TECHNICAL REPORT

ISO/IEC TR 19075-7

First edition
2017-03

Information technology — Database languages — SQL Technical Reports —

Part 7: Polymorphic table functions in SQL

Technologies de l'information — Langages de base de données — SQL rapport techniques —

Partie 7: Fonctions de table polymorphes dans SQL

Reference number
ISO/IEC TR 19075-7:2017(E)

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

Contents

Page

Tables

Table

Page

Figures

Figure

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/IEC JTC 1, *Information technology,* Subcommittee SC 32 *Data management and interchange.*

A list of all parts in the ISO 19075 series can be found on the ISO website.

NOTE 1 — The individual parts of multi-part technical reports are not necessarily published together. New editions of one or more parts may be published without publication of new editions of other parts.

Introduction

The organization of this part of ISO/IEC 19075 is as follows:

1) Clause 1, “Scope”, specifies the scope of this part of ISO/IEC 19075.

2) Clause 2, “Normative references”, identifies additional standards that, through reference in this part of ISO/IEC 19075, constitute provisions of this part of ISO/IEC 19075.

3) Clause 3, “Introduction to Polymorphic Table Functions”, provides an introduction to polymorphic table functions, the requirements leading to their incorporation into SQL, and illustrations of their use.

4) Clause 4, “PTF processing model”, describes the abstract processing model for polymorphic table functions in the context of an SQL-implementation.

5) Clause 5, “Specification”, describes the manner in which polymorphic table functions are specified in the SQL standard.

6) Clause 6, “Data definition language”, provides the syntax and semantics of the SQL statements that create, modify, and drop polymorphic table functions.

7) Clause 7, “Implementation”, guides authors of polymorphic table functions through the steps required to create all of the functions necessary to accomplish particular purposes.

8) Clause 8, “Invocation”, supplies the information necessary for application writers, especially SQL query authors, to take advantage of the polymorphic table functions that are available to them.

9) Clause 9, “Compilation”, is directed at the authors of polymorphic table functions and of SQL database systems to guide them in the steps required to compile polymorphic table functions in the context of a particular SQL-implementation.

10) Clause 10, “Optimization”, describes the various aspects of polymorphic functions of which the authors of such functions and the authors of SQL-implementations must be aware to adequately optimize the execution of such functions.

11) Clause 11, “Execution”, discusses the details of executing polymorphic table functions in the context of the processing model.

12) Clause 12, “Examples”, supplies numerous examples in detail with commentaries to explain the various use cases, the requirements that relate to polymorphic table functions, and the specifics of the solutions for each use case.

Information technology — Database languages — SQL Technical Reports —

Part 7:
Polymorphic table functions in SQL

1 Scope

This Technical Report describes the definition and use of polymorphic table functions in SQL.

The Report discusses the following features of the SQL Language:

— The processing model of polymorphic table functions in the context of SQL.

— The creation and maintenance of polymorphic table functions.

— Issues related to methods of implementing polymorphic table functions.

— How polymorphic table functions are invoked by application programs.

— Issues concerning compilation, optimization, and execution of polymorphic table functions.

(Blank page)

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

2.1 ISO and IEC standards

[ISO9075-2] ISO/IEC 9075-2:2016, *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*.

(Blank page)

3 Introduction to Polymorphic Table Functions

A polymorphic table function (abbreviated PTF) is a function that returns a table whose row type is not declared when the function is created. Rather, the row type of the result may depend on the function arguments in the invocation of a PTF, and therefore may vary depending on the precise syntax containing the PTF invocation. In addition, a PTF may have generic table parameters (*i.e.*, no row type declared when the PTF is created), and the row type of the result might depend on the row type(s) of the input tables. This Technical Report is intended to provide an informal description of PTFs, using examples and practical step-by-step advice on how to add a PTF capability to a relational DBMS, how to write a PTF, and how to invoke a PTF in an application.

3.1 Audiences

This Technical Report is written for three audiences:

1) The DBMS developer.

2) The PTF author.

3) The query author.

It is important to recognize that a PTF is somewhat like a view, only more procedural. With a view, there are the same three parties: DBMS, view author, and query author. The DBMS is the intermediary between the view author and the query author. The view is a way for the view author to "publish" an interface to tables without exposing the inner workings of the interface. Similarly, a PTF is a way for the PTF author to publish an interface to a procedural mechanism that defines a table. The query author only sees the published interface, whereas the DBMS and the PTF author share a more complex "private" interface. In particular, the query author sees a single PTF function, whereas the DBMS and the PTF author see a family of one to four related SQL-invoked procedures, called the PTF component procedures, and possibly additional private data.

Many sections of the Technical Report begin with a caption "Primary audience: xyz" stating which audience is most likely to find the section useful. However, anyone is welcome to read any section. Understanding the complete picture from all three perspectives is likely to be helpful to all three audiences.

A fourth audience can also be distinguished, the database administrator (DBA) as the one who is responsible for allocating disk storage for large tables. This is because disk allocation strategies can impact the performance of queries. The one allocating disk storage may be the query author or may be a separate role in the organization. If these are separate roles, the DBA may need to consult with the query author before the data is loaded. This is a complex issue because it may not be possible to anticipate the query at the time that the data is loaded, or the set of queries may be such that no one disk allocation scheme will allow optimal performance of all queries. The most relevant section for a DBA allocating disk storage is Subclause 11.1, "Partitions and virtual processors", where the audience is nominally the DBMS and the PTF author. However, specific advice about disk allocation is beyond the scope of this Technical Report.

3.2 Motivating examples

We begin by showing eight motivating examples that illustrate the capabilities of PTFs. These examples are presented from the standpoint of the query author, hiding the role of the PTF author and DBMS. The objective here is to get a taste of the power and generality of PTFs. The perspectives of the DBMS and the PTF author are explored at length in Clause 12, “Examples”.

3.2.1 CSVreader

A spreadsheet can usually output a comma-separated list of values. Generally, the first line of the file contains a list of column names, and subsequent lines of the file contain data. The data in general can be treated as a large VARCHAR. However, some of the fields may be numeric or datetime.

The PTF author has provided a PTF called CSVreader designed to read a file of comma-separated values and interpret this file as a table. The query author can see this PTF in the Information Schema and knows that it has the following signature:

```
FUNCTION CSVreader (
    File VARCHAR(1000),
    Floats DESCRIPTOR DEFAULT NULL,
    Dates DESCRIPTOR DEFAULT NULL )
  RETURNS TABLE
  NOT DETERMINISTIC
  CONTAINS SQL
```

This signature has two parameter types that are distinctive to PTFs:

1) DESCRIPTOR is a type that is capable of describing a list of column names, and optionally for each column name, a data type. There is a helper function provided for the query author to construct a PTF descriptor area.

2) TABLE denotes the generic table type, a type whose value is a table. The row type of the table is not specified, and may vary depending on the invocation of the PTF.

In this example, the return type of CSVreader is TABLE. This is a distinguishing characteristic of every polymorphic table function: it returns a generic table.

The PTF author has published a user reference for CSVreader. The user reference tells the query author the semantics of the input parameters and what the output will be. In this example, the user reference documents the following:

1) The first parameter, File, is the name of a file on the query author's system. This file must contain the comma-separated values that are to be converted to a table. The first line of the file contains the names of the resulting columns. Succeeding lines contain the data. Each line after the first will result in one row of output, with column names as determined by the first line of the input.

2) Floats is a PTF descriptor area, which should provide a list of the column names that are to be interpreted numerically. These columns will be output with the data type FLOAT.

3) Dates is a PTF descriptor area, which provides a list of the column names that are to be interpreted as datetimes. These columns will be output with the data type DATE.

Based on the documentation in the user reference, the query author may write a query such as the following:


```
SELECT *
FROM TABLE ( CSVreader ( File => 'abc.csv',
                         Floats => DESCRIPTOR ("principle", "interest")
                         Dates => DESCRIPTOR ("due_date")
                        ) ) AS S
```

In the FROM clause, the TABLE operator introduces the invocation of a table function. A table function might be either a conventional (monomorphic) table function or a PTF. In this case, because CSVreader is declared with return type TABLE, this is a PTF invocation.

This invocation says that CSVreader should open the file called abc.csv. The list of output column names is found in the first line of the file. Among these column names, there must be columns named 'principle' and 'interest', which should be interpreted as numeric values, and a column named 'due_date', which should be interpreted as a date.

For example, suppose that the contents of abc.csv are

```
docno,name,due_date,principle,interest
123,Mary,01/01/2014,234.56,345.67
234,Edgar,01/01/2014,654.32,543.21
```

The result will be

docno	name	due_date	principle	interest
123	Mary	2014-01-01	234.56	345.67
234	Edgar	2014-01-01	654.32	543.21

The distinguishing feature of this example is that there are no input tables. Subsequent examples show various possibilities involving input tables.

This example is continued in detail in Subclause 12.2, “CSVreader”.

3.2.2 Pivot

In general, a pivot is an operation that reads a row and outputs several rows. Generally, the input is denormalized and the output is normalized. For example, maybe an input table has six columns, forming three pairs of (phone type, phone number), and the user wishes to normalize this into a table with two columns.

The PTF author has provided a PTF called Pivot; the query author can see the following signature in the Information Schema:

```
FUNCTION Pivot (
    Input TABLE PASS THROUGH WITH ROW SEMANTICS,
    Output_pivot_columns DESCRIPTOR,
    Input_pivot_columns1 DESCRIPTOR,
    Input_pivot_columns2 DESCRIPTOR DEFAULT NULL,
    Input_pivot_columns3 DESCRIPTOR DEFAULT NULL,
    Input_pivot_columns4 DESCRIPTOR DEFAULT NULL,
    Input_pivot_columns5 DESCRIPTOR DEFAULT NULL
```

```
  ) RETURNS TABLE
  DETERMINISTIC
  READS SQL DATA
```

The PTF author has provided a user reference that documents the following semantics for this PTF:

1) The first parameter, Input, is a generic table. This table is declared to have two options, “PASS THROUGH” and “WITH ROW SEMANTICS”. These options have the following implications for the query author:

   a) WITH ROW SEMANTICS means that the result is determined on a row-by-row basis. The alternative, set semantics, will be seen in subsequent examples. At most one input table can have row semantics.

   b) PASS THROUGH means that, for each input row, the PTF makes the entire input row available in the output, qualified by a range variable associated with the input table. The alternative, NO PASS THROUGH, will be seen in some subsequent examples.

2) The second parameter, Output_pivot_columns, is a PTF descriptor area that lists the names of the columns that the query author wants to see in the result.

3) The third parameter, Input_pivot_columns1, is mandatory. This parameter is a PTF descriptor area that lists the names of the columns of the input table which are to be pivoted into the corresponding columns of the output table. There must be the same number of column names in the Output_pivot_columns PTF descriptor area and in the Input_pivot_columns1 descriptor area.

4) The remaining parameters, Input_pivot_columns2, Input_pivot_columns3, Input_pivot_columns4, and Input_pivot_columns5, are optional (indicated by the DEFAULT NULL declaration). If supplied, these are additional PTF descriptor areas for the input columns that are to be pivoted into the output columns. Each of these PTF descriptor areas must have the same number of column names as Output_pivot_columns.

This shows the capability to pivot at most 5 sets of columns. Of course, the PTF author could support many more by simply adding more optional parameters to the signature.

Based on this user documentation, the query author might write the following invocation:

```
SELECT D.Id, D.Name, P.Phonetype, P. Phonenumber
FROM TABLE ( Pivot ( Input => TABLE (Joe.Data) AS D,
                     Output_pivot_columns => DESCRIPTOR (phonetype, phonenumber),
                     Input_pivot_columns1 => DESCRIPTOR (phtype1, phonenumber1),
                     Input_pivot_columns2 => DESCRIPTOR (phtype2, phonenumber2)
                   ) ) AS P
```

In this invocation, the first TABLE () operator encloses the PTF invocation. The first parameter, called Input, passes a table, and uses the TABLE () operator to enclose the table name. This second TABLE () operator is required because SQL would normally interpret syntax such as Joe.Data as a column name rather than a table name. Since Joe.Data is a table, it is possible to assign it a correlation name, D in this example. (If an explicit correlation name is not provided, then the table name Joe.Data, or just Data, may be used as a range variable to reference it.) The remaining arguments are PTF descriptor areas, first of the output pivot columns and then the corresponding pairs of input pivot columns.

The query has two correlation names, D and P. D is associated with the input table Joe.Data whereas P is associated with the output of the PTF.

For input tables with pass-through columns, as in this example, the correlation name of the input table may be used as a qualifier to reference any column of the associated input table. (Input tables with set semantics follow a slightly different rule to be presented later.) In this example, D has been used to qualify the columns D.Id and D.Name.

P may be used to reference the columns that are produced by the PTF. In this example, P has been used to qualify the columns P.Phonetype and P.Phonenumber.

The result of the PTF invocation is a multiset of rows, each row having some columns qualified by D and some columns qualified by P. Every column of the input table Joe.Data is accessible in the columns qualified by D; the output columns of the PTF are referenceable using the correlation name P. In effect, each input row is concatenated with the columns that are produced by the PTF. This is a consequence of the fact that the input table has pass-through columns and row semantics: the result of the PTF is determined on a row-by-row basis (row semantics), and the entire input row is concatenated with the result of the PTF (pass-through columns). (The PTF may produce more than one row for a given input row; this will cause a "multiplier effect" in the output.)

For example, suppose Joe.Data has the following data:

ID	NAME	PHTYPE1	PHNUMBER1	PHTYPE2	PHNUMBER2
123	Mary	home	pqr	cell	stu
234	Edgar	home	vwx	work	xyz

The result will be

ID	NAME	PHONETYPE	PHONENUMBER
123	Mary	home	pqr
123	Mary	cell	stu
234	Edgar	home	vwx
234	Edgar	work	xyz

This example is continued in detail in Subclause 12.3, "Pivot".

3.2.3 Score

Score has two input tables:

1) One input table contains rows to be scored according to some algorithm.

2) The other input table (the model) contains the parameters for the algorithm that is used to score a row.

Each row can be scored independently of every other row. In contrast, every row of the model is required to specify the scoring algorithm.

The PTF author has provided a PTF called Score; the query author can see the following signature in the Information Schema:

```
FUNCTION Score (
    Data TABLE PASS THROUGH WITH ROW SEMANTICS,
```

```
    Model TABLE NO PASS THROUGH
      WITH SET SEMANTICS PRUNE WHEN EMPTY
) RETURNS TABLE (Score REAL)
DETERMINISTIC
READS SQL DATA
```

The first input table, called Data, contains the rows to be scored. Each row is scored independently of every other row, as indicated by WITH ROW SEMANTICS. The entire input row is accessible in the output, as indicated by PASS THROUGH.

The second input table, called Model, contains the parameters used for scoring a row. Since the entire data set is required to specify the algorithm, this table is declared as WITH SET SEMANTICS. A table with set semantics may be partitioned and/or ordered. Partitioning and ordering are decisions made by the query author, expressed in query syntax, as we shall see.

NO PASS THROUGH indicates that columns of the Model table are not copied to the output. However, if the input is partitioned, then the partitioning column(s) are still available in the output.

Since the algorithm cannot work with an empty model, the qualifier PRUNE WHEN EMPTY is added, indicating that the result of the PTF is empty if the model table is empty. This enables the DBMS to optimize by not even invoking the PTF when this table is empty.

The result, for each input row of Data, is a row concatenated from the following three sources:

1) The entire row of Data (because this has row semantics with pass-through columns).

2) The partitioning columns of Model, if any (because this has set semantics without pass-through columns).

3) An additional column named SCORE of type REAL containing the score for that row of Data.

The query author has a table containing a number of different models, which can be used to score rows for comparison against different “what if” scenarios. The query author writes the following query:

```
SELECT D.Id, D.S, D.T, M.Modelid, T.Score
FROM TABLE (Score ( Data =>  TABLE (MyData) AS D
                    Model => TABLE (Models) AS M
                               PARTITION BY Modelid
) AS T
```

This example has three correlation names, corresponding to the three sources for columns in the output rows:

1) D is the correlation name for the input table MyData. MyData has row semantics with pass-through columns and its correlation name D may be used to qualify any column of MyData.

