   enddo;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
inner join TABLE_B  B
      on   B.KEY_COLUMN = A.KEY_COLUMN       --matching key rows only
order by   A.KEY_COLUMN, B.DATE_COLUMN desc  --desc corresponds to readpe
```

Notes:

- This is a minor variation of previous examples. Refer to previous example explanations if needed.

Example #25:

- Process up to TWO matching key rows from TABLE_B per TABLE_A row, in descending date column order. The TABLE_B rows chosen for the matching TABLE_A row are the TWO rows with the latest DATE_COLUMN values.
- Matching key row is required in TABLE_B: if no matching key TABLE_B row is present, the TABLE_A row is discarded (i.e. not processed).
- For each TABLE_A row, up to TWO rows from TABLE_B are processed (the two rows).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
     leave;
   endif;
   setgt ( A.KEY_COLUMN ) TABLE_B;
   for x = 1 to 2;
      readpe ( A.KEY_COLUMN ) TABLE_B;
      if %eof( TABLE_B );
         leave;
      endif;
      // process TABLE_A, TABLE_B row combination here
   endfor;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
cross join lateral (
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN = A.KEY_COLUMN       --matching key rows only
  order by B.DATE_COLUMN desc    --deliver rows with latest DATE_COLUMN values
  fetch    first 2 rows only     --corresponds to for x = 1 to 2
) as B
order by   A.KEY_COLUMN, B.DATE_COLUMN desc  --desc corresponds to readpe
```

Notes:

- This is a minor variation of previous examples. Refer to previous example explanations if needed.

Example #26:

- Process up to THREE matching key rows from TABLE_B per TABLE_A row. The TABLE_B rows chosen for the matching TABLE_A row are the THREE rows with the latest DATE_COLUMN values.
- Process TABLE_B rows in descending date column order.
- A matching key row is NOT required in TABLE_B in order to process the TABLE_A row. If no matching key row in TABLE_B, the TABLE_A row is NOT discarded from result set, and is processed.

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   setll ( A.KEY_COLUMN ) TABLE_B;
   if %equal( TABLE_B );
      setgt ( A.KEY_COLUMN ) TABLE_B;
      for x = 1 to 3;
         readpe ( A.KEY_COLUMN ) TABLE_B;
         if %eof( TABLE_B );
           leave;
         endif;
         // process TABLE_A, TABLE_B row combination here
       endfor;
    else;
       // process TABLE_A row here by itself, since no TABLE_B match found.
   endif;
enddo;
```

SQL:

```
select     A.*
          ,coalesce( B.DATE_COLUMN, date( '0001-01-01' ) )  as B_DATE_COLUMN
          ,coalesce( B.COLUMN_1, 0 )   as B_COLUMN_1
          ,coalesce( B.COLUMN_2, '' )  as B_COLUMN_2
          --etc. for additional TABLE_B columns, as needed
from       TABLE_A  A
left join lateral (  --optional row match left join delivers all TABLE_A rows
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN = A.KEY_COLUMN       --matching key rows only
  order by B.DATE_COLUMN desc    --deliver rows with latest DATE_COLUMN values
  fetch    first 3 rows only     --corresponds to for x = 1 to 3
) as B on 1 = 1
order by   A.KEY_COLUMN, B.DATE_COLUMN desc  --desc corresponds to readpe
```

Notes:

- This is a minor variation of previous examples. Refer to previous example explanations if needed.

CHAIN example (#27)

Assumptions / notes:

1) TABLE_B has a **UNIQUE** key of KEY_COLUMN, SEQUENCE_#

Example #27:

- Process only ONE matching key row from TABLE_B per TABLE_A row. The TABLE_B row chosen for the matching TABLE_A row is the row with the smallest SEQUENCE_# value.
- Matching key row required in TABLE_B: if no matching key TABLE_B row, the TABLE_A row is discarded (i.e. not processed).

RPG:

```
dou %eof( TABLE_A );
   read TABLE_A;
   if %eof( TABLE_A );
      leave;
   endif;
   chain ( A.KEY_COLUMN ) TABLE_B;
   if %found( TABLE_B );
      // process TABLE_A, TABLE_B row combination here
   endif;
enddo;
```

SQL:

```
select     *
from       TABLE_A  A
cross join lateral (
  select   B.*
  from     TABLE_B  B
  where    B.KEY_COLUMN = A.KEY_COLUMN  --matching key rows only
  order by B.SEQUENCE_#                 --deliver row with lowest SEQUENCE_#
  fetch    first row only               --one TABLE_B row per matching key
) as B
order by   A.KEY_COLUMN
```

Notes:

- This example is an alternate way, using RPG, to code example #6. The RPG is different, but the SQL is the same.

Complex result sets

The examples provided here are all fairly simple solutions to relatively simple examples. However, many real world RPG programs are building, albeit typically one row at a time, much more complex sets of data in memory, coming from more than two tables, and in some cases many more than two. You may be concerned that SQL can't handle those building those result sets. **In virtually every case, SQL absolutely can handle building those complex result sets.** A robust discussion of how to efficiently build complex result sets is beyond the scope of this document, and warrants a very lengthy discussion unto itself.

One important tip for building complex result sets, which especially applies when your skill level has a lot of room for improvement, is to add only one table to the result set at a time, and to perform unit testing of the revised query after each single table is added. Build and unit test in steps, where each step is a single table added to the query. As your skill progresses, you find less need to use a step based build / test approach, unless you're working on extremely complex result set logic, in which case the step based approach should still be used even at an expert skill level.

SQL support for complexity

When you look at the fine print of an SQL Reference manual for most databases, you see support for complexity almost everywhere you look. To help illustrate the vision of SQL's ability to produce complex result sets, consider the following partial capabilities for DB2® on IBM i®:

```
select     [column-reference]
         --,etc.
from       [table-reference]
inner join [table-reference]
      on   [join-condition]
left join  [table-reference]
      on   [join-condition]
--etc. joining many tables w/ inner, left, right, full outer, and exception joins
where      [where-clause]
group by   [group by-clause]
having     [having-clause]
order by   [order by-clause]
```

[column-reference]

In addition to a simple column reference, these can also reference subqueries inside parenthesis, complex calculations referring to a mixture of data from table reference columns, references to scalar functions supplied by the database and/or user defined scalar functions, and/or host program variables.

[table-reference]

In addition to a simple table name, these can also reference table functions, SQL views, common table expressions (reusable subqueries), and/or subqueries enclosed in parentheses, where each can contain their own complex logic.

[join-condition]

In addition to a simple clause tying columns together, these can also reference complex calculated expressions, host program variables, basic predicate subqueries, and/or scalar functions.

[where-clause]

In addition to a simple filter referencing columns, these can include complex calculations referring to a mixture of data from table references, scalar functions supplied by the database and/or user defined scalar functions, and/or host program variables. You can also reference subqueries here inside parenthesis.

[group by-clause]

In addition to grouping data through simple column references, these can include complex aggregate calculations referring to a mixture of data from table references, aggregate functions supplied by the database and/or user defined aggregate functions, and host program variables. You can also reference subqueries here inside parenthesis.

[having-clause]