2) M is the correlation name for the input table Models. Models has set semantics. It does not have pass-through columns but its correlation name M can be used to qualify the partitioning column Modelid.

3) T is the correlation name for the result of the PTF. T is used to qualify the additional column named SCORE.

This example introduces the PARTITION BY clause. Only tables with set semantics may be partitioned. The input table is partitioned as specified by the column(s) in the PARTITION BY clause; the PTF is evaluated independently on each partition. The SQL standard uses an abstraction called a virtual processor to specify the evaluation of a PTF. In this example, each partition is assigned to a separate virtual processor.

For example, perhaps Models contains the following rows:

Modelid	pname	pvalue
wet	x	19
wet	y	28
wet	z	37
dry	x	4
dry	y	5
dry	z	6

This tables contains two models, “wet” and “dry”, each having three parameters named “x”, “y”, and “z”, with parameter values in the column pvalue.

Table MyData may contain information to be scored by these two models:

id	s	t
122	9.4	3.4
233	8.4	6.5
344	10.2	9.3
455	11.0	8.8

The result might look like this:

id	s	t	Modelid	score
122	9.5	3.4	wet	12.9
233	8.4	6.5	wet	14.9
344	10.2	9.3	wet	19.5
455	11.0	8.8	wet	19.8
122	9.5	3.4	dry	6.4
233	8.4	6.5	dry	7.4
344	10.2	9.3	dry	9.2
455	11.0	8.8	dry	9.4

In the result, the first three columns are copied from MyData. The next column, M.Modelid, comes from the partitioning of Models. Every row of MyData is analyzed by both models, “wet” and “dry”. Since there are four rows in MyData and two models, there are eight rows in the result, four for each model. The last column is the score produced by the PTF.

This example is continued in detail in Subclause 12.4, “Score”.

3.2.4 TopNplus

TopNplus takes an input table that has been sorted on a numeric column. It copies the first *n* rows through to the output table. Any additional rows are summarized in a single output row in which the sort column has been summed and all other columns are null.

The query author sees the following signature in the Information Schema:

```
FUNCTION TopNplus (
    Input TABLE NO PASS THROUGH
          WITH SET SEMANTICS PRUNE WHEN EMPTY,
    Howmany INTEGER
  ) RETURNS TABLE
  NOT DETERMINISTIC
  READS SQL DATA
```

The PTF author has provided the following user documentation:

1) The first parameter, Input, is the input table. This table has set semantics, meaning that the result depends on the set of data (since the last row is a summary row). In addition, the table is marked as PRUNE WHEN EMPTY, meaning that the result is necessarily empty if the input is empty. The query author must order this input table on a single numeric column (syntax below).

2) The second parameter, Howmany, specifies how many input rows that the user wants to be copied into the output table; all rows after this will contribute to the final summary row in the output.

Using the user documentation, the query author might write the following query:

```
SELECT S.Region, T.*
FROM TABLE ( TopNplus ( Input => TABLE (My.Sales) AS S
                                 PARTITION BY Region
                                 ORDER BY Sales DESC,
                        Howmany => 3
                      )
           ) AS T
```

This example shows an input table that is both partitioned and ordered. In general, an input table with set semantics may be partitioned or ordered or both.

Consider the following input data representing the content of the table My.Sales:

Region	Product	Sales
East	A	1234.56
East	B	987.65

Region	Product	Sales
East	C	876.54
East	D	765.43
East	E	654.32
West	E	2345.67
West	D	2001.33
West	C	1357.99
West	B	975.35
West	A	864.22

The first five rows make up the partition with Region = 'East' and the last five rows make up the partition with Region = 'West'. Also notice that each partition has been sorted in descending order on Sales.

The DBMS creates two virtual processors, one for each partition. For example, on the virtual processor for Region = 'East', TopNplus sees the following input as S:

Region	Product	Sales
East	A	1234.56
East	B	987.65
East	C	876.54
East	D	765.43
East	E	654.32

In the other partition, for Region = 'West', TopNplus sees the following input as S:

Region	Product	Sales
West	E	2345.67
West	D	2001.33
West	C	1357.99
West	B	975.35
West	A	864.22

On each virtual processor, TopNplus copies the first 3 rows to the output (because Howmany = 3). However, it does not copy the partitioning column, since that is available to the query using the correlation name S. Then, TopNplus reads the remaining rows and computes the sum. In partition Region = 'East', the sum is 1419.75; in the other partition, the sum is 1839.57.

The result of the PTF invocation is shown below:

S	T	
Region	**Product**	**Sales**
East	A	1234.56
East	B	987.65
East	C	876.54
East		1419.75
West	E	2345.67
West	D	2001.33
West	C	1357.99
West		1839.57

Note that the result uses two correlation names: S, to qualify the partitioning column, and T, to qualify the columns that were output by TopNplus.

This example has been designed to show how the PTF can copy rows of input to the output without using Feature B205, “Pass-through columns”, which is an optional feature and may not be available in every implementation of polymorphic table functions. The example can also be modified a little to exploit pass-through columns if Feature B205, “Pass-through columns” is available. The modification is necessary because pass-through columns are an “all or nothing” capability — either an entire input row is copied to the output, or a row of nulls (except for the partitioning columns). In the results above, the first three rows in each partition can be copied to the output, where they would be qualified by S rather than T. The summary row, on the other hand, is not copied from any input row; therefore, the summary row would be null in the S.Product and S.Sales columns. To report the summary statistic, the PTF would use a separate column, qualified by T. Thus the result might look like this:

S			T
Region	**Product**	**Sales**	**Sales**
East	A	1234.56	1234.56
East	B	987.65	987.65
East	C	876.54	876.54

S			T
Region	**Product**	**Sales**	**Sales**
East			1419.75
West	E	2345.67	2345.67
West	D	2001.33	2001.33
West	C	1357.99	1357.99
West			1839.57

This example is continued in detail in Subclause 12.5, “TopNplus”.

3.2.5 ExecR

R is a programming language used for analytic calculations. ExecR executes an R script on an input table. The PTF receives the R script as an input character string, but lacks the sophistication to analyze this R script to determine the row type of the result. Consquently, the query author bears the burden of specifying the output row type.

The query author sees the following signature in the Information Schema:

```
FUNCTION ExecR (
    Script VARCHAR(10000),
    Input TABLE NO PASS THROUGH
          WITH SET SEMANTICS KEEP WHEN EMPTY,
    Rowtype DESCRIPTOR )
  RETURNS TABLE
  NOT DETERMINISTIC
  READS SQL DATA
```

The PTF author supplies the following user documentation:

1) The first argument, Script, is a character string containing the R script to be executed.

2) The second argument, Input, is a generic input table with set semantics. This input table will be passed to the R script using an interface defined by the PTF author. The R script is expected to process the input table and produce an output table. Since it is possible that the R script might produce an output even if there is empty input, this generic table is marked KEEP WHEN EMPTY. Since the R script will not have the ability to copy input rows into output rows, the input table is NO PASS THROUGH.

3) The third argument, Rowtype, is a PTF descriptor area of the row type that the R script will produce.

Based on the PTF documentation, the query author might write this query:

```
SELECT D.Region, R.Name, R.Value
FROM TABLE (ExecR ( Script => '...',
                    Input => TABLE (My.Data) AS D
                               PARTITION BY Region,
```

```
                Rowtype =>
                  DESCRIPTOR (Name VARCHAR(100), Value REAL)
              )
        ) AS R
```

In this invocation, the things to note are:

1) The first argument is the R script to be evaluated.

2) The second argument is an input table with set semantics, partitioned by Region. In this example, the input is not sorted.

3) The third argument, Rowtype, provides a PTF descriptor area of two columns, called Name and Value, with types VARCHAR(100) and REAL, respectively.

The result rows are concatenated from two sources:

1) The partitioning column D.Region, qualified by the correlation name for the partitioned input table.

2) The columns R.Name and R.Value that are output by the PTF, qualified by R, the correlation name associated with the PTF.

This result is supported as follows:

1) The input data is partitioned on Region.

2) Each region effectively constitutes an independent data set.

3) The DBMS creates a virtual processor for each partition.

4) The virtual processor reads the data in a partition and produces a row with two columns, Name and Value.

5) The Region code is constant for all input rows on a virtual processor, and so the DBMS is able to augment the result from the PTF with the partition column.

6) In the SELECT list, the partitioning column is referenced using the correlation name D declared in the PARTITION clause, whereas the output of the PTF is referenced using the correlation name R.

Refer to Subclause 3.2.4, “TopNplus”, for a fuller example of partitioning, including an example of how the correlation names and result column names work.

The ExecR example is continued in detail in Subclause 12.6, “ExecR”.

3.2.6 Similarity

Similarity performs an analysis on two data sets, which are both tables of two columns, treated as the x and y axes of a graph. The analysis results in a number which indicates the degree of similarity between the two graphs, with 1 being perfectly identical and 0 being completely dissimilar. The numeric result is returned in a table with one row and one column. The result column is called Val and is of type REAL.

The query author sees the following signature in the Information Schema:

```
FUNCTION Similarlity (
    Input1 TABLE NO PASS THROUGH
             WITH SET SEMANTICS KEEP WHEN EMPTY,
    Input2 TABLE NO PASS THROUGH
```



```
            WITH SET SEMANTICS KEEP WHEN EMPTY )
  RETURNS TABLE (Val REAL)
  NOT DETERMINISTIC
  READS SQL DATA
```

Note that in this example the result row type is known when creating the PTF; therefore, it can be specified in the DDL. The PTF author supplies the following documentation:

1) The two parameters are generic tables with set semantics. Each input table must be sorted on two numeric columns; these columns are interpreted as providing the points in an x-y plot.

2) Similarity performs an analysis on two data sets, resulting in a number which indicates the degree of similarity between the two graphs, with 1 being perfectly identical and 0 being completely dissimilar. The numeric result is returned in a table with one row and one column. The result column is called Val and is of type REAL.

The query author might write a query such as the following:

```
SELECT T1.Country, T2.Code, S.Val
FROM TABLE ( Similarity ( Input1 => TABLE (Sales) AS T1
                                    PARTITION BY Country
                                    ORDER BY (Qtr, Revenue),
                          Input2 => TABLE (Countries) AS T2
                                    PARTITION BY Code
                                    ORDER BY (Quarter, GDP)
                                    COPARTITION (T1, T2)
                         )
           ) AS S
```

This example has two partitioned input tables. When there is more than one partitioned input table, then Feature B202, “PTF Copartitioning” is relevant. If the SQL-implementation supports this feature, then the query syntax supports an optional COPARTITION clause. The COPARTITION clause specifies that the input tables identified by the correlation names T1 and T2 are to be copartitioned. Each partitioning list must have the same number of columns, and corresponding column names must be the comparable. In this example, the length of each partitioning list is 1, and the corresponding columns T1.Country and T2.Code must be comparable.

In execution, copartitioning works like this: The DBMS effectively forms a master list of all country codes from S3 and T3, eliminating duplicates. One way to do this is to perform this full outer equijoin:

```
SELECT *
FROM ( SELECT Country, 1 AS One
       FROM Sales ) AS S3
     FULL OUTER JOIN
     ( SELECT Code, 1 AS One
       FROM Countries ) AS T3
     ON ( S3.Country IS NOT DISTINCT FROM T3.Code )
```

(The IS NOT DISTINCT FROM predicate is *True* if the two comparands are equal or both null.)

For example, suppose that the distinct values of Sales.Country are 'CAN', 'JPN', and 'USA', whereas the distinct values of Countries.Code are 'CAN', 'JPN', and 'GBR'. The result of the preceding query is

S3.Country	S3.One	T3.Code	T3.One
CAN	1	CAN	1

S3.Country	S3.One	T3.Code	T3.One
JPN	1	JPN	1
USA	1		
		GBR	1

Thus there are four copartitions (for 'CAN', 'JPN', 'USA', and 'GBR') and the DBMS must start a virtual processor for each of them.

The result of the PTF invocation therefore has three columns:

1) The copartitioning column called Country.

2) The copartitioning column called Code.

3) The result of the PTF itself, called Val.

This example is continued in detail in Subclause 12.7, “Similarity”.

3.2.7 UDjoin

UDjoin performs a user-defined join. It takes two input tables, *T1* and *T2*, and matches rows according to some join criterion. It is intended that *T2* is ordered on a timestamp. UDjoin will analyze this ordered data into “clusters” of related rows, where each cluster is interpreted as representing some “event”. If two rows are tied in the ordering, they are placed in the same cluster. Some rows may be interpreted as “noise”, not representing any event.

After analyzing *T2* into event clusters, rows from *T1* are matched to the most relevant event cluster. It is possible that some rows of *T1* have no matching event cluster. It is also possible that some event clusters have no match in *T1*.

The output resembles a full outer join. If a row *R* of *T1* matches an event cluster *EC* of *T2*, then in the output *R* is joined to every row of *EC*. If *R* has no matching event cluster, then *R* is output with a null-extended row in place of the event cluster. Conversely, if an event cluster *EC* is not matched, then every row of *EC* is output with nulls in the portion of the output corresponding to *T1*.

Like a full outer join, there are range variables associated with each input table, which qualify output columns that correspond to columns of the input.

The PTF author creates this PTF with the following signature:

```
CREATE FUNCTION UDjoin (
  Candidates TABLE PASS THROUGH
    WITH SET SEMANTICS KEEP WHEN EMPTY,
  EventStream TABLE PASS THROUGH
    WITH SET SEMANTICS KEEP WHEN EMPTY
) RETURNS ONLY PASS THROUGH
```

The RETURNS ONLY PASS THROUGH syntax declares that the PTF does not generate any columns of its own; instead, the only output columns are passed through from input columns.

The query author might write the following query:

```
SELECT G.*, S.*
FROM TABLE (UDjoin ( Candidates => TABLE (Goods) AS G,
                     EventStream => TABLE (TimeSeries) AS S
                       ORDER BY Tstamp ) )
```

Note that the PTF does not generate any columns; therefore, there is no correlation name for the PTF itself, only for the input tables.

The result might look like this:

G			S		
Gid	**Golly**	**Wiz**	**Tstamp**	**Color**	**Shape**
125	Molly	Oz	3	Crimson	Diamond
125	Molly	Oz	4	Aquamaroon	Star
125	Molly	Oz	5	Vermilion	Pentagon
			8	Violet	Crescent
			9	Purple	Circle
			10	Plum	Ellipse
126	Dolly	Narnia			

In the results, the row (G.Gid = 125, G.Golly = 'Molly', G.Wiz = 'Oz') is matched to an event of three rows with Tstamp = {3, 4, 5}. The next event, with Tstamp = {8, 9, 10}, has no match in G, so the columns of G are null. The final row (G.Gid = 126, G.Golly = 'Dolly', G.Wiz = 'Narnia') has no matching event in S, where the columns are null.

3.2.8 MapReduce

MapReduce is a data processing paradigm using two phases, called Map and Reduce. In the classic “word count” example of the MapReduce paradigm, the Map phase reads one or more input files. Each input file is parsed into words separated by delimiters. Map outputs a series of records, each record being a tuple comprising a word and a count. These records are then partitioned on word. In the Reduce phase, the counts in each partition are summed. The final result is a list of words appearing in any of the input files, with their counts.

Map can be implemented using a PTF that takes as input a list of files, producing an output table with two columns, word and count. Reduce can then be performed using conventional SQL grouping and the COUNT aggregate.

In more general terms, the MapReduce paradigm has two phases, Map and Reduce. The Map phase analyzes its input into some fixed format suitable for input to the Reduce phase. The data is partitioned and Reduce performs some analysis on the partitioned data, which might not be supported by an SQL aggregate. In this

general paradigm, both the Map and the Reduce phases can be implemented by PTFs. To get the complete paradigm, the query author will write a functional composition of the two phases, schematically:

```
SELECT ...
FROM TABLE (Reduce (Map (...))) AS MT
```

The invocation of Map nested within Reduce does not need a TABLE operator because Map is known to return a table.

This paper does not discuss the MapReduce paradigm further. However, Subclause 12.9, "Nested PTF invocation", provides examples of nested PTF invocations.

3.3 The life cycle of a PTF

This paper will weave among the three audiences in order to arrive at a comprehensive understanding of the whole. We assume that both the PTF author and the query author edit text files in which they build their SQL statements. Realistically, the development cycle will include many iterations by the PTF author, and the query author may iterate the design of the query. These iterations are not shown in the our examples.

You, the reader, may fill any or all of these roles. If you are a DBMS developer, you will of course write the implementation of the DBMS functionality; during development and testing you will also play the role of PTF author and query author. If you are a PTF author, then you will also fill the role of query author to test your PTF. When testing a PTF, it is recommended that you use separate SQL-schemas for the PTF implementation and the PTF test suite. If you are a query author, you may still find it useful to understand how the DBMS and the PTF body interoperate to deliver the result of the PTF.

The remaining sections of this Technical Report and their primary audiences are shown in Table 1, "Primary audiences for Clauses and Subclauses in this Technical Report":

Table 1 — Primary audiences for Clauses and Subclauses in this Technical Report

PTF Author	DBMS Developer	Query Author
Subclause 4.1, "Processing phases"		
Subclause 4.2, "Virtual processors"		
Subclause 4.3, "PTF component procedures"		
Subclause 4.4, "Input table characteristics"		
Subclause 4.5, "Partitioning and ordering"		
Subclause 4.6, "Flow of control"		
Subclause 4.7, "Flow of information"		
Subclause 4.8, "Flow of row types"		
Subclause 4.9, "Pass-through columns"		

PTF Author	DBMS Developer	Query Author
Subclause 4.10, “Security model”		
Subclause 4.11, “Conformance features”		
Clause 5, “Specification”		
Clause 6, “Data definition language”		
Clause 7, “Implementation”		
	Clause 8, “Invocation”	
	Subclause 9.1, “Calling the describe component procedure”	
Subclause 9.2, “Inside the describe component procedure”		
	Subclause 9.3, “Using the result of describe”	
Clause 10, “Optimization”		
	Subclause 11.1, “Partitions and virtual processors”	
	Subclause 11.2, “Calling the start component procedure”	
Subclause 11.3, “Inside the start component procedure”		
	Subclause 11.4, “Calling the PTF fulfill component procedure”	
Subclause 11.5, “Inside the PTF fulfill component procedure”		
	Subclause 11.6, “Closing cursors”	
	Subclause 11.7, “Calling the PTF finish component procedure”	
Subclause 11.8, “Inside the PTF finish component procedure”		
	Subclause 11.9, “Collecting the output”	
	Subclause 11.10, “Cleanup on a virtual processor”	

PTF Author	DBMS Developer	Query Author
	Subclause 11.11, “Final result”	

The examples are developed in Clause 12, “Examples”, using the same outline.

4 PTF processing model

All of this section is addressed to the DBMS developer and the PTF author. The query author needs to understand input table semantics (Subclause 4.4, "Input table characteristics") and partitioning (Subclause 4.5, "Partitioning and ordering").

4.1 Processing phases

Primary audience: DBMS developer and PTF author

Processing a PTF invocation is divided into two phases: compilation and execution. These phases correspond to PREPARE and EXECUTE in dynamic SQL. In static SQL, the query author is not aware of the phases, since compilation is followed immediately by execution. However, the DBMS and the PTF author are always aware of these phases.

4.2 Virtual processors

Primary audience: DBMS developer and PTF author

The execution phase is described using an abstraction called a virtual processor. A virtual processor is a processing unit capable of executing a sequential algorithm. A virtual processor might be an actual physical processor (with associated operating system, *etc*.). Using techniques such as multiprocessing, a single physical processor might host several virtual processors. Virtual processors may execute independently and concurrently, either on a single physical processor or distributed across multiple physical processors. There is no communication between virtual processors. The DBMS is responsible for collecting the output on each virtual processor; the union of the output from all virtual processors is the result of the PTF.

4.3 PTF component procedures

Primary audience: DBMS developer and PTF author

The query author perceives a single PTF. The PTF author and the DBMS perceive that the PTF is composed of one to four PTF component procedures, which are invoked at various points during compilation and execution, as follows:

1) PTF describe component procedure: called once during compilation. The primary task of the PTF describe component procedure is to determine the row type of the output table. It can also initialize private data that will be passed to subsequent PTF component procedures. (This component procedure is optional.)

2) PTF start component procedure: called once per virtual processor to perform any initialization that is not done by the DBMS or the PTF describe component procedure (This component procedure is optional.)

3) PTF fulfill component procedure: called once per virtual processor to deliver the output table by “piping” rows to the DBMS. (This component procedure is required.)

4) PTF finish component procedure: called once per virtual processor to perform any clean up not performed by the DBMS. (This component procedure is optional.)

Thus a polymorphic table function is actually an organized collection of SQL-invoked procedures.

Any input scalars that are not compile-time constants are passed to the PTF describe component procedure as null. If the non-null input scalars are insufficient for the PTF describe component procedure to determine the output row type, then it may return an error, which effectively becomes a syntax error. Thus, the PTF author may effectively impose syntax constraints on the the query author by returning an error from the describe component procedure if the syntax requirements are not met.

The run-time PTF component procedures are used to process the input table(s) and generate the output table.

4.4 Input table characteristics

Primary audience: DBMS developer, PTF author, and query author.

Input tables are classified by three characteristics:

1) Input tables have either row semantics or set semantics, as follows:

   a) Row semantics means that the the result of the PTF is decided on a row-by-row basis. As an extreme example, the DBMS could atomize the input table into individual rows, and send each single row to a different virtual processor.

   b) Set semantics means that the outcome of the function depends on how the data is partitioned. A partition may not be split across virtual processors, nor may a virtual processor handle more than one partition.

2) The second characteristic, which applies only to input tables with set semantics, is whether the PTF can generate a result row even if the input table is empty. If the PTF can generate a result row on empty input, the table is said to be “keep when empty”, meaning that the DBMS must actually instantiate a virtual processor (or more than one virtual processor in the presence of other input tables). The alternative is called “prune when empty”, meaning that the DBMS can prune virtual processors from the query plan if the input table is empty. (Tables with row semantics are always effectively “prune when empty”, so this choice is not relevant to them.)

3) The third characteristic is whether the input table supports pass-through columns or not. Pass-through columns is a mechanism enabling the PTF to copy every column of an input row into columns of an output row.

In the examples, the table parameters have characteristics as shown in the following table:

Example	Table Parameter	Row or set semantics?	Keep or prune when empty?	Pass-through?
CSVreader	no table parameters			

Example	Table Parameter	Row or set semantics?	Keep or prune when empty?	Pass-through?
Pivot	Input	row	(N/A)	yes
Score	Data	row	(N/A)	yes
	Model	set	prune	no
TopNplus	Input	set	keep	no (example could be worked using pass-through)
ExecR	Input	set	keep	no
Similarity	Input1	set	keep	no
	Input2	set	keep	no
UDjoin	Candidates	set	prune	yes
	EventStream	set	prune	yes

4.5 Partitioning and ordering

Primary audience: DBMS developer, PTF author, and query author.

The input tables with set semantics may be partitioned on one or more columns, at the discretion of the query author. An input table with at least one partitioning column is said to be partitioned (even if, in fact, the partitioning column has only one value). An input table with no partitioning column is said to be broadcast. In the examples, TopNplus, ExecR, and Similarity illustrate partitioning.

The input tables with set semantics may be ordered on one or more columns (other than partitioning columns — there is no benefit to ordering on a partitioning column), also at the discretion of the query author. The PTF is aware of the ordering during all PTF component procedures, and may utilize the ordering semantically (TopNplus illustrates this possibility). If the PTF does not use the ordering semantically, then there is no benefit to the query author in ordering the input table. Ordering an input table does not imply an ordering to the results or the query as a whole.

Input tables with row semantics may not be partitioned or ordered. Row semantics implies that the result can be determined on a row-by-row basis; therefore, the DBMS can assign rows of such tables to virtual processors arbitrarily, for example, using random, round robin, or load balancing algorithms. The Pivot and Score examples illustrate input tables with row semantics.

When there is more than one partitioned input table, they can, at the option of the query author, be copartitioned. With copartitioning, the copartitioned table arguments must have the same number of partitioning columns, and corresponding partitioning columns must be comparable. The DBMS effectively performs a full outer equijoin on the copartitioning columns, assigning one virtual processor to each combination of partitions

resulting from this equijoin. The Similarity example illustrates copartitioning. For detailed examples of copartitioning, see Subclause 12.7.9, “Virtual processors for Similarity”.

4.6 Flow of control

Primary audience: DBMS developer and PTF author.

This section presents a conceptual architecture for the flow of control during the compilation and execution.

Refer to the following diagram, showing a schematic execution plan involving multiple virtual processors P_1, P_2, P_3, ...

Compile Time		**Virtual Processor**	**Run Time**		
Describe			**Start**	**Fulfill**	**Finish**
		P_1			
		P_2			
		P_3			

The flow of execution moves basically left to right. The describe step is not shown using any virtual processor, since it is an indivisible computational step.

The purpose of compilation is to set up for the subsequent execution phases. Compilation is performed without the ability to read the input data, though the row type and sort order of the input tables are passed to the PTF describe component procedure via PTF descriptor areas.

Compilation results in two kinds of information:

1) The row type of the result.

2) Values of the private variables of the PTF, if any. These values are saved by the DBMS and re-instantiated as input to the run-time PTF component procedures. This provides for information flow from the PTF describe component procedure to the run-time PTF component procedures.

The time between compilation and execution is indicated by the vertical blank space between them. If the query is prepared and executed as separate steps, there can be many executions, not portrayed in this diagram. For example, if the PTF invocation is in a view definition, then the PTF invocation is compiled when the view is defined and executed when the view is referenced in a query.

At run-time, the DBMS assembles the input data, partitions it, and directs each partition to a separate virtual processor. Each virtual processor executes independently of every other virtual processor. Virtual processors may be scheduled sequentially on the same physical processor, or concurrently on the same or different physical processors. Scheduling virtual processors is implementation-dependent.

Note that partitioning is only semantically correct if the overall task can be decomposed as a union of disjoint tasks. If an input table has row semantics, then the input table can be partitioned arbitrarily, so in that case the

partitioning decision is made by the DBMS according to an implementation-dependent algorithm. If an input table has set semantics, then the partitioning decision is the responsibility of the query author, and the default is to form a single partition.

4.7 Flow of information

Primary audience: DBMS developer and PTF author.

Information flows between stages of execution as follows:

1) The PTF describe component procedure receives a description of the input tables and their ordering (if any) as well as any scalar input arguments that are compile-time constants.

2) The PTF describe component procedure returns the row type of the PTF's result to the DBMS. This row type is also made available to all the run-time PTF component procedures. (If any table argument has pass-through columns, the handling of the result row type has some additional complexity, discussed in Subclause 4.9, “Pass-through columns”.)

3) The PTF describe component procedure might have private information to communicate to the other PTF component procedures that will execute in run time. For example, the PTF describe component procedure may go to some trouble to analyze the input scalars and tables, and it may be useful to pass a digest of this analysis to the run-time PTF component procedures, so that they do not need to repeat the analysis.

4) The run-time PTF component procedures may have information to pass from one stage to another. For example, if a resource is allocated during the start component procedure, then a handle for that resource should be passed to the fulfill component procedure to use the resource, and to the finish component procedure to deallocate it.

The following diagram illustrates the flow of information in a PTF invocation:

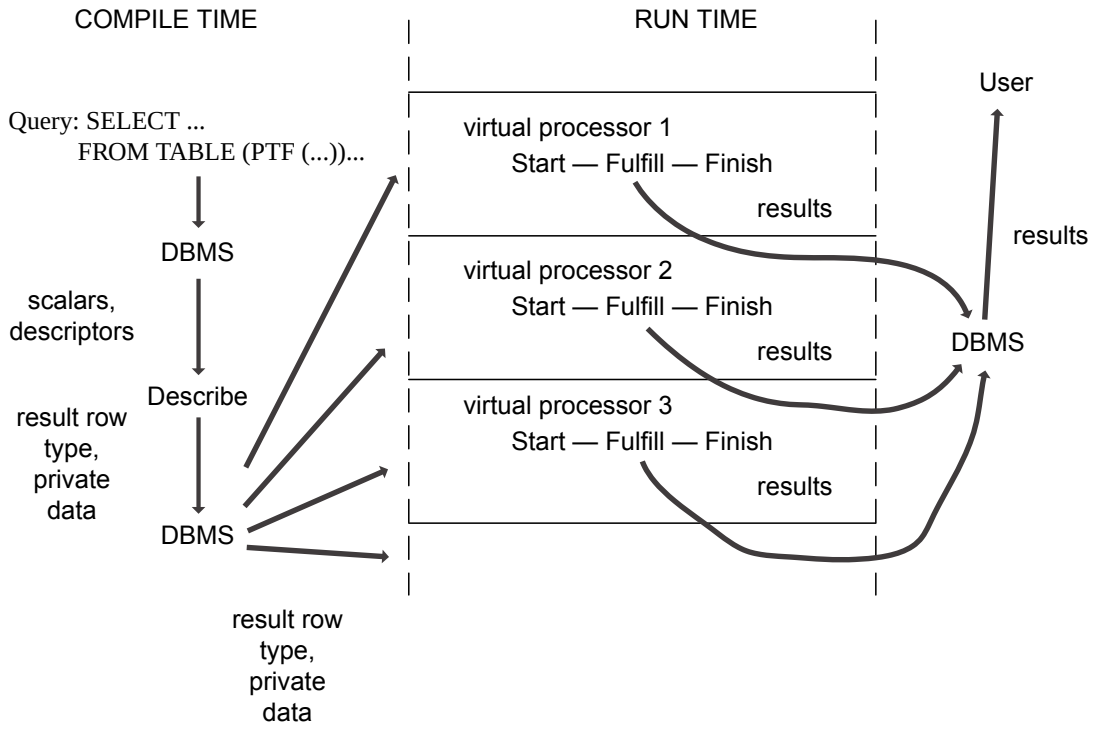

Figure 1 — PTF information flow

4.8 Flow of row types

Primary audience: DBMS developer and PTF author.

For each table argument, a PTF invocation has the following input row types:

1) The full row type of an input table, including every column of the input table. If the input table has column renaming (specified by a <parenthesized derived column list>), then the column names in the <parenthesized derived column list> are used; otherwise, the original column names of the input table are used.

2) The requested row type of an input table; this includes only those columns that the PTF describe component procedure requests to receive from the DBMS. The PTF describe component procedure is responsible for describing this row type; it must not be empty. (If the PTF describe component procedure is absent, then this is a copy of the full row type.)

3) The cursor row type of an input table. This is the same as the requested row type, plus one additional column (the pass-through input surrogate column) if the input table has pass-through columns (see Subclause 4.9, “Pass-through columns”).

A PTF invocation also has the following result row types:

1) The initial result row type. This row type lists the columns that the PTF itself will generate (called the proper result columns of the PTF).

2) The intermediate result row type. This is identical to the initial result row type, plus one additional column (the pass-through output surrogate column) for each input table that has pass-through columns. This is the row type that must be used when performing <pipe row statement> during the execution phase to output a row.

3) The external result row type. This is the same as the initial result row type, except that the proper result columns may be renamed by the query using a <parenthesized derived column list>.

4) The complete result row type (called just the "row type of <table primary>" in the standard) comprising the external result row type plus, for each table argument *TA*:

   a) If *TA* has pass-through columns, then, for every column of *TA*, a result column having the same name and data type.

   b) Otherwise, for every partitioning column of *TA*, a result column having the same name and data type.