In addition to a simple grouping filter referencing columns, these can include complex calculations referring to a mixture of grouped data from table references, aggregate functions supplied by the database and/or user defined aggregate functions, and host program variables. You can also reference subqueries here inside parenthesis.

[order by-clause]

In addition to simple columns references, these can include complex calculations referring to a mixture of data from table references, scalar functions supplied by the database and/or user defined functions, and host program variables. You can also reference subqueries here inside parenthesis.

Note: full support for these more robust abilities varies from one database platform to another, and within a platform, from one database version to another. When working on different databases or versions, there are times when you have to code things a different way to accomplish the same result.

Complex example #28

Following is an example of using fairly complex SQL to produce a complex result set. This is by no means a full discussion of how to produce complex results sets using SQL. The goals for including this example are:

- The previous examples provided an introduction to thinking, designing, and coding in terms of SQL result sets. This complex example should give you a taste of SQL's abilities to support building complex result sets.
- Prior to seeing this next example, many readers will have thought this next example wasn't even possible in SQL. Not only is it possible, but SQL supports much more complexity than demonstrated in this next example.
- I hope this motivates you to consider using SQL instead of RPG when you need a complex result set.

This example is not intended as something most readers will be able to understand without a great deal of study, depending on your current SQL skill level. If you don't have the time to fully understand the SQL in this example at this point in time, please try to do the following:

- Envision the result set that is being produced from the comments and column names in the final result set delivered.
- Imagine how much larger the code volume would be if you were to produce the same result set in RPG.

It typically takes a developer longer to learn how to think, design, and code in terms of result sets, than it does to become highly proficient with SQL. Although, since the best benefits of using SQL are yielded when performing SET based SQL, one can make a good argument that you're not highly proficient with SQL until you're highly proficient with working with sets. This next example introduces both advanced result set thinking, plus advanced SQL, and will overwhelm many readers when trying to understand every aspect of the code. At a minimum, I hope you come away from this example with the thought "Wow, I had no idea you could do that in SQL".

Assumptions / notes:

1) TABLE_B has a **UNIQUE** key of KEY_COLUMN, DATE_COLUMN.
2) This example introduces a new column TABLE_B.DOLLAR_COLUMN.
3) There are many ways to code the SQL to produce the same result set as in this example. The choices made in this example's SQL code were mainly chosen to build upon the techniques used in the previous examples, and to show how they can be extended. The resulting code is not intended to be the absolute best way to code this example.
4) This example is provided only in SQL, and not in RPG, for two reasons:
    a. Coding this example in RPG would take a much larger volume of code.
    b. The author would find it a bit irritating and time consuming to code this example in RPG.
5) This example uses TABLE_A and TABLE_B to produce a result set which is a mixture of summary (aggregate) and detail row data in the same row. Mixing aggregate and detail row columns in the same result set row typically involves complexity, whether SQL or RPG is used. The lateral joins used in the example greatly simplify the task of mixing summary and detail columns in the same row.
6) This example produces the following result set, with at most one row per combination of A.KEY_COLUMN, year(B.DATE_COLUMN):