The relationships between these row types is illustrated in the following diagram:

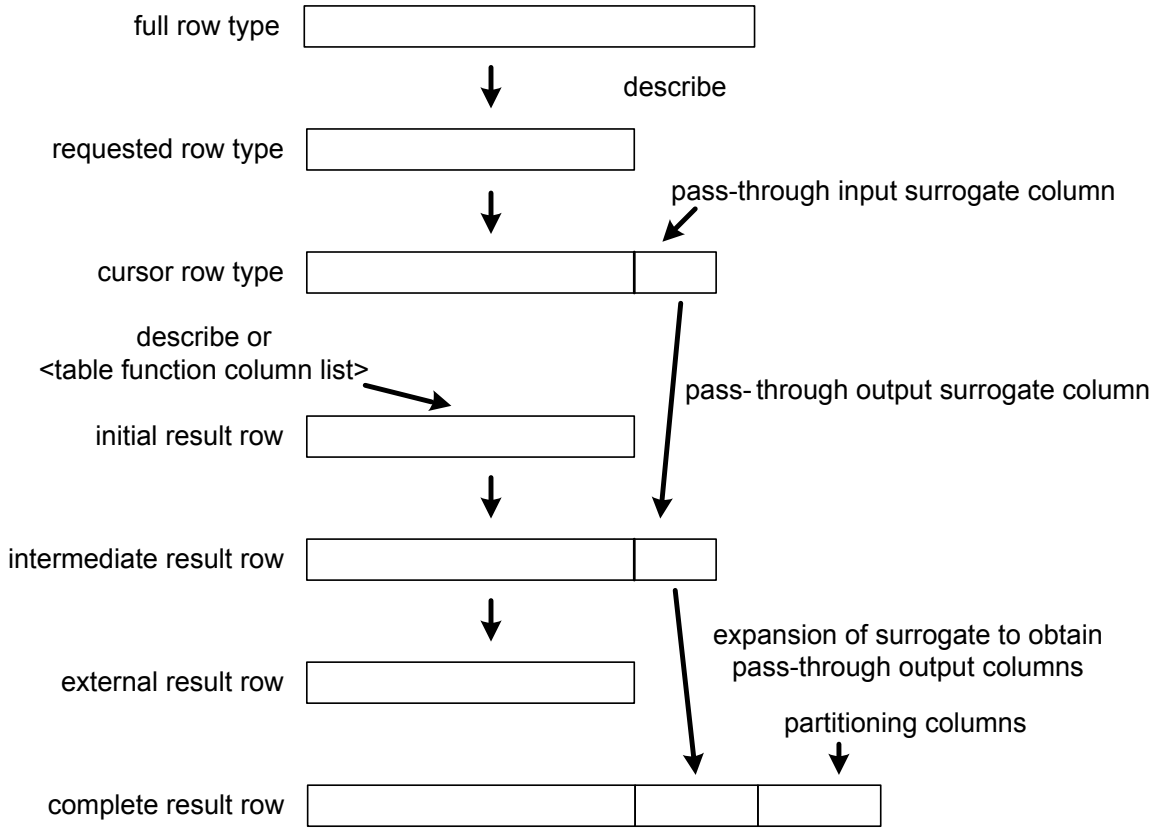

Figure 2 — Row type relationships

4.9 Pass-through columns

Primary audience: DBMS developer and PTF author.

Pass-through columns are a mechanism whereby a PTF can copy all columns of an input row into a result row, without needing to fetch the columns individually, and without needing to understand their data types. From the PTF's perspective, pass-through columns are condensed into a single surrogate value. The query is not aware of the surrogates and instead sees that an input row has been copied into the result.

Refer to Figure 2, "Row type relationships". For each table argument, typically the PTF describe component procedure requests a subset of the full row type, comprising just the columns that the PTF needs to examine for its semantics. If the input table has pass-through columns, then the cursor row type is formed by appending a single column of implementation-dependent name and type, called the pass-through input surrogate column.

As for the result of a PTF, either the PTF describe component procedure or the <table function column list> declared in the CREATE FUNCTION statement specifies the initial result row type, consisting of the proper result columns (the columns that the PTF itself will generate). The intermediate result row type consists of a copy of the initial result row type, plus one pass-through output surrogate column for each table argument that has pass-through columns.

During the execution phase, the PTF fulfill component procedure receives PTF descriptor areas for the cursor row types and the intermediate result row type. When the PTF fulfill component procedure fetches from a cursor, the DBMS populates the pass-through input surrogate column with an opaque value that represents the non-partitioning columns of the table, including both requested and non-requested columns. For example, this surrogate might be formed by compressing the columns into a BLOB, or it might be a candidate key. (Note that by using common column projection algorithms, the surrogate value actually only needs to represent those columns that are actually referenced later in the query.)

The PTF fulfill component procedure can copy the value of the pass-through input surrogate column to the pass-through output surrogate column in the intermediate result row. The intermediate result row is sent to the DBMS by a <pipe row statement>, whereupon the DBMS expands the surrogate value to reconstruct the non-partitioning columns that the surrogate value represents.

Alternatively, the PTF fulfill component procedure can place a null value in the pass-through output surrogate column. When transmitted to the DBMS by a <pipe row statement>, the null value of the surrogate will expand into null values in all the pass-through result columns. (Note that this has no effect on partitioning columns, since the surrogate does not represent them.) This is the only output scenario available to the PTF start and finish component procedures, which can use a <pipe row statement>, but have no cursor and hence no input surrogate value.

Partitioning columns are handled separately from pass-through columns. The values of partitioning columns are invariants on a virtual processor. Since the virtual processor is the processing unit that processes a <pipe row statement>, the virtual processor is able to add the values of the partitioning columns into the complete result row.

4.10 Security model

Primary audience: DBMS developer and PTF author.

The security model for PTFs is as follows:

1) The query author must have EXECUTE privilege on the PTF in order to invoke it.

2) The query author must have SELECT privilege on input tables (more precisely, on the columns of the input tables).

3) In this Technical Report, the owner of the PTF is generally portrayed as the owner of the PTF component procedures. This is likely to be the case in practice, but the minimal requirement is merely that the owner of the PTF has EXECUTE privilege on all PTF component privileges.

4) The PTF does not need SELECT privilege on the input tables or their columns; only the query author needs privilege on the input tables. The DBMS opens the input tables and passes open read-only cursors to the PTF fulfill component procedure. These cursors are anonymous in the sense that the PTF does not know the identity of the input tables. The only operation the PTF can perform on these input cursors is FETCH.

5) If a PTF needs a side table to perform a “table lookup”, the PTF author has three ways to do this:

   a) If the lookup table is proprietary to the PTF (perhaps it is the intellectual property of the PTF author), then the PTF may perform SELECT operations on the proprietary table by opening it in the PTF component procedures using “definer's rights”. There is no need to grant SELECT on the proprietary table to the query author.

   b) If the lookup table is not proprietary to the PTF, then the PTF can expect the query author to pass the lookup table as an input table, in which case the table will be subject to access checking using the query author's privileges.

   c) If the preceding techniques are not sufficient, then the PTF can expect the query author to pass text arguments containing the names of tables, *etc*., from which the PTF can build a dynamic query. This dynamic query must be access-checked using the query author's privileges, so the PTF component procedure that does this must be created with “invoker's rights”.

Note that the implementation techniques described in Clause 7, “Implementation”, have no access checking and can be used freely in either definer's rights or invoker's rights component procedures. The PTF author does not need to be concerned with definer's rights or invoker's rights unless the PTF falls under either scenario a) or c) above.

4.11 Conformance features

Support for polymorphic table functions is an optional feature of SQL. If the DBMS provides minimal support for polymorphic table functions, as specified in [ISO9075-2], then the DBMS can claim support for Feature B200, “Polymorphic table functions”.

[ISO9075-2] specifies additional advanced features that require support for Feature B200, “Polymorphic table functions”, and enrich that minimal support with extra functionality. These additional conformance features are as follows:

— Feature B201, “More than one PTF generic table parameter”

  This feature permits a polymorphic table function to have more than one generic table parameter. Examples of multi-table input to a PTF are found in Subclause 12.4, “Score”, Subclause 12.7, “Similarity”, and Subclause 12.8, “UDjoin”.

— Feature B202, “PTF Copartitioning”

This feature provides support for copartitioning. The example for copartitioning is found in Subclause 12.7, "Similarity".

— Feature B203, "More than one copartition specification"

This feature permits a polymorphic table to have more than one copartitioning specification. At least four input tables are required to utilize this feature (two input tables for each of two copartitioning specifications). There are no examples of this feature in this Technical Report.

— Feature B204, "PRUNE WHEN EMPTY"

This feature permits the DBMS to avoid creating a virtual processor for a partition that is known to be empty. Syntactically, it can be specified by either the PTF author or the query author.

The PTF author specifies PRUNE WHEN EMPTY in DDL syntax of a table parameter if the PTF author knows that the result of the PTF for a partition is empty when the input partition has no rows. From the standpoint of the DBMS, it is an optimization if the DBMS can avoid creating a virtual processor for an empty input partition. However, from a functionality standpoint, the outcome is the same whether the virtual processor is created or not, since the result is empty in either case. If the PTF can generate a result even when given an empty input data set, the PTF author should not specify PRUNE WHEN EMPTY. Examples of this DDL syntax are found in Subclause 12.4, "Score", and Subclause 12.5, "TopNplus".

If the PTF can generate a result on an empty input partition, the query author may not be interested in that result. In that case the query author can specify PRUNE WHEN EMPTY in the query syntax. An example of PRUNE WHEN EMPTY in query syntax is found in Subclause 12.7, "Similarity".

— Feature B205, "Pass-through columns"

Pass-through columns are a device that the DBMS can provide to the PTF author, making it easy for the PTF author to copy an input row into the output. Examples of this are found in Subclause 12.1, "Projection", Subclause 12.3, "Pivot", Subclause 12.4, "Score", and Subclause 12.8, "UDjoin". In addition, Subclause 12.5, "TopNplus", shows how the PTF author can copy an input row to an output row even if the DBMS does not support this Feature.

— Feature B206, "PTF descriptor parameters"

PTF descriptor parameters are a mechanism for the query author to pass a row type as an argument to a polymorphic table function. Examples of PTF descriptor parameters are found in Subclause 12.1, "Projection", Subclause 12.2, "CSVreader", Subclause 12.3, "Pivot", and Subclause 12.6, "ExecR".

— Feature B207, "Cross products of partitionings"

With this feature, if an invocation of a polymorphic table function has more than one partitioned input table, then it is the query author's choice whether to relate the partitioned input tables using copartitioning. (Otherwise, all partitioned input tables must be related to one another via a single copartitioning specification.) The possibility of multiple partitioned input tables that are not copartitioned is illustrated in Subclause 12.7.9, "Virtual processors for Similarity", following the example of copartitioning.

There are also two conformance features that are relevant only to the PTF author and the DBMS developer:

— Feature B208, "PTF component procedure interface"

[ISO9075-2] specifies an optional interface between the DBMS and the polymorphic table function. The interface is provided as a specification device, to specify the semantics of an invocation of a polymorphic table function. An SQL-implementation is not required to use the specified interface; it may substitute an equivalent interface that provides the same functionality to the PTF author. If the DBMS adheres to the

interface as specified in [ISO9075-2], then the DBMS may claim conformance to Feature B208, “PTF component procedure interface”. All of the examples in this Technical Report assume this interface.

— Feature B209, “PTF extended names”

PTF extended names are a distinctive category of dynamic extended names, used to name PTF cursors and PTF descriptor areas. PTF extended names are part of the optional interface between the DBMS and the polymorphic table function. It is possible that an SQL-implementation may choose to support PTF extended names without supporting other aspects of the interface. In that case, the DBMS may claim conformance to Feature B209, “PTF extended names”, even if it does not conform to Feature B208, “PTF component procedure interface”. All of the examples in this Technical Report assume this feature.

(Blank page)

5 Specification

Primary audience: PTF author

Specification is the planning phase of software development, prior to implementation. For our purpose in this Technical Report, the specification phase results in what we call “skeleton DDL”, that is, all of the CREATE statements necessary to define a PTF, except for the body of the PTF. Specification can be divided into two subphases:

— Functional specification specifies the software as seen from the outside (as a “black box”). This phase specifies the user interface and semantics of the software. The functional specification can become the basis for documentation addressed to the query author.

— Design specification specifies the software as seen from the inside (“white box”). This may be proprietary information not divulged to the query author.

5.1 Functional specification

The first step in the life cycle is to write a functional specification for the PTF. The functional specification will describe the user interface and semantics of the PTF, without describing the inner design of the PTF. The functional specification becomes the basis for documentation supplied to the query author.

5.1.1 Parameter list

The first step in writing a functional specification is to decide the parameter list, which might include the following things:

1) The input table(s). These are generic tables, so the input tables should be thought of in terms of their role within the transformation that the PTF implements. Note that Feature B201, “More than one PTF generic table parameter”, is required if the PTF has more than one input table.

2) The scalar inputs. The PTF can use scalar inputs to parameterize the behavior of the PTF.

3) The PTF descriptor area inputs. A PTF descriptor area can provide a list of column names, possibly augmented by data types. Of course, a list of column names can be provided via a character string scalar; however, this requires the PTF to provide a parsing capability, with attention to case-sensitivity rules. In general, if a PTF can deduce a list of columns without a PTF descriptor area, that will be preferable from the standpoint of the PTF author's customer, the query author. However, in totally dynamic situations, the query author may have to provide column lists, and PTF descriptor areas will probably be the most convenient way to do this. PTF descriptor areas are discussed in Subclause 7.1, “PTF descriptor areas”. The examples Pivot and ExecR illustrate descriptor parameters. Note that Feature B206, “PTF descriptor parameters”, is required if there are any PTF descriptor area inputs.

After deciding on the parameter list, the PTF author is ready to write the first skeleton CREATE FUNCTION statement. At this stage we have an incomplete CREATE FUNCTION because it only lists the input parameters (there is more DDL to come later). The parameters are declared with the following types:

— Input tables have parameter type TABLE.

— Input scalars have their usual parameter types (VARCHAR, INTEGER, *etc.*).

— Input PTF descriptor areas have parameter type DESCRIPTOR.

Thus, at this stage, the function declaration looks something like this:

```
CREATE FUNCTION ptf (
  Input TABLE,
  Scalar INTEGER,
  Columns DESCRIPTOR )
RETURNS TABLE
```

At this stage the PTF author may also be able to decide which inputs are optional. Optional inputs are indicated by specifying a default value; for example:

```
CREATE FUNCTION ptf (
  Input TABLE,
  Scalar INTEGER DEFAULT 3,
  Columns DESCRIPTOR DEFAULT NULL )
RETURNS TABLE
```

Input tables are always mandatory; you cannot specify a default for a table input.

Advice to the DBMS developer: you want to support the PTF author at this early stage. The CREATE FUNCTION statement above is incomplete and not suitable for actual use. Nevertheless, the DBMS may want to allow the PTF author to load such a definition as a kind of “invalid” function definition. This will facilitate a DBMS tool that can assist the PTF author as the latter goes through the stages of development outlined here. We will talk more about how the DBMS can assist the PTF author later in Subclause 5.2.4, “Component procedure signatures”.

5.1.2 Input table semantics

After listing the parameters, the next specification step is to classify each input table as row semantics or set semantics.

— Row semantics means that the the result of the PTF is decided on a row-by-row basis. As an extreme example, the DBMS could atomize the input table into individual rows, and send each single row to a different virtual processor. Or the DBMS might process them all on the same virtual processor. A table should be given row semantics if the PTF does not care how rows are assigned to virtual processors.

— Set semantics means that the outcome of the function depends on how the data is partitioned. A table should be given set semantics if all rows of a partition should be processed on the same virtual processor. This is the default semantics.

At most one input table may have row semantics; all other input tables must have set semantics. A PTF that has an input table with row semantics is said to be a per-row PTF; otherwise, the PTF is said to be a per-set PTF.

In our examples:

— CSVreader has no input tables.

— Pivot has an input table with row semantics.

— Score has one input table with row semantics and one with set semantics.

— TopNplus has an input table with set semantics.

— ExecR has an input table with set semantics.

— Similarity and UDjoin have two input tables with set semantics.

5.1.3 Prunability

If the DBMS supports Feature B204, “PRUNE WHEN EMPTY”, then prunability is the next step in functional specification. In this case, if an input table has set semantics, then there is a further property: whether the table can be pruned or not. An input table is prunable if the the result of the PTF with an empty input table is an empty output table. (Prunability is not a choice for input tables with row semantics, since an empty input row necessarily generates no output rows.)

We have five examples that have input tables with set semantics:

— Score has one table parameter with row semantics and one with set semantics. The second table parameter is used to define a model to score rows of the first parameter. It is impossible to set up a model with an empty table, so this is be PRUNE WHEN EMPTY.

— TopNplus: on an empty input, it would be very reasonable to make an empty output, so we could specify PRUNE WHEN EMPTY. Alternatively, we might want to generate a single row with a sum of 0 in the sort column and the other columns null. If we do that, then we should specify KEEP WHEN EMPTY.

— ExecR: An R script could potentially have output even on an empty input, therefore ExecR should be KEEP WHEN EMPTY

— Similarity: we'll assume that the similarity algorithm computes that an empty table is completely dissimilar from a non-empty table (result = 0), and completely similar to another empty table (result = 1). Thus, there is a result even if one or both inputs is empty; therefore, both input tables should be KEEP WHEN EMPTY.

— UDjoin: since the semantics resemble a full outer join, we may have a result even if either table is empty, so both input tables should be KEEP WHEN EMPTY.

If the DBMS does not support Feature B204, “PRUNE WHEN EMPTY”, then every table is effectively KEEP WHEN EMPTY. This does not impact the functionality of the PTF (an empty partition simply generates an empty result), but it may cost some performance since the DBMS may needlessly instantiate virtual processors for empty partitions.

5.1.4 Pass-through columns

If the SQL-implementation supports Feature B205, “Pass-through columns”, then the final characteristic of input tables is whether they support pass-through columns or not. Pass-through columns is a feature that enables

the PTF to copy all columns of a row of an input table into an output row, without copying the columns individually and without needing to understand the data types of the columns. (See Subclause 4.9, "Pass-through columns", for a discussion of how this works.) If a table parameter has pass-through columns, then every column of the table argument is available to the query, qualified by the range variable of the table argument. The examples Pivot, Score, and UDjoin illustrate pass-through columns (TopNplus could be redesigned to exploit pass-through columns too). The keywords PASS THROUGH indicate that a table parameter uses pass-through columns; NO PASS THROUGH indicates that a table parameter does not. If the SQL-implementation does not support Feature B205, "Pass-through columns", then this choice is not available in DDL and every input table is effectively NO PASS THROUGH. A PTF can still copy an input row into the output, but it requires more effort on the PTF author's part, and there may be a performance penalty because the PTF may need to request input columns it would not otherwise need in order to copy them to the output. This scenario is illustrated in Subclause 12.5, "TopNplus".

5.1.5 Result row type

The result row type of a PTF invocation consists of the following:

1) Columns generated by the PTF; these are called the proper result columns of the PTF.

2) Partitioning columns of partitioned table arguments.

3) For any table argument that has pass-through columns, the non-partitioning columns of that table argument.

Item 2), and item 3) are determined by decisions already discussed. As for the proper result columns, there are three options:

— If the proper result columns are already known during the specification stage, they can be specified in the return type CREATE FUNCTION statement. The examples Score and Similarity illustrate this possibility; if TopNplus is redesigned to use pass-through columns, then it could be redesigned to do this as well.

— If there are no proper result columns, then the return type RETURNS ONLY PASS THROUGH can be specified. The example UDjoin illustrates this possibility.

— Otherwise, the PTF describe component procedure is mandatory and determines the proper result columns.

5.1.6 Determinism

A PTF is deterministic if it necessarily produces the same set of rows when re-executed using a particular set of inputs. The inputs are the values of scalar arguments, descriptor arguments, and table arguments. The value of a table argument comprises a multiset of rows, as well as the partitioning and ordering of those rows as specified by the query.

In particular, if the result of a PTF depends on the ordering of rows, then the PTF is nondeterministic. TopNplus is a good example. The second parameter, Howmany, tells how many rows in each partition to copy from the input to the output. Suppose that Howmany is 3, and suppose that the first four rows of a partition are tied in the ordering. Then it is non-deterministic which three rows will be copied into the output. If, on the other hand, TopNplus was defined so that it copies Howmany rows, plus any ties to the last of these rows, then TopNplus becomes deterministic.

Thus, input ordering is potentially significant to the determinism of a PTF. On the other hand, the order in which output rows are generated is not significant to the determinism of a PTF.

The contents of SQL-data that is not passed in table argument(s) is not regarded as input to a PTF. Thus if the result of the PTF depends on the contents of proprietary data (see Subclause 4.10, “Security model”, item 5)a))) or on the result of a query constructed dynamically (see Subclause 4.10, “Security model”, item 5)c))), then the PTF is non-deterministic.

5.1.7 SQL-data access

SQL-data access is a property of SQL-invoked routines that specifies the degree of access to SQL-data that the SQL-invoked routine requires. There are four choices: NO SQL, CONTAINS SQL, READS SQL DATA, and MODIFIES SQL DATA. A PTF will almost never have SQL-data access of NO SQL, because at the very least, the PTF fulfill component procedure will use a PIPE ROW statement to output a row. At the other extreme, a PTF is not allowed MODIFIES SQL DATA.

This leaves CONTAINS SQL or READS SQL DATA. READS SQL DATA is appropriate if the PTF has an input table, or if it performs a lookup in a side table as described in Subclause 4.10, “Security model”, item 5). Otherwise, CONTAINS SQL is the appropriate SQL-data access.

5.1.8 Documenting the PTF to the query author

After completing the functional specification, the PTF author can write the documentation that will be made available to the query author, telling the query author how to write an acceptable invocation of the PTF and what the result will be.

5.2 Design specification

Next, the PTF author can start a design specification. The principle difference is that the functional specification should be made available to the query author in some fashion, such as user documentation, whereas the design specification can remain confidential to the PTF author.

5.2.1 Name the component procedures

The PTF author must choose names for the PTF component procedures. There are one to four PTF component procedures:

— "describe": PTF component procedure to be invoked during query compilation (optional).

— "start": PTF component procedure to be invoked at the start of execution on a virtual processor (optional).

— "fulfill": PTF component procedure to be invoked during execution; this is the component procedure that reads the input tables and generates the output table (mandatory).

— "finish": PTF component procedure to be invoked at the end of execution on a virtual processor (optional).

The PTF describe component procedure is optional if the PTF has no proper result columns, or if the proper result columns are declared statically in the CREATE FUNCTION statement; otherwise, it is mandatory. However, even if it is optional, it may still be useful because the PTF describe component procedure can validate the input arguments, initialize the private data, and reduce the list of columns in the input cursor(s) to just the columns that the PTF needs semantically.

Many PTFs will not need a start or finish component procedure. The DBMS will provide complete infrastructure for the input tables and the output stream, so the start component procedure is not needed to initialize that infrastructure and the finish component procedure is not needed to clean it up.

Thus, the start and finish component procedures only need to worry about other resources that the PTF needs during the execution phase. For example, a PTF may wish to open an operating system file during the start component and close it again during the finish component. Alternatively, the fulfill component procedure could do both the file open and close. Thus, it is more a matter of programming style whether to have start and finish component procedures.

The PTF fulfill component procedure is mandatory. This is the only one that receives cursors to read the input table arguments.

5.2.2 Private data

The design specification can also specify private data for the PTF component procedures. The private data is passed between the PTF component procedures (with the DBMS as an intermediary), but it is not exposed to the query author. The DBMS perceives the private data as a set of variables that it must allocate and pass to the PTF component procedures. Each PTF component procedure perceives the private data as arguments in its argument list.

Private data can be of any SQL types that have bindings in the implementation language. The private data is passed by the DBMS as INOUT parameters to the PTF component procedures. The DBMS makes no use of the private data, simply passing it around between the PTF component procedures to enable them to communicate.

Private data serves two purposes:

1) The describe component procedure may analyze the input arguments, and pass a digest to the later PTF component procedures in the execution phase. This way, the execution phase component procedures do not need to re-analyze the input arguments.

2) If the execution phase has start and finish component procedures, then the private data can be used for communication during the execution phase. For example, the start component procedure might open a file and place a “handle” in the private data, so that the fulfill component procedure can read the file and the finish component procedure can close the file.

Optionally you can specify default values for the private data. Private data defaults to null if no explicit default is specified. The describe component procedure will see the private data initialized to the default values. Subsequent component procedures see the private data as it was last set by the preceding component procedure in the sequence: describe — start — fulfill — finish.

5.2.3 Routine characteristics of the component procedures

The PTF author must decide the routine characteristics of the component procedures. The routine characteristics are as follows:

Table 2 — PTF routine characteristics

Characteristic	Value
<language clause>	SQL, C, *etc.*
<parameter style clause>	Not used with language SQL. For external languages, either PARAMETER STYLE GENERAL or PARAMETER STYLE SQL
<specific name>	PTF author's choice
<deterministic characteristic>	A PTF component procedure is deterministic if it produces the same results (initial result row type, result rows, status code) when invoked with the same scalar arguments, PTF descriptor areas and PTF extended cursors. A PTF fulfill component procedure is non-deterministic if the result rows depend on the order of the PTF extended cursor(s). The PTF describe component procedure must be deterministic. The PTF start, fulfill, and finish component procedures must be deterministic if the PTF is to be deterministic.

Characteristic	Value
<SQL-data access indication>	— The describe component procedure will usually be CONTAINS SQL, though READS SQL DATA is possible. — If the start and finish component procedures only manage external resources such as files, then they will usually be CONTAINS SQL if the implementation language is SQL and NO SQL for external languages such as C, though READS SQL DATA is possible. If they output rows using <pipe row statement>, then they will usually be CONTAINS SQL. — If there is a table argument, then the fulfill component procedure will probably be READS SQL; otherwise, CONTAINS SQL may be sufficient.
<null-call clause>	(omit)
<returned result sets characteristic>	Must be DYNAMIC RESULT SETS 0.
<savepoint level indication>	Must be OLD SAVEPOINT LEVEL
security (<rights clause> if the language is SQL/PSM; otherwise, <external security clause>)	Irrelevant if the only SQL statements are FETCH from a table argument cursor or SET/GET/COPY DESCRIPTOR of PTF descriptor arguments. Definer's rights may be used if the PTF executes SQL against tables proprietary to the PTF (as opposed to the query author's data, which should be passed via table arguments). Invoker's rights is permitted but will usually not be necessary.

Advice to the DBMS: the PTF component procedure characteristics cannot be deduced from the PTF declaration. The DBMS tool should provide some kind of interface for the PTF author to specify them, preferably in a file, to support an iterative development process.

5.2.4 Component procedure signatures

At this point, the PTF author has skeleton DDL that declares the PTF input parameters, the PTF private data, and the names of the PTF component procedures. The DBMS should provide some kind of DBMS tool for the next step, which is generating the parameter lists of the PTF component procedures. The DDL already specified implies the parameter lists for the PTF component procedures, so this step can be done manually in principle, but in practice it will be useful to have a DBMS tool to do this step.

In general, the parameter list of a PTF component procedures is derived from the parameter list of the PTF itself as follows:

1) The private parameters are placed at the head of the parameter list of the PTF component procedure, in order of declaration in the skeleton DDL.

2) The parameters of the PTF come next, in order of declaration.

   a) Scalar parameter declarations are simply copied from the PTF parameter list to the PTF component procedure parameter list.

   b) A DESCRIPTOR parameter is passed as a single VARCHAR parameter, for the PTF extended name of the PTF descriptor area.

   c) A table parameter is passed as a consecutive list of two to four VARCHAR parameters that name the PTF descriptor areas and cursors relevant to that table parameter and PTF component procedure. The precise list of VARCHAR parameters depends on the table parameter's semantics (row or set semantics) and the PTF component procedure, as shown in the following table:

Table 3 — Table parameter semantics

PTF component procedure	Table parameter semantics	
	Row semantics	**Set semantics**
describe	full row type descriptor name requested row type descriptor name	full row type descriptor name partitioning descriptor name ordering descriptor name requested row type descriptor name
start		partitioning descriptor name ordering descriptor name
fulfill	cursor row type descriptor cursor name	cursor row type descriptor partitioning descriptor name ordering descriptor name cursor name
finish		partitioning descriptor name ordering descriptor name

3) Next there is a single VARCHAR parameter for the PTF extended name of the result row type's PTF descriptor area.

   a) For the describe component procedure, this parameter is called the initial result row type descriptor. The describe component procedure populates the initial result row type descriptor to describe the proper result columns of the PTF. (This parameter is omitted for the PTF describe component procedure if there are no proper result columns — RETURN ONLY PASS THROUGH — or the return type declares a fixed row type.)

   b) For the start, fulfill and finish component procedures, this parameter is called the intermediate result row type descriptor. The DBMS forms the intermediate result row type descriptor from the initial result row type descriptor plus pass-through output surrogate columns if there are any pass-through table parameters.

4) Finally, there is a CHAR(5) parameter for the status code.

As seen above, many arguments that are passed to the PTF component procedures are PTF extended names. PTF extended names are discussed in Subclause 7.2, "PTF extended names". PTF extended names are character strings generated by the DBMS, so the DBMS controls their lengths. They can usually be rather short names; 1, 2 or 3 characters will probably suffice. For example, using ten digits and 26 letters can support up to 36 different input tables, each with a different distinctive character. Another character can be used to distinguish the type of PTF extended name (*e.g.*, C for cursor, R for row type, P for partitioning, and S for sort order.) Thus, 2-character names will easily support up to 36 input tables and 36 DESCRIPTOR parameters. In the following table, let *n* be the maximum number of characters in a PTF extended name.

The transformation of PTF private data and PTF input parameter to PTF component procedure parameter is summarized in the following table:

Table 4 — Corresponding PTF component procedure parameters

PTF private data and input parameters	**Corresponding PTF component procedure parameter(s)**
For each parameter:	
scalar	scalar
PTF descriptor area	VARCHAR(*n*)
table	two to four VARCHAR(*n*) parameters as described above
At the end of the parameter list:	
result row type descriptor area	VARCHAR(*n*) (omitted if the PTF has no proper result columns, or the proper result columns are declared in the CREATE FUNCTION)
status code	CHAR(5)

For example, consider the following skeleton DDL:

```
CREATE FUNCTION Ptf (
  Input1 TABLE WITH ROW SEMANTICS,
  Input2 TABLE WITH SET SEMANTICS,
  Par INTEGER,
  Desc DESCRIPTOR )
RETURNS TABLE
NOT DETERMINISTIC
READS SQL DATA
PRIVATE DATA (
  Priv INTEGER )
DESCRIBE WITH PROCEDURE Ptf_describe
START WITH PROCEDURE Ptf_start
FULFILL WITH PROCEDURE Ptf_fulfill
FINISH WITH PROCEDURE Ptf_finish
```

Then the signature for Ptf_describe is generated as follows:

```
CREATE PROCEDURE Ptf_describe (
```



```
  /* private data */
  INOUT Priv INTEGER,
  /* Input1 (row semantics) */
  IN Input1_row_descr VARCHAR(2),
  IN Input1_request_descr VARCHAR(2),
  /* Input2 (set semantics) */
  IN Input2_row_descr VARCHAR(2),
  IN Input2_pby_descr VARCHAR(2),
  IN Input2_order_descr VARCHAR(2),
  IN Input2_request_descr VARCHAR(2),
  /* Par */
  IN Par INTEGER,
  /* Desc */
  IN Desc VARCHAR (2)
  /* status code */
  INOUT Status CHAR(5)
) DETERMINISTIC
READS SQL DATA
```

Note that Ptf_describe must be deterministic, even if Ptf is not. The DBMS tool can copy the SQL-data access READS SQL DATA from the declaration for the PTF.

The parameter list for the Ptf_start is

```
CREATE PROCEDURE Ptf_start (
  /* private data */
  INOUT Priv INTEGER,
  /* Input1 (row semantics) */
  /* Input2 (set semantics) */
  IN Input2_pby_descr VARCHAR(2),
  IN Input2_order_descr VARCHAR(2),
  /* Par */
  IN Par INTEGER,
  /* Desc */
  IN Desc VARCHAR (2)
  /* status code */
  INOUT Status CHAR(5)
) NOT DETERMINISTIC
READS SQL DATA
```

Ptf was declared as NOT DETERMINISTIC; therefore, at least one of its component procedures is not deterministic. The DBMS tool can just assume that all PTF component procedures of Ptf are non-deterministic. If Ptf was declared as DETERMINISTIC, then all component procedures must be deterministic as well.