```
 KEY_COLUMN
,SUMMARY_YEAR
,MIN_DATE_PER_YEAR
,MAX_DATE_PER_YEAR
,CLOSEST_MID_DATE_PER_YEAR
,YEAR_ROW_COUNT
,YEAR_TOTAL_DOLLARS
,YEAR_AVERAGE_DOLLARS
,MIN_DATE_DETAIL_DOLLARS
,MAX_DATE_DETAIL_DOLLARS
,CLOSEST_MID_DATE_DETAIL_DOLLARS
,MIN_DATE_DAY_NAME
,MAX_DATE_DAY_NAME
,CLOSEST_MID_DATE_DAY_NAME
```

```
--These "with" queries are common table expressions (CTE's) that can be referenced
--multiple times within a single query (reusable subqueries).
--This method of date calculation should work regardless of the date format in use.
with  --Tip: for full reusability, implement CTE_CURRENT_YEAR in an SQL view.
 CTE_CURRENT_YEAR ( CURRENT_YEAR_START_DATE, CURRENT_YEAR_END_DATE ) as (
   values ( CURRENT_DATE - day( CURRENT_DATE ) days + 1 day -
               ( month( CURRENT_DATE ) - 1 ) months
           ,CURRENT_DATE - day( CURRENT_DATE ) days + 1 day -
               ( month( CURRENT_DATE ) - 1 ) months + 1 year - 1 day
          )
 )  --Tip: for full reusability, implement CTE_LAST_YEAR in an SQL view.
,CTE_LAST_YEAR as (
   select  CURRENT_YEAR_START_DATE - 1 year  as LAST_YEAR_START_DATE
          ,CURRENT_YEAR_END_DATE   - 1 year  as LAST_YEAR_END_DATE
   from    CTE_CURRENT_YEAR  --Note: one CTE referencing a previous CTE
 )
select     A.KEY_COLUMN  --Columns delivered are mix of summary & detail data
          ,YEAR_SUMMARY.SUMMARY_YEAR
          ,YEAR_SUMMARY.MIN_DATE_PER_YEAR
          ,YEAR_SUMMARY.MAX_DATE_PER_YEAR
          ,CLOSEST_MID_DETAIL.CLOSEST_MID_DATE_PER_YEAR
          ,YEAR_SUMMARY.YEAR_ROW_COUNT
          ,YEAR_SUMMARY.YEAR_TOTAL_DOLLARS
          ,YEAR_SUMMARY.YEAR_AVERAGE_DOLLARS
          ,MIN_DETAIL.MIN_DATE_DETAIL_DOLLARS
          ,MAX_DETAIL.MAX_DATE_DETAIL_DOLLARS
          ,CLOSEST_MID_DETAIL.CLOSEST_MID_DATE_DETAIL_DOLLARS
          ,DAY_NAMES.MIN_DATE_DAY_NAME
          ,DAY_NAMES.MAX_DATE_DAY_NAME
          ,DAY_NAMES.CLOSEST_MID_DATE_DAY_NAME
from       TABLE_A  A
--Get SUMMARY by year data (deliver one matching TABLE_B row per DATE_COLUMN year)
cross join lateral (
  select   year(  B.DATE_COLUMN )                 as SUMMARY_YEAR
          ,min(   B.DATE_COLUMN )                 as MIN_DATE_PER_YEAR
          ,max(   B.DATE_COLUMN )                 as MAX_DATE_PER_YEAR
          ,count( B.DATE_COLUMN )                 as YEAR_ROW_COUNT
          ,sum( B.DOLLAR_COLUMN )                 as YEAR_TOTAL_DOLLARS
          ,dec( avg( B.DOLLAR_COLUMN ), 11, 2 )   as YEAR_AVERAGE_DOLLARS
  from     TABLE_B  B
  where    B.KEY_COLUMN = A.KEY_COLUMN
  group by year( B.DATE_COLUMN ) --Note: grouping using a calculated value
) as YEAR_SUMMARY
--Tack on DETAIL column from min date per year summary row
cross join lateral (
  select   B.DOLLAR_COLUMN as MIN_DATE_DETAIL_DOLLARS
  from     TABLE_B  B
  where    B.KEY_COLUMN  = A.KEY_COLUMN
       and B.DATE_COLUMN = YEAR_SUMMARY.MIN_DATE_PER_YEAR
) as MIN_DETAIL
--Tack on DETAIL column from max date per year summary row
cross join lateral (
  select   B.DOLLAR_COLUMN as MAX_DATE_DETAIL_DOLLARS
  from     TABLE_B  B
  where    B.KEY_COLUMN  = A.KEY_COLUMN
       and B.DATE_COLUMN = YEAR_SUMMARY.MAX_DATE_PER_YEAR
) as MAX_DETAIL
--Example continued on next page...
```

```
--Tack on DETAIL columns from row with date closest to the midpoint (half way)
--between the min and max dates within each year. I don't expect users to be likely to
--give you requirements to do this, but this is a good example of SQL's support for
--complexity. This will take significant effort for most readers to understand.
cross join lateral (
  select   B.DATE_COLUMN   as CLOSEST_MID_DATE_PER_YEAR
          ,B.DOLLAR_COLUMN as CLOSEST_MID_DATE_DETAIL_DOLLARS
  from     TABLE_B  B
  cross join lateral (
    values YEAR_SUMMARY.MIN_DATE_PER_YEAR +
           ( ( days( YEAR_SUMMARY.MAX_DATE_PER_YEAR ) -
               days( YEAR_SUMMARY.MIN_DATE_PER_YEAR )
             ) / 2
           ) days
  ) as MID_DATE ( CLOSEST_MID_DATE_PER_YEAR )
  where    B.KEY_COLUMN  = A.KEY_COLUMN
       and B.DATE_COLUMN
           between YEAR_SUMMARY.MIN_DATE_PER_YEAR
               and YEAR_SUMMARY.MAX_DATE_PER_YEAR
  order by abs( days( MID_DATE.CLOSEST_MID_DATE_PER_YEAR ) -
                days( B.DATE_COLUMN ) --abs of gap between dates (in days)
              )  --Note: order by using a calculated, absolute value (abs)
  fetch    first row only  --Only one row ensures we do not increase row count
) as CLOSEST_MID_DETAIL    --Fetch first row only (row with smallest gap in dates)
--Tack on day of week names for min, max, and closest mid dates per year.
--Note: 3 new columns are calculated and tacked onto the result set columns without
--      changing the number of rows in the result set.
--Cross join lateral to a values statement is a great way to tack columns onto a
--result set. The tacked on columns may now be referenced everywhere in the query
--result set except for table references above this point. This allows you to reduce
--repeating calculated expressions in multiple places in the query.
cross join lateral (
  values ( dayname( YEAR_SUMMARY.MIN_DATE_PER_YEAR )
          ,dayname( YEAR_SUMMARY.MAX_DATE_PER_YEAR )
          ,dayname( CLOSEST_MID_DETAIL.CLOSEST_MID_DATE_PER_YEAR )
         )
) as DAY_NAMES ( MIN_DATE_DAY_NAME
                ,MAX_DATE_DAY_NAME
                ,CLOSEST_MID_DATE_DAY_NAME )
--Only include A.KEY_COLUMN rows if activity is present in BOTH of the last 2 years.
where      exists(
             select     1
             from       TABLE_B  B
             cross join CTE_CURRENT_YEAR C --Note: reference to CTE within subquery
             where      B.KEY_COLUMN = A.KEY_COLUMN
                  and   B.DATE_COLUMN
                        between C.CURRENT_YEAR_START_DATE
                            and  C.CURRENT_YEAR_END_DATE
           )
       and exists(
             select     1
             from       TABLE_B  B  --6th and final reference to TABLE_B
             cross join CTE_LAST_YEAR C
             where      B.KEY_COLUMN = A.KEY_COLUMN
                  and   B.DATE_COLUMN
                        between C.LAST_YEAR_START_DATE
                            and C.LAST_YEAR_END_DATE
           )
--Example continued on next page...
```

```
--Sort rows: 1) By SUMMARY_YEAR descending first,
          -- 2) Within a year:
          --    Rows with largest gap between min and max dates first,
          --      followed by rows with smaller gap between min and max dates.
          --Note: ordering using calculated values.