For Ptf_fulfill, there are also parameters for the cursor names, so the signature looks like this:

```
CREATE FUNCTION Ptf_fulfill (
  /* private data */
  INOUT Priv INTEGER,
  /* Input1 */
  IN Input1_cursor_descr VARCHAR(2),
  IN Input1_cursor_name VARCHAR(2),
  /* Input2 */
  IN Input2_cursor_descr VARCHAR(2),
  IN Input2_pby_descr VARCHAR(2),
  IN Input2_order_descr VARCHAR(2),
  IN Input2_cursor_name VARCHAR(2),
```

```
  /* Par */
  IN Par INTEGER,
  /* Desc */
  IN Desc VARCHAR (2)
  /* status code */
  INOUT Status CHAR(5)
) NOT DETERMINISTIC
READS SQL DATA
```

The parameter list for the Ptf_finish is:

```
CREATE PROCEDURE Ptf_finish (
  /* private data */
  IN Priv INTEGER,
  /* Input1 (row semantics) */
  /* Input2 (set semantics) */
  IN Input2_pby_descr VARCHAR(2),
  IN Input2_order_descr VARCHAR(2),
  /* Par */
  IN Par INTEGER,
  /* Desc */
  IN Desc VARCHAR (2)
  /* status code */
  INOUT Status CHAR(5)
) NOT DETERMINISTIC
READS SQL DATA
```

The only difference in the parameter lists of Ptf_start and Ptf_finish is that the private parameters are passed as IN to Ptf_finish, since there is no later stage to read the private parameters.

6 Data definition language

Primary audience: DBMS developer and PTF author.

This topic has largely been covered implicitly in the preceding sections Subclause 5.1, “Functional specification”, and Subclause 5.2, “Design specification”. Here we summarize the DDL relevant to PTFs.

6.1 PTF creation

The following shows the parts of the BNF from [ISO9075-2], Subclause 11.60, “<SQL-invoked routine>”, that are relevant to the declaration of a PTF. BNF productions that do not apply to PTF declaration have been omitted.

```
<schema function> ::=
  CREATE <SQL-invoked function>

<SQL-invoked function> ::=
  <function specification> <routine body>

<SQL parameter declaration list> ::=
  <left paren>
      [ <SQL parameter declaration>
      [ { <comma> <SQL parameter declaration> }... ] ]
  <right paren>

<SQL parameter declaration> ::=
  [ <SQL parameter name> ]
  <parameter type>
  [ DEFAULT <parameter default> ]

<parameter default> ::=
    <value expression>
  | <contextually typed value specification>
  | <descriptor value constructor>

<parameter type> ::=
    <data type> [ <locator indication> ]
  | <generic table parameter type>
  | <descriptor parameter type>

<generic table parameter type> ::=
  TABLE [ <pass through option> ]
      [ <generic table semantics> ]

<pass through option> ::=
    PASS THROUGH
  | NO PASS THROUGH

<generic table semantics> ::=
    WITH ROW SEMANTICS
```

```
  | WITH SET SEMANTICS [ <generic table pruning> ]

<generic table pruning> ::=
    PRUNE ON EMPTY
  | KEEP ON EMPTY

<descriptor parameter type> ::=
  DESCRIPTOR

<function specification> ::=
  FUNCTION <schema qualified routine name>
      <SQL parameter declaration list>
      <returns clause>
      <routine characteristics>

<routine characteristics> ::=
  [ <routine characteristic>... ]

<routine characteristic> ::=
    SPECIFIC <specific name>
  | <deterministic characteristic>
  | <SQL-data access indication>

<returns clause> ::=
  RETURNS <returns type>

<returns type> ::=
  <returns table type>

<returns table type> ::=
    TABLE [ <table function column list> ]
  | ONLY PASS THROUGH

<routine body> ::=
  <polymorphic table function body>

<polymorphic table function body> ::=
  [ <PTF private parameters> ] [ DESCRIBE WITH <PTF describe component procedure> ] [ START
  WITH <PTF start component procedure> ] FULFILL WITH <PTF fulfill component procedure> [
  FINISH WITH <PTF finish component procedure> ]

<PTF private parameters> ::=
  PRIVATE [ DATA ] <SQL parameter declaration list>

<PTF describe component procedure> ::=
  <specific routine designator>

<PTF start component procedure> ::=
  <specific routine designator>

<PTF fulfill component procedure> ::=
  <specific routine designator>

<PTF finish component procedure> ::=
  <specific routine designator>
```

The defaults for the optional syntax are:

1) <pass through option> defaults to NO PASS THROUGH.

2) <generic table semantics> defaults to WITH SET SEMANTICS.

3) <generic table pruning> defaults to KEEP ON EMPTY.

4) <specific name> defaults to an implementation-dependent specific name.

5) <deterministic characteristic> defaults to NOT DETERMINISTIC.

6) <SQL-data access indication> defaults to READS SQL DATA if there is an input table; otherwise, the default is CONTAINS SQL.

6.2 PTF component procedures

A PTF requires one to four PTF component procedures. These are declared as conventional SQL-invoked procedures. This means that the PTF is dependent on its component procedures. Although the PTF author will start the specification and design process with the PTF, the implementation will start with the PTF component procedures, which should be declared before the PTF itself is. The order of creation and declaration is PTF component procedures first, then the PTF itself. This is analogous to a view: the underlying tables must be created first, then the view.

The syntax to create a PTF component procedure is the same as for any other SQL-invoked procedure. There is no special syntax in the declaration of a PTF component procedure that announces that it is intended for use by a PTF.

The PTF component procedure does not have parameters for generic tables or descriptor areas. Instead, the PTF component procedure has parameters for the PTF extended names of cursors and descriptor areas. This is explained in Subclause 5.2.4, “Component procedure signatures”.

There are syntactic restrictions on the characteristics of PTF component procedures, as explained in Subclause 5.2.3, “Routine characteristics of the component procedures”. Since there is no syntax indicating that an SQL-invoked procedure will be subsequently used as a PTF component procedure, these restrictions are not syntax checked when the PTF component procedure is created. Instead, they are checked when the PTF is created.

6.3 Altering PTF component procedures and PTFs

The SQL standard has very limited capabilities to alter SQL-invoked routines in general. Only external routines may be altered, not routines written in SQL. An external routine can be altered only if there are no dependencies on it. Since a PTF is dependent on its component procedures, this means that once an SQL-invoked procedure is named as a component procedure of a PTF, it can no longer be altered. The only recourse is to drop the PTF before altering its components, after which the PTF may be re-created.

The SQL standard has no syntax to alter a PTF itself.

6.4 Dropping a PTF and its component procedures

Because of the dependency relationship, a PTF must be dropped before altering or dropping its component procedures.

7 Implementation

Primary audience: PTF author

After specifying the PTF, the PTF author will implement it, that is, supply the programming logic for the routine bodies of the PTF component procedures. To do that, the PTF author will need to understand the following:

1) How the DBMS will process and invoke of the PTF.

2) How to read and write PTF descriptor areas.

3) How to read the input PTF cursor.

4) How to output a row.

Item 1) is presented in Clause 4, "PTF processing model", Clause 9, "Compilation", and Clause 11, "Execution". In this Subclause, we talk about the remaining topics in a generic fashion. This Subclause describes the infrastructure that the DBMS provides so that the PTF author can implement the PTF.

Implementation of a PTF depends upon the interface provided by the DBMS. [ISO9075-2], specifies an interface as a specification device. This interface is optional, and is governed by conformance Feature B208, "PTF component procedure interface" (and the feature that it implies, Feature B209, "PTF extended names"). If the DBMS provides a different interface, then the implementation of a PTF will need to use that interface rather than the one shown in this Technical Report.

7.1 PTF descriptor areas

PTFs use PTF descriptor areas for the following purposes:

1) To describe the row type of input tables.

2) To describe the partitioning and ordering of input tables.

3) To describe the row type of the result.

4) To receive descriptions of row types supplied by the query author.

5) (Optionally) to receive rows of the input tables.

6) To pass output rows from the PTF back to the DBMS.

See Subclause 4.8, "Flow of row types", for a list of the various input and output row types.

PTF descriptor areas are a kind of SQL descriptor area. This Subclause discusses only the features of SQL descriptor areas that pertain to PTF descriptor areas.

A PTF descriptor area consists of a header and zero or more SQL item descriptor areas. SQL item descriptor areas within a PTF descriptor area are numbered sequentially beginning at 1. A PTF descriptor area can be visualized like this:

Table 5 — PTF descriptor area

Descriptor area header
SQL item descriptor area 1
SQL item descriptor area 2
SQL item descriptor area 3
. . .

The header is used to provide information about the PTF descriptor area as a whole. An SQL item descriptor area presents information about a single column of a table, a partitioning, or an ordering.

The header and each SQL item descriptor area have several named components. There is no prescribed data structure for the header or SQL item descriptor area (nothing like a C `struct`). Instead, each component of a header or SQL item descriptor area is referenced by a keyword, essentially the name of the component.

The PTF can get the value of one or more components using a GET DESCRIPTOR command. The PTF can set the value of components in the header or SQL item descriptor areas using a SET DESCRIPTOR command. There is also a COPY DESCRIPTOR command that may be used to copy an SQL item descriptor area from a source PTF descriptor area to a destination PTF descriptor area.

Because SQL has constructed types, which permit arbitrary nesting of data types, the SQL item descriptor areas can form a tree. The tree is flattened as pictured above, by walking the tree from root to leaves and left to right, emitting an SQL item descriptor area whenever a node is first entered. The LEVEL component indicates how deeply nested an SQL item descriptor area is from the root. Scanning the list of SQL item descriptor areas from first to last, if LEVEL goes up by 1, that means to create a child node; if LEVEL remains the same, that means to create a sibling node; if LEVEL goes down by 1, that means the previous set of children is done. LEVEL is 0 in SQL item descriptor areas that are not subordinate to a constructed type. The header component COUNT is the total number of SQL item descriptor areas (including subordinate ones), while the TOP_LEVEL_COUNT is the number of columns.

7.1.1 PTF descriptor area header

A PTF descriptor area header has the following components:

Table 6 — PTF descriptor area header

Name	Data type	Use
TOP_LEVEL_COUNT	integer	Number of columns described by the SQL item descriptor areas
COUNT	integer	Number of SQL item descriptor areas. This is equal to TOP_LEVEL_COUNT if there are no constructed types (collections or rows).

7.1.2 SQL item descriptor areas for row types

An SQL item descriptor area describes a column of a table, a partitioning, or an ordering. When used to describe a column of a table, it consists of a number of components with three purposes:

1) Name the column.

2) Define a data type.

3) Pass a value of that data type to or from the DBMS.

When used to describe a column of a partitioning, the only purpose is to name the column.

When used to describe a column of an ordering, the components describe the following:

1) The name of the column.

2) Whether the column is sorted in ascending or descending order.

3) Whether nulls are sorted first or last.

We consider the components relevant to a column of a table first.

The column name is in the NAME component. This is a variable length character string that is case sensitive and holds the column name after any case conversion or escaping have been applied. Thus, a column name that was created with the <identifier> Col is represented as 'COL', whereas "Col" is represented as 'Col'.

The most important component for defining a type is called TYPE, which is a small integer with predefined codes for the various data types (for example, CHARACTER is 1, INTEGER is 4, *etc.*). The DBMS may define additional data types using negative integers, which this Technical Report cannot enumerate. Many data types are also qualified by additional components. The complete set of relevant components for each SQL type is shown in the following table:

Table 7 — Relevant SQL item descriptor components

Data type	Relevant SQL item descriptor components
CHAR(*n*) CHARACTER SET *cat1.sch1.set* [COLLATION *cat2.sch2.coll*]	TYPE = 1 LENGTH = *n* CHARACTER_SET_CATALOG = *cat1* CHARACTER_SET_SCHEMA = *sch1* CHARACTER_SET_NAME = *set* Optionally, a collation may be specified: COLLATION_CATALOG = *cat2* COLLATION_SCHEMA = *sch2* COLLATION_NAME = *coll*

Data type	Relevant SQL item descriptor components
VARCHAR(*n*) CHARACTER SET *cat1.sch1.set* [COLLATION *cat2.sch2.coll*]	TYPE = 12 LENGTH = *n* CHARACTER_SET_CATALOG = *cat1* CHARACTER_SET_SCHEMA = *sch1* CHARACTER_SET_NAME = *set* Optionally, a collation may be specified: COLLATION_CATALOG = *cat2* COLLATION_SCHEMA = *sch2* COLLATION_NAME = *coll*
CLOB(*n*) CHARACTER SET *cat1.sch1.set* [COLLATION *cat2.sch2.coll*]	TYPE = 40 LENGTH = *n* CHARACTER_SET_CATALOG = *cat1* CHARACTER_SET_SCHEMA = *sch1* CHARACTER_SET_NAME = *set* Optionally, a collation may be specified: COLLATION_CATALOG = *cat2* COLLATION_SCHEMA = *sch2* COLLATION_NAME = *coll*
BINARY(*n*)	TYPE = 60 LENGTH = *n*
VARBINARY(*n*)	TYPE = 61 LENGTH = *n*
BLOB(*n*)	TYPE = 30 LENGTH = *n*
NUMERIC(*prec*, *sc*)	TYPE = 2 PRECISION = *prec* SCALE = *sc*
DECIMAL(*prec*, *sc*)	TYPE = 3 PRECISION = *prec* SCALE = *sc*
SMALLINT	TYPE = 5
INTEGER	TYPE = 4
BIGINT	TYPE = 25
FLOAT(*prec*)	TYPE = 6 PRECISION = *prec*
REAL	TYPE = 7
DOUBLE PRECISION	TYPE = 8

Data type	Relevant SQL item descriptor components
DECFLOAT(*prec*)	TYPE = 26 PRECISION = *prec*
BOOLEAN	TYPE = 16
DATE	TYPE = 9 DATETIME_INTERVAL_CODE = 1
TIME(*prec*) WITHOUT TIME ZONE	TYPE = 9 DATETIME_INTERVAL_CODE = 2 PRECISION = *prec*
TIME(*prec*) WITH TIME ZONE	TYPE = 9 DATETIME_INTERVAL_CODE = 4 PRECISION = *prec*
TIMESTAMP(*prec*) WITHOUT TIME ZONE	TYPE = 9 DATETIME_INTERVAL_CODE = 3 PRECISION = *prec*
TIMESTAMP(*prec*) WITH TIME ZONE	TYPE = 9 DATETIME_INTERVAL_CODE = 5 PRECISION = *prec*
INTERVAL { YEAR(*prec*) MONTH(*prec*) DAY(*prec*) HOUR(*prec*) MINUTE(*prec*) }	TYPE = 10 DATETIME_INTERVAL_PRECISION = *prec* The value of DATETIME_INTERVAL_CODE depends on the interval qualifier as follows: YEAR: 3 MONTH: 2 DAY: 3 HOUR: 4 MINUTE: 5
INTERVAL { YEAR(*prec*) TO MONTH DAY(*prec*) TO HOUR DAY(*prec*) TO MINUTE HOUR(*prec*) TO MINUTE }	TYPE = 10 DATETIME_INTERVAL_PRECISION = *prec* The value of DATETIME_INTERVAL_CODE depends on the interval qualifier as follows: YEAR TO MONTH: 7 DAY TO HOUR: 8 DAY TO MINUTE: 9 HOUR TO MINUTE: 11

Data type	Relevant SQL item descriptor components
INTERVAL { DAY(*prec*) TO SECOND(*frac*) HOUR(*prec*) TO SECOND(*frac*) MINUTE(*prec*) TO SECOND(*frac*) SECOND(*prec*, *frac*) }	TYPE = 10 DATETIME_INTERVAL_PRECISION = *prec* The value of DATETIME_INTERVAL_CODE depends on the interval qualifier as follows: DAY TO SECOND: 10 HOUR TO SECOND: 12 MINUTE TO SECOND: 13 SECOND: 6
ROW (*field*$_1$, *field*$_1$, ... *field*$_n$)	TYPE = 19 DEGREE = *n* There are *n* immediately subordinate SQL item descriptors that describe *field*$_1$, *field*$_1$, ... *field*$_n$
user-defined type	TYPE = 17 USER_DEFINED_TYPE_CATALOG = *cat1* USER_DEFINED_TYPE_SCHEMA = *sch1* USER_DEFINED_TYPE_NAME = *ty*
REF (*cat1.sch1.ty*) SCOPE *cat2.sch2.tab*	TYPE = 20 USER_DEFINED_TYPE_CATALOG = *cat1* USER_DEFINED_TYPE_SCHEMA = *sch1* USER_DEFINED_TYPE_NAME = *ty* SCOPE_CATALOG = *cat2* SCOPE_SCHEMA = *sch2* SCOPE_NAME = *tab*
ty ARRAY [*card*]	TYPE = 50 CARDINALITY = *card* There is one immediately subordinate SQL item descriptor area that describes the element type *ty*
ty MULTISET [*card*]	TYPE = 55 CARDINALITY = *card* There is one immediately subordinate SQL item descriptor area that describes the element type *ty*

Constructed types (arrays, multisets, rows) are described using several consecutive SQL item descriptor areas. The first SQL item descriptor area specifies the kind of constructed type, and subsequent SQL item descriptor areas describe the components: the element type of a collection, or the fields of a row type. The LEVEL component is used for bookkeeping to keep track of the depth of nesting. At the top level, LEVEL is 0, and LEVEL is incremented by 1 to descend a level when describing a constructed type.

There are also components to indicate if the described column is nullable, or if it is a member of the primary key or candidate key. These components are not needed for PTFs, and are not considered in this Technical Report.

Each SQL item descriptor area has a component called DATA that can hold the value of the corresponding column, which must be of the type described by the other components. DATA is conceptually similar to a union

in C, since all types can be placed there. Subordinate item descriptor areas are not used to pass elements of a collection or components of a row type; the entire value must be passed at the top level of a constructed type. The DATA component may be used when reading a row of an input table, or when creating an output row.

7.1.3 SQL item descriptor areas for partitioning

The only relevant components when describing a partitioning are LEVEL and NAME. Since type information is not present, there are no subordinate SQL item descriptor areas. Therefore, LEVEL is 0 in every SQL item descriptor area, and in the header, COUNT = TOP_LEVEL_COUNT is the number of partitioning columns.

7.1.4 SQL item descriptor areas for ordering

PTF descriptor areas are also used to describe the ordering of input tables with set semantics. There is no need for type information, so there is no need for subordinate item descriptor areas when describing an ordering. Consequently COUNT and TOP_LEVEL_COUNT in the header are the same value, which is the number of columns in the ORDER BY clause. In the SQL item descriptor areas, four components are used:

— LEVEL (always 0).

— NAME, the name of the column in the ORDER BY clause.

— ORDER_DIRECTION, either +1 for ASC (ascending) or –1 for DESC (descending).

— NULL_PLACEMENT, either +1 for NULLS FIRST or –1 for NULLS LAST.

7.2 PTF extended names

In general, the SQL standard has three namespaces for SQL descriptor areas: the non-extended namespace, the extended namespace, and the PTF namespace. The distinguishing feature of the PTF namespace is that the DBMS assigns the names rather than the PTF author or the query author. Non-extended names and extended names are assigned to SQL descriptor areas using the ALLOCATE DESCRIPTOR command. In contrast, all the SQL descriptor areas discussed in this Technical Report are created automatically by the DBMS (the PTF author does not write an ALLOCATE DESCRIPTOR command for them). The names of these SQL descriptor areas are called PTF extended names and they constitute the PTF namespace. The DBMS assigns unique names within the PTF namespace, and it is these unique names that are passed in descriptor area arguments to the PTF component procedures. When a query has multiple PTF invocations, then each one has its own PTF namespace.

For example, suppose that Input_descr is an input argument to a PTF component procedure containing the name of a PTF descriptor area for the row type of an input table. The value of Input_descr might be 'I1'. This value is assigned by the DBMS in an implementation-dependent manner. The name is meaningless to the PTF and there is no reason for a PTF component procedure to examine the value of this input argument. Instead, the name can simply be passed along in various commands (GET DESCRIPTOR, SET DESCRIPTOR, and COPY DESCRIPTOR), as explained in succeeding subsections.

Even if the DBMS does not support the entire interface described in this Clause, it may choose to support PTF extended names, in which case it can claim conformance to Feature B209, “PTF extended names”, without claiming conformance to Feature B208, “PTF component procedure interface”.

7.3 Reading a PTF descriptor area

PTF descriptor areas are read using the GET DESCRIPTOR command. For example, suppose that Input_descr is an argument that contains the name of a PTF descriptor area. The PTF component procedure might contain:

```
EXEC SQL GET DESCRIPTOR PTF :Input_descr :Items = COUNT;
```

This command gets the COUNT component from the header of the PTF descriptor area in the PTF namespace whose name is given by Input_descr. The result is placed in the host variable items.

If the implementation language is SQL/PSM, the command looks very similar:

```
GET DESCRIPTOR PTF Input_descr items = COUNT;
```

The main difference here is that embedded variables are preceded by a colon, whereas SQL/PSM variables are not.

After getting the number of SQL item descriptors, the PTF component procedure will typically set up a loop to examine all the items. The loop might examine the column names and data types, for example. Let J be a variable used to index into an array, with J=1 for the first column, etc. To obtain column name and data type information on the J-th column, in embedded C the command would be:

```
EXEC SQL GET DESCRIPTOR PTF :Input_descr VALUE :J
    :name[J-1] = NAME,
    :dtype[J-1] = TYPE;
```

The VALUE clause specifies which item descriptor area is desired. Note the use of J-1 to account for 0-relative arrays in C *vs* 1-relative items in SQL descriptor areas. In SQL/PSM the command is:

```
GET DESCRIPTOR PTF Input_descr
    VALUE J
    name[J] = NAME,
    dtype[J] = TYPE;
```

Data types are represented by codes defined in the SQL standard. Depending on the type, additional components of the item descriptor area may be relevant, such as LENGTH, PRECISION, or SCALE. See Subclause 7.1.2, “SQL item descriptor areas for row types”, for more details.

7.4 Writing a PTF descriptor area

The PTF describe component procedure must populate PTF descriptor areas for two purposes:

1) The requested row type: this is essentially just a list of the names of the columns that the PTF wishes to read on the cursor for an input table.

2) The initial result row type: if the CREATE FUNCTION that created the PTF does not declare the proper columns (either through <table function column list> or RETURNS ONLY PASS THROUGH), then the PTF describe component is responsible for describing the names and types of the proper result columns.

For each of these purposes, the DBMS allocates an empty PTF descriptor area in the PTF descriptor namespace and passes the name of the PTF descriptor area to the describe PTF component procedure. Initially, the COUNT

in the PTF descriptor area header will be 0; before returning, the PTF describe component procedure should set this to the number of requested columns or output columns.

In addition, during execution, the PTF component procedures write to the DATA components of the intermediate result row descriptor.

Writing to a PTF descriptor area is one of the most challenging parts of writing a PTF. We will present three different ways from which the PTF author may choose, depending on the complexity of the PTF.

7.4.1 Using DESCRIBE to populate a PTF descriptor area

The DESCRIBE command may be used to populate the initial result row type PTF descriptor area. To start with an unlikely but simple example, suppose that the result is always a single column called V of type DOUBLE PRECISION. In that case, the PTF describe component procedure might use a DESCRIBE such as the following:

```
EXEC SQL PREPARE Stmt
    FROM 'SELECT CAST (0 AS DOUBLE PRECISION) AS V
          FROM INFORMATION_SCHEMA.TABLES';
EXEC SQL DESCRIBE Stmt
    USING DESCRIPTOR PTF :Initial_result_row;
```

The precise FROM clause in the prepared statement is irrelevant. This example used a table known to exist and be readable by PUBLIC in the FROM clause, so that the PREPARE statement is guaranteed to succeed. Similarly, the precise column definition in the SELECT list does not matter. The point is to prepare any statement with a single column named V of type DOUBLE PRECISION. Using a CAST and a column alias is a clear and certain way to do this.

This example is unrealistic because the PTF author has a better way to declare that the PTF always returns a single column V of type DOUBLE PRECISION, by simply declaring it in the CREATE FUNCTION statement. However, the example can be generalized by making it more dynamic. For example, suppose that the variable IsDouble is a boolean variable that is *True* if the column should be DOUBLE PRECISION; otherwise, the column should be INTEGER. In that case, one might write the following in SQL/PSM:

```
SET String =    'SELECT CAST (0 AS '
             ||
                    CASE WHEN IsDouble THEN
                      'DOUBLE PRECISION'
                    ELSE
                      'INTEGER'
                    END
             ||
                    ') AS V
                FROM INFORMATION_SCHEMA.TABLES';
PREPARE Stmt FROM String;
DESCRIBE Stmt USING DESCRIPTOR PTF Initial_result_row;
```

7.4.2 Using SET DESCRIPTOR to populate a PTF descriptor area

To use SET DESCRIPTOR, first item descriptor area(s) must be added to the empty PTF descriptor area. This can be done by setting COUNT to a non-zero value. The command in embedded SQL might look like this:

```
EXEC SQL SET DESCRIPTOR PTF :Initial_result_row
               COUNT = :ncol;
```

or, if the implementation language is SQL/PSM, then this:

```
SET DESCRIPTOR PTF Initial_result_row
      COUNT = ncol;
```

After item descriptor area(s) have been added to the PTF descriptor area, the various components must be set appropriately. Each item descriptor area requires a LEVEL, NAME, and TYPE, and possibly other components, as explained in Subclause 7.1.2, “SQL item descriptor areas for row types”.

For example, to specify that the column whose ordinal position in the row is given by Ncol and whose name given by Colname and is of type VARCHAR with maximum length 100, this could be used:

```
EXEC SQL SET DESCRIPTOR PTF :Initial_result_row
               VALUE :Ncol
               LEVEL = 0,
               NAME = :Colname,
               TYPE = 12,
               LENGTH = 100;
```

Here, 12 is the code for VARCHAR type.

It may be convenient to build the result row type descriptor in a loop. The loop might increment variable Ncol by 1 on each iteration, then set COUNT in the header to Ncol to allocate another SQL item desciptor, and then set the column name and type information in the item descriptor area indexed by Ncol.

7.4.3 Using COPY DESCRIPTOR to populate a PTF descriptor area

COPY DESCRIPTOR is a command to copy either an entire PTF area descriptor, or just a single SQL item descriptor area. Here, we examine situations where each of these can be useful.

Sometimes, the result row type is the same as either an input table (for example, TopNplus in our running examples) or is simply provided by the query author (ExecR in our running examples). In that case, the most convenient way to populate the result row type is to copy it in its entirety from some input PTF descriptor area. To copy an entire PTF area descriptor, the PTF author might use something like the following:

```
EXEC SQL COPY DESCRIPTOR PTF :Input_table_descr
                     TO PTF :Result_row_type;
```

The preceding simply copies the entire PTF descriptor from Input_table_descr to Result_row_type.

In other circumstances, it may be that only selected SQL item descriptors should be copied. For example, perhaps an output column has the same column name and/or type information as a column of an input table. In that case, it might be convenient to just copy the column name and/or type information from the input table's PTF descriptor area to the result's PTF descriptor area. For example, in embedded SQL,

```
EXEC SQL COPY DESCRIPTOR
             PTF :Source_descr
                 VALUE :c1 (NAME, TYPE)
          TO PTF :Dest_descr
                 VALUE :c2;
```


This copies from the PTF descriptor area identified by Source_descr in the item identified by c1 to the PTF descriptor area identified by Dest_descr, placing the information in the item descriptor area identified by c2. This example copies the NAME and TYPE components from the source to the destination. When copying TYPE, any other components that are required to complete the type specification are also copied.

This technique can also be used in conjunction with SET DESCRIPTOR as explained above in Subclause 7.4.2, "Using SET DESCRIPTOR to populate a PTF descriptor area". For example, suppose that the first column of the output should be the same as the first column of the input, except that the column name is always X. In that case, one might use COPY DESCRIPTOR to copy the type information and SET DESCRIPTOR to set the column name, like this:

```
EXEC SQL COPY DESCRIPTOR
             PTF :Source_descr
               VALUE 1 TYPE
          TO PTF :Dest_descr
               VALUE 1;
EXEC SQL SET DESCRIPTOR PTF :Dest_descr
                        VALUE 1 NAME = 'X';
```

7.5 Reading a PTF input cursor

For each input table IT, during execution on a virtual processor VP, the DBMS will create a cursor that reads the rows of the partition of IT that is assigned to VP. The DBMS will give this cursor a name in the PTF namespace for cursors. The PTF fulfill component procedure will receive the name of the cursor in an input argument. The cursor is already open, so the PTF fulfill component procedure can simply issue FETCH commands to read the cursor. In addition, the row type of the input cursor is described by a PTF descriptor area whose name is in another input argument. The PTF fulfill component procedure can simply FETCH from the input cursor into the PTF descriptor area for that input table.

For example, suppose the input cursor name is passed in the parameter Input_cursor and the name of the PTF descriptor area for the row type is passed in the parameter Input_row_descr. Then, using embedded SQL, the PTF fulfill component procedure might use this command:

```
EXEC SQL FETCH FROM PTF :Input_cursor
               INTO DESCRIPTOR PTF :Input_row_descr;
```

or if the implementation language is SQL/PSM,

```
FETCH FROM PTF Input_cursor
      INTO DESCRIPTOR PTF Input_row_descr;
```

After fetching a row into a PTF descriptor area, the PTF fulfill component procedure will want to access the data in the columns of the row. This data is in a component of the item descriptor area called DATA. Unlike other components of item descriptor areas, DATA has no fixed type; instead, its type is simply the type of the column, which is of course described by other components of the same item descriptor area.

For example, suppose that Var is a variable of an appropriate type to receive the value of a column found in the item descriptor area indicated by Colno. Then, the value of that column can be obtained in Var using this command in embedded SQL:

```
EXEC SQL GET DESCRIPTOR PTF :Input_row_descr VALUE :Colno
                            :Var = DATA;
```

or, using SQL/PSM:

```
GET DESCRIPTOR PTF Input_row_descr VALUE Colno
                   Var = DATA;
```

If the input data can be of several types, then it will generally be necessary to set up conditional logic that tests the TYPE of a column so that the DATA can be assigned to an appropriately typed variable.

Typically the PTF fulfill component procedure will fetch rows from the input cursor until it reaches the end of the cursor. At that point, the PTF fulfill component procedure does not need to close the cursor; this will be handled automatically by the DBMS when the PTF fulfill component procedure returns control to the DBMS.

7.6 Outputting a row

During run-time on a virtual processor VP, the PTF start, fulfill, and/or finish component procedures need to generate and output row(s) for the result. Outputting a row is a two-step process:

1) First, the output row is populated by setting the DATA component of the SQL item descriptor areas of the result row descriptor.

2) Second, a PIPE ROW command is used to send the row to the DBMS as output.

For example, suppose the output row has two columns. Suppose that the value of the first column has been computed in variable X and the value of the second column has been computed in variable Y. Suppose that the name of the result row descriptor is in the argument Intermediate_result_row. Then, these commands could be used to populate the output row:

```
EXEC SQL SET DESCRIPTOR PTF :Intermediate_result_row
                            VALUE 1 DATA = :X;
EXEC SQL SET DESCRIPTOR PTF :Intermediate_result_row
                            VALUE 2 DATA = :Y;
```

It is also possible to use COPY DESCRIPTOR to transfer DATA from an input row to the result row. If the input row has precisely the same row type as the result row (corresponding column names and types match), then COPY DESCRIPTOR without VALUE can be used, like this:

```
EXEC SQL COPY DESCRIPTOR PTF :Input_table_descr (DATA)
                       TO PTF :Intermediate_result_row;
```

To copy a single column from an input row to the result row, the VALUE clause is needed:

```
EXEC SQL COPY DESCRIPTOR PTF :Input_table_descr
                              VALUE :InColNo (DATA)
                       TO PTF :Intermediate_result_row
                              VALUE :OutColNo;
```

This technique can be used with pass-through surrogate values. For any input table with pass-through columns, the pass-through input surrogate column is the last one in the cursor row type. The DBMS will give it a distinctive implementation-dependent name. The corresponding pass-through output surrogate column can be found in the intermediate result row descriptor by searching for the matching name. Having located the surrogates in the input and output rows, COPY DESCRIPTOR can be used to copy the surrogate value from the input row to the output row.