order by   SUMMARY_YEAR desc
          ,( days( YEAR_SUMMARY.MAX_DATE_PER_YEAR ) -
             days( YEAR_SUMMARY.MIN_DATE_PER_YEAR ) ) desc
```

Notes:

- This example is performing a lot of work using ONE request to the database, which is a dramatic reduction in I/O requests compared to achieving the same result in RPG one row at a time.
- This example is fairly complex for those readers who have not yet reached an advanced level in both SQL and result set oriented design and coding. Many readers at an introductory skill level in SQL, who are also new to result set design and coding, would likely say "no way would I ever code something like that in SQL". If faced with similar complex requirements, they would then proceed to code this in RPG. However, the resulting code in RPG would be far more complex than the above SQL, a much greater volume of code, more difficult to maintain, and much more likely to contain defects. Those same readers would argue that the above SQL is more complex. I disagree. I think the difference in opinion lies in a difference in skill level, or more accurately, a lack of SQL skill. Complexity is a perception of our skill level. Over the course of decades, I've coded a lot more RPG than SQL, can easily code this example in either language, and find the SQL to be much simpler.
- If your SQL skill doesn't permit it, it is certainly fine to code something complex like this in RPG. We all have deadlines to meet, and tend to code to our current abilities. However, in the long term, SQL, especially set based SQL, can simplify your coding, and improve your productivity, dramatically. Once you're able to easily code an example like this in SQL, I guarantee you won't go back to coding something like this in RPG (unless someone forces you to). If someone forces you to code a complex example like this in RPG, my suggestion is to send that person a copy of this document.
- The more complex the result set, the greater the chance you may have to performance tune the resulting code. This may involve a combination of adding or modifying indexes, and/or restructuring the code, and/or breaking the SQL into multiple statements, using one or more work tables or in memory result sets to communicate between statements. If adding and modifying indexes and restructuring the code doesn't yield the desired performance, don't hesitate to break the single SQL statement into two or more statements, using one or more work tables, or in memory result sets, or other methods, to pass the results between statements. There are times when breaking a single, complex SQL statement into multiple statements decreases the complexity enough for the SQL engine to make a better access plan, yielding higher performance than possible using a single statement. Most of the time that is not necessary, but occasionally it is for complex SQL.
- The more complex the SQL, the greater the chance you'll encounter defects in the SQL engine. I've found some defects in the SQL engine (valid syntax that halts at runtime) using extremely complex SQL. In order to meet your milestones, when working with extremely complex SQL, be prepared to occasionally have to work around defects in order to avoid delays in your projects. The DB2® SQL engine does an amazing job at processing extremely complex SQL, but it isn't perfect. For me, waiting for a fix from IBM® isn't an option. I've been 100% successful at coding the same complexity in a different way to work around a defect. If you encounter an SQL0901 error, you've probably hit a defect in the SQL engine. You can wait for a database patch to be applied, or code around it (my recommendation).

Tips for testing SQL

- Testing SQL is easier than testing RPG code, because most SQL can be initially tested without having to compile a program, and SQL allows you to easily query your test results. In fact, I highly recommend testing your SQL outside of a program first. Once your SQL proves its accuracy outside of a compiled program, then place it inside the code that will become your compiled program (or the database table intended for storing and running SQL from the database, if you're using that design).
- I recommend initially testing your SQL inside a GUI SQL environment like the "Run SQL Scripts" feature of IBM i® Navigator, or using any one of the many other GUI SQL runtime environments available such as Linoma™ Surveyor/400™ or IBM® Data Studio. Any GUI SQL runtime environment that supports ODBC or OLE-DB will likely work well against the IBM i® database, or other database if you're developing for a different platform.
- You can test your SQL inside the IBM i® STRSQL program, but the limited screen size and limited functionality of the green screen user interface is not as efficient as a GUI SQL interface. With a GUI, you can see more code and data results at one time, easily copy/paste results, more easily work with large SQL statements, and from some GUI SQL environments like IBM i® Navigator's Run SQL Scripts feature, you get runtime feedback, detailed error messages, syntax checking, and excellent performance tuning advice from the Visual Explain feature.
- During initial testing of SQL against high row count tables, I recommend adding a temporary WHERE clause, or other filter code to temporarily limit your test data results to a small subset of the data. Once the subset test data results look accurate, remove the temporary filter logic, and then test to ensure the SQL accurately produces the full set of results. SQL can process very large sets of data at a time, at times requiring significant runtime, and initially testing a smaller subset can save time by delivering a small set of the results quickly for review. Early testing is where you identify most of your bugs, and you don't want to wait long to find them. Testing in that fashion is much more convenient in SQL versus RPG.
- SQL is best utilized when processing full result sets of data. In some cases, those result sets may be very large (millions, billions, or more rows). It is not practical to manually review every result set row when working with very large row counts. Early testing of SQL result sets containing very large row counts should be captured into a table for review, as follows:

```
create table MY_SCHEMA.MY_RESULTS as (
   select * from MY_TABLE
) with data
```

  - Replace "`select * from MY_TABLE`" with your SQL query to be tested. Replace MY_SCHEMA with the name of a test of development schema to receive your test results. Replace MY_RESULTS with your desired name of the table to be created containing your test results.
  - Run the create table statement to capture your results.
  - Run many SQL queries against the resulting table created to validate the results.
  - If the results are incorrect, correct your SQL query, run "`drop table MY_SCHEMA.MY_RESULTS`", and repeat this process until the results are correct.
  - Once correct, copy/paste the SQL query being tested into the final destination program (or database if you're running SQL from a database table).

Conclusions / observations

- LATERAL joins are used in many of the SQL examples. They greatly simplify the task of picking any number of rows when there are many to choose from. Unfortunately, many IBM i® developers using SQL have not noticed them, or realized their usefulness, and they are underutilized. I highly recommended and use them frequently. If you learn only one thing from this document, preferably it is the usefulness of LATERAL joins.
    - Given many of the examples use lateral joins, and the variations in the SQL code are minor across those examples, this demonstrates the flexibility of lateral joins.
    - In addition, I've not noticed any significant performance issues with using them. On DB2® for IBM i®, LATERAL joins appear to be optimized well.
    - Any SQL is subject to performance issues if indexes are missing, or the SQL is poorly structured to take advantage of existing indexes, or complexity of the SQL overwhelms the database engine's ability to calculate an efficient access plan. DB2® for IBM i® does an amazing job of calculating high performance access plans, and it gets better with each new release, but on rare occasions it makes less than optimum choices, even when given strong hints in the SQL code on how to efficiently achieve the desired results.
    - There are many other ways to structure SQL to accomplish the same results as a LATERAL join, but those alternatives typically result in a larger code volume, decreased clarity, and increased runtime.
    - I find myself frequently using LATERAL joins in cases where I need to pick one row out of many and the "many" table contains rows with start and end date columns. For example, the "many" table contains many date ranges, and the SQL needs to pick only one row, usually the row with the latest date range.
    - A common defect in SQL queries, is picking more than one row out of many (typically during a JOIN), when only one row is the desired result. LATERAL joins make it simple to prioritize and pick that single row, and to ensure that only one row is chosen.
    - To reduce defects, every time you perform a join, ask yourself two questions, and structure the code according to the answers:
        - How many rows do I want the join to return?
        - How many rows are possible for the join to return?
            - The presence of unique indexes may answer this question for you. If a unique index is not present, run queries over the data to determine how many rows are possible for the join to return. Queries of the data may show only one row is returnable today, but if a unique index is not present to enforce that result, in the future, multiple rows could be returned. Structure your code accordingly.
- If you expand these examples to include access to more tables, the SQL code becomes dramatically more readable versus the RPG (i.e. the more tables added, the greater the difference between the readability and volume of the SQL code versus the RPG).
- On DB2® for IBM i®, the maximum row length of an SQL result set is limited to 32,766 total bytes, without LOB (large objects) included. Exceeding a row length of 32,766 requires use of LOB objects. Except for that limit, SQL can produce ANY result set of rows and columns that an RPG program would build in memory. SQL contains all the foundational building blocks necessary to build virtually ANY result set. In nearly all cases, compared to RPG, SQL can build the same result set faster, cheaper, with a reduced code volume, greater clarity, and a reduced chance for defects. For reasons of improving performance, reducing complexity, or structuring code for reusability, occasionally a result set may need to be split into two or more queries producing two or more result sets.
- I highly recommend using SQL, especially set based SQL, to perform ALL your relational database input / output, since it will dramatically increase your productivity, and its benefits are numerous.
- SQL has a much larger labor pool to choose from than RPG, and is a skill in much greater demand. At the time of this writing, SQL is one of the most widely used and desired skills in the software development industry.
- Replacing RPG I/O with SQL, and/or building very complex result sets in SQL, is primarily limited by your skill, and is rarely limited by SQL or the database.
- When your SQL skill level is on par with your RPG skill level, you'll find SQL easier and faster to develop, maintain, and read, with improved performance, especially if using a set based design.

About the author

I've been developing software professionally since 1983, have a fanatical passion for set based SQL, and for pushing it to its limits. I've been an avid SQL user since the early 1990's. Since 2008, I've been performing very large, very complex, database migrations, transformations, subsets, and extractions on DB2®, using SQL as a complete runtime environment, and using solely set based SQL. I've developed a database migration and transformation architecture, exclusively in SQL, that runs almost entirely server side, where the database engine essentially communicates with itself, with very little need to communicate with programs outside the database. It stores SET based SQL scripts inside the database, uses custom SQL user defined functions to execute those SQL statements, and it invokes the script execution functions in the middle of query result set generation. The net result is extremely high speed, server side data migrations and transformations, using only set based SQL, initiated from any SQL environment command line or tool. Accomplishing this required building a robust set of custom SQL functions, which were also written in, you guessed it, SQL.

Some of the more advanced things I've done with SQL are:

- Consistently achieve data migration / transformation rates from 10,000 to 350,000+ rows per second on modestly sized hardware.
- For test teams, subset ~2TB databases with ~1700 tables down to ~20% of original size, maintaining referential integrity, using ~86% generated SQL, with a runtime of ~7 hours, on modestly sized hardware.
- Using SQL to generate SQL to update the database (inserts, updates, deletes, create tables, create indexes, drop tables, etc.).
- Using SQL to generate SQL views to transparently transform data (i.e. using generated SQL views to make data appear as if it has been transformed, without altering the source data). This is very useful for data masking of privacy data, and converting code and type columns to new values. It allows for very high speed data migration and transformation at the same time, using generated SQL instead of manually crafted SQL.
- Created a robust set of SQL user defined table and scalar functions, written in SQL, for a variety of parsing, data transformation, leveraging metadata, SQL generation, and other utility needs.
- Created SQL user defined table and scalar functions as wrappers around IBM i® OS API's. This allows the execution of operating system API's from SQL queries. This provides a much more flexible interface to accessing operating system API's versus creating the same wrappers in RPG, CL, or C.
- Created custom SQL user defined functions to process SQL scripts. Envision an enhanced RUNSQLSTM ability that can be invoked in the middle of SQL queries and statements by calling user defined table and scalar SQL functions – i.e. execute SQL scripts in the middle of execution of SQL queries or other SQL statements.
- Created user defined functions to execute high level language (HLL) programs and commands from within SQL statements and queries (allows embedding CL, RPG, C, and other HLL programs inside SQL).
- Storing all database migration / transformation SQL code in the database itself, which provides these advantages:
    - The ability to perform SQL queries against SQL code. This makes it easy to analyze and cross reference an SQL code base using SQL queries.
    - The ability to update SQL code, potentially mass updates, using SQL UPDATE statements.
    - The ability for the database to fetch and execute SQL without communicating with external programs or systems. Removing the need to communicate with external programs improves performance significantly. Housing SQL code in the database itself simplifies the task of maintaining a server side execution architecture, which greatly improves performance, and helps simplify complex database tasks.

There are numerous benefits when you implement functionality in set based SQL versus using RPG or CL. For one, doing so allows you to maximize server side processing, and allows you to maximize processing performed within the database engine itself. There is a lot of overhead when a program outside the database communicates with the database engine. When you move functionality from RPG and CL into SQL, in an architecture designed to support it, you reduce overhead by allowing the database engine to communicate with itself, instead of external programs. Properly implemented, the result is very high performance.

Feedback

If you have comments regarding this document, please send them to Mike.Jones.SysDev@gmail.com. All comments are welcome (praise, criticism, errors, omissions, missing coding patterns, SQL training wish list, etc.). If you've benefited from this document, feel free to connect with me on LinkedIn®.

Legal disclaimers

The author is providing this document "as is", free of charge, without warranty of any kind (express, implied, statutory, fitness for a particular purpose, or warranty that it is error-free). In no event shall the author incur any liability for damages, including direct, indirect, special, or consequential damages arising out of the use of this document.

Trademarks

DB2, IBM i, i5/OS, OS/400 are trademarks or registered trademarks of IBM Corporation.

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

Linoma and Surveyor/400 are trademarks of Linoma Software.

LinkedIn is a registered trademark or trademark of LinkedIn Corporation and its affiliates in the United States and/or other countries.

Copyright

© Copyright 2015 Michael R. Jones. All rights reserved. Distribution permission of this document is granted only if this copyright notice and contact information is included in the distribution, and the document is distributed in its entirety and free-of-charge.

Index