Once the output row has been populated, the PTF start, fulfill, or finish component procedure can write this row to output using this command:

```
EXEC SQL PIPE ROW PTF :Intermediate_result_row;
```

If there is more than one output row, the result row descriptor can be reused for each output row, for example, in a loop.

(Blank page)

8 Invocation

Primary audience: DBMS developer and query author.

Once the PTF and its component procedures have been written, it is possible to write an invocation. During development, the PTF author will write test invocations; in deployment, the query author will write invocations. Examples of the invocation syntax have already been presented in Subclause 3.2, "Motivating examples". Here, we present the invocation syntax formally.

8.1 <table primary>

A PTF is invoked only in a FROM clause, as a kind of <table primary>. There are many kinds of <table primary>, the most common being a table name. For PTFs, the relevant syntax begins as follows:

```
<table primary> ::=
    ...
  | <PTF derived table>
      [ <correlation or recognition> ]
  | ...
```

Thus a PTF invocation consists of a <PTF derived table> and sometimes a <correlation or recognition>. The BNF above suggests that the query author can choose to have the <correlation or recognition> or not, but in fact its presence or absence is dictated by the DDL that created the PTF, as we shall see in Subclause 8.3, "Proper result correlation name and proper result column naming".

8.2 <PTF derived table>

```
<PTF derived table> ::=
  TABLE <left paren> <routine invocation> <right paren>
```

A <PTF derived table> consists of the keyword TABLE and a parenthesized <routine invocation> that invokes the PTF. This is the same syntax that is used to invoke a monomorphic table function.

8.3 Proper result correlation name and proper result column naming

```
<correlation or recognition> ::=
  [ AS ] <correlation name>
      [ <parenthesized derived column list> ]
```

The correlation name is used to qualify the proper result columns of the PTF, that is, the columns that the PTF itself generates. We will call this the proper result correlation name to distinguish it from the table argument correlation names that may be associated with input tables (see Subclause 8.6, "<table argument proper>", and

Subclause 8.7, “Table argument correlation name”, regarding table argument correlation names). Here is a skeleton example:

```
FROM TABLE (Ptf (...) ) AS P
```

Optionally, you can rename the proper result columns. In the preceding example, suppose that there is one proper result column, named Score, and you want to rename it to Val:

```
FROM TABLE (Ptf (...) ) AS P (Val)
```

If the PTF is declared RETURNS ONLY PASS THROUGH, then there are no proper result columns and hence the proper result correlation name (and column renaming) is forbidden. Otherwise, the proper result correlation name is required, and the column renaming is optional.

8.4 <routine invocation>

<routine invocation> is enhanced to support table and descriptor arguments (for PTF only, of course):

```
<routine invocation> ::=
  <routine name> <SQL argument list>

<routine name> ::=
  [ <schema name> <period> ] <qualified identifier>
```

The <routine invocation> must, of course, invoke a PTF. The usual name resolution rules of the SQL standard apply, including the use of the SQL path and the precise argument list to determine the specific PTF to invoke. The complete rules for subject routine resolution are complex and outside the scope of this Technical Report. The PTF author can avoid most of this complexity by avoiding duplicate PTF names. Even though the standard permits overloading of SQL-invoked routines, it is better to use optional parameters in a single PTF definition, rather than defining multiple PTFs of the same name and different parameter lists. The query author may do well to use fully qualified schema names when invoking a PTF, though this Technical Report has not done so in its examples.

Note that the query author does not invoke the PTF component procedures explicitly, these being hidden within the PTF. The query author only needs EXECUTE privilege on the PTF, not on the PTF component procedures.

```
<SQL argument list> ::=
  <left paren> [ <SQL argument>
      [ { <comma> <SQL argument> }... ] ]
      [ <copartition clause> ] <right paren>
```

The optional <copartition clause> is used in copartitioning and will be presented later in Subclause 8.13, “Copartitioning”.

```
<SQL argument> ::=
    <value expression>
  | <generalized expression>
  | <target specification>
  | <contextually typed value specification>
  | <named argument specification>
  | <table argument>
  | <descriptor argument>

<named argument specification> ::=
```



```
  <SQL parameter name> <named argument assignment token>
      <named argument SQL argument>

<named argument SQL argument> ::=
    <value expression>
  | <target specification>
  | <contextually typed value specification>
  | <table argument>
  | <descriptor argument>
```

Input values can be passed to a PTF either positionally or by parameter names (named arguments). There are two kinds of arguments that are allowed only in PTF invocations: <table argument> and <descriptor argument>. As their names imply, a <table argument> is used to pass an input table, and a <descriptor argument> is used to pass a descriptor to a PTF.

All of the examples in this Technical Report use named arguments as a “best practice” from a readability standpoint; however, positional argument lists are also permitted. Optional arguments may be omitted, in which case the default is taken.

8.5 <table argument>

Now we focus on <table argument>:

```
<table argument> ::=
  <table argument proper>
      [ [ AS ] <table argument correlation name>
      [ <table argument parenthesized derived column list> ] ]
      [ <table argument partitioning> ]
      [ <table argument pruning> ]
      [ <table argument ordering> ]


<table argument correlation name> ::=
  <correlation name>

<table argument parenthesized derived column list> ::=
  <parenthesized derived column list>
```

That is, a <table argument> consists of up to six components:

1) The <table argument proper>, which specifies the input table.

2) An optional <correlation name> for the input table.

3) If there is a <correlation name>, then optional column renaming of the input table.

4) An optional partitioning (set semantics tables only).

5) An optional pruning (set semantics tables only).

6) An optional ordering (set semantics tables only).

8.6 <table argument proper>

Now let's look at <table argument proper>:

```
<table argument proper> ::=
    TABLE <left paren> <table or query name> <right paren>
  | TABLE <table subquery>
  | <routine invocation>
```

Thus there are three ways to specify a table: <table or query name>, <table subquery>, or <routine invocation> (nested table function invocation). The next three subsections consider each of these in turn.

8.6.1 <table or query name>

This has two subcases:

1) A <table name>, which of course might be schema qualified and might be a DDL view name. Using the syntax presented so far, one could write any of the following:

   ```
   FROM TABLE (Ptf ( TABLE (Emp) ) ) AS P

   FROM TABLE (Ptf ( TABLE (My.Emp) ) ) AS P

   FROM TABLE (Ptf ( TABLE (Emp) AS E ) ) AS P

   FROM TABLE (Ptf ( TABLE (Emp) AS E(E1,E2,E3) ) ) AS P
   ```

   In the first two, the default range variable is Emp or My.Emp. Range variables do not matter with the syntax presented so far, but they are used to reference input tables in <copartition clause> (presented later in Subclause 8.13, "Copartitioning"), and also outside of the PTF invocation to reference the partitioning columns (if the table has set semantics) or any column of the input table (if the table has pass-through columns).

2) A <query name>, which is the name of an in-line view declared in the WITH clause. If Qn is a <query name>, then one could write any of the following:

   ```
   FROM TABLE (Ptf ( TABLE (Qn) ) ) AS P

   FROM TABLE (Ptf ( TABLE (Qn) AS E ) ) AS P

   FROM TABLE (Ptf ( TABLE (Qn) AS E(E1,E2,E3) ) ) AS P
   ```

   In the first example above, the default range variable is Qn.

8.6.2 <table subquery>

Note that <table subquery> is <left paren> <query expression> <right paren>, so this case also has parentheses. This permits the following example:

```
FROM TABLE (Ptf ( TABLE ( SELECT * FROM Emp ) ) ) AS P
```


There is no default range variable in this case; it is implementation-dependent and unknowable to the query author. The following example provides an explicit correlation name:

```
FROM TABLE (Ptf ( TABLE ( SELECT * FROM Emp ) AS V ) ) AS P
```

8.6.3 Nested table function invocation

A <routine invocation> used as a <table argument proper> must invoke a table function, either monomorphic or polymorphic. No TABLE operator is required (or permitted) in this case because the function's return type is a table. There is no default correlation name, but one can be provided explicitly. Some examples:

```
FROM TABLE (Reduce ( Map (...) ) ) AS P

FROM TABLE (Reduce ( Map (...) AS M ) ) AS P
```

In the preceding examples, Reduce is a polymorphic table function, and Map is invoked as a table argument of Reduce. Map may be either monomorphic or polymorphic table function. Note that a <routine invocation> used as a table argument is necessarily a nested table function invocation.

If the nested table function is monomorphic, then the correlation name qualifies all result columns of the nested table function. If the nested table function is polymorphic, then the correlation name qualifies only the proper result columns; any pass-through or partitioning columns are qualified by the appropriate range variables established within the nested <routine invocation>.

For example, suppose that Map is a polymorphic table function that has one table argument, which has pass-through columns. Consider the following invocation:

```
FROM TABLE ( Reduce
             ( Map ( TABLE ( Emp AS E ) ) AS M
                       PARTITION BY E.E1, M.M1 )
) AS P
```

Then E qualifies the pass-through columns of Emp, whereas M qualifies the proper result columns of Map. The result of Map in the preceding example is partitioned on E.E1 and M.M1.

8.7 Table argument correlation name

An optional correlation name for a table argument may be supplied after the <table argument proper>; examples have already been provided above. In the absence of a table argument correlation name, a <table or query name> provides a default range variable to reference the input table; the other kinds of <table argument proper> do not have default range variables. A range variable may be used for the following purposes:

1) For use in a <copartition clause>, if any.

2) To qualify column names in <table argument partitioning> (see Subclause 8.10, “Partitioning”) or <table argument ordering> (see Subclause 8.12, “Ordering”).

3) If the input table has set semantics, then its correlation name may be used to reference the partitioning columns later in the query (“later” means in any subsequent lateral joins in the FROM clause, as well as the WHERE, GROUP BY, and HAVING clauses, and the SELECT list).

4) If the input table has pass-through columns, then its range variable may be used to reference all columns of the input table later in the query.

8.8 Table argument column renaming

A table argument correlation name may be followed by an optional parenthesized list of column names, used to rename the columns of the input table. If the table argument is a <routine invocation> that invokes a polymorphic table function, then this only renames the proper result columns of the nested PTF invocation. If columns are renamed, those new names are the ones to reference in the partitioning and ordering clauses. The new names are also the column names that the PTF will see; this could be important if the PTF author has designed the PTF to look for specific column names in the input.

8.9 Range variables and column renaming in nested PTF

When there are nested PTF invocations, range variables and column renaming can occur at many levels. The important thing to note is that a column has only one opportunity to receive a range variable, and this is also its only opportunity to be renamed. This opportunity is the innermost scope in the syntax where a range variable for that column can be determined. Once a column receives its range variable (and optional renaming), it cannot receive a different range variable (or renaming) in an outer scope.

Here is an example:

```
FROM TABLE ( G (F (TABLE (Emp) AS E (Eno))
             AS R(Rno)))
     AS S(Sno)
```

In this example:

— Emp has one column.

— F has one proper result column.

— G has one proper result column.

Then:

— The column of Emp is renamed Eno.

— The proper result column of F is renamed Rno.

— The proper result column of G is renamed Sno.

Thus, every column has one opportunity to be renamed, which is at the place in the syntax where the correlation name for that column can be introduced.

8.10 Partitioning

After the <table argument proper>, there is the optional <table argument partitioning>:


```
<table argument partitioning> ::=
  PARTITION BY <table argument partitioning list>

<table argument partitioning list> ::=
    <column reference>
  | <left paren> [ <column reference>
      [ { <comma> <column reference> }... ] ] <right paren>
```

Thus <table argument partitioning> is PARTITION BY with a list of zero or more columns. The list can always be enclosed in parentheses. If there is only one partitioning column, then the parentheses are optional.

Here are some examples with a single table using PARTITION BY:

```
FROM TABLE (Ptf ( TABLE (My.Emp)
                    PARTITION BY () ) ) AS P

FROM TABLE (Ptf ( TABLE (My.Emp)
                    PARTITION BY Deptno ) ) AS P

FROM TABLE (Ptf ( TABLE (My.Emp)
                    PARTITION BY (Deptno) ) ) AS P

FROM TABLE (Ptf ( TABLE (My.Emp)
                    PARTITION BY (Deptno, Jobclass) ) ) AS P
```

The first example uses () to indicate explicitly that there are no partitioning columns. The second example shows a single partitioning column without parentheses. The third example shows a single partitioning column with parentheses. The fourth example shows a list of two partitioning columns with parentheses.

8.11 Pruning

If Feature B204, “PRUNE WHEN EMPTY”, is supported, then a table with set semantics supports DDL to declare either PRUNE WHEN EMPTY or KEEP WHEN EMPTY. PRUNE WHEN EMPTY means that there is no point in invoking the PTF on an empty partition because the result will be empty. If a table parameter is declared to be KEEP WHEN EMPTY, then the PTF may be capable of producing a result, but the query author might be uninterested in it. So, in that case the query author can ask to prune anyway with this syntax:

```
<table argument pruning> ::=
    PRUNE WHEN EMPTY
  | KEEP WHEN EMPTY
```

KEEP WHEN EMPTY is the default.

If Feature B204, “PRUNE WHEN EMPTY”, is not supported, then the preceding syntax is not available, and the only behavior is the default behavior, KEEP WHEN EMPTY.

8.12 Ordering

Ordering is specified by an ORDER BY clause:

```
<table argument ordering> ::=
```

```
  ORDER BY <table argument ordering list>

<table argument ordering list> ::=
    <table argument ordering column>
  | <left paren> <table argument ordering column>
      [ { <comma> <table argument ordering column> }... ] <right paren>

<table argument ordering column> ::=
  <column reference> [ <ordering specification> ]
      [ <null ordering> ]

<ordering specification> ::=
    ASC
  | DESC

<null ordering> ::=
    NULLS FIRST
  | NULLS LAST
```

Thus <table argument ordering> is ORDER BY with a list of one or more columns. Each column may optionally be sorted in either ascending (ASC) or descending (DESC) direction; the default is ASC. Each column may optionally be be sorted with either nulls first (NULLS FIRST) or nulls last (NULLS LAST); the default is implementation-defined. The list can always be enclosed in parentheses. If there is only one ordering column, then the parentheses are optional. Note that this differs from the ORDER BY clause in some other contexts in that only columns may be sorted (not arbitrary expressions) and the list must be parenthesized if more than one column is listed.

Here are some examples with a single table using ORDER BY:

```
FROM TABLE (Ptf ( TABLE (My.Emp)
                    ORDER BY Empno ) ) AS P

FROM TABLE (Ptf ( TABLE (My.Emp)
                    ORDER BY Empno DESC ) ) AS P

FROM TABLE (Ptf ( TABLE (My.Emp)
                    ORDER BY (Empno DESC) ) ) AS P

FROM TABLE (Ptf ( TABLE (My.Emp)
                    ORDER BY (Deptno ASC, Jobclass DESC) ) )
            AS P
```

8.13 Copartitioning

If the DBMS supports Feature B202, “PTF Copartitioning”, then a PTF invocation may specify copartitioning with the following syntax:

```
<copartition clause> ::=
  COPARTITION <copartition list>

<copartition list> ::=
  <copartition specification>
      [ { <comma> <copartition specification> }... ]

<copartition specification> ::=
```



```
  <left paren> <range variable> <comma> <range variable>
      [ { <comma> <range variable> }... ]

<range variable> ::=
    <table name>
  | <correlation name>
```

A <copartition clause> is used if there are multiple partitioned tables and copartitioning is desired. By default, if there are multiple partitioned tables, then the cross product of the partitions is formed to determine the virtual processors. Copartitioning is an alternate way of determining virtual processors, in which a full outer equijoin of the partitioning keys is used to associate partitions on a virtual processor. (Right, left and inner joins can also be obtained depending on whether any of the partitioned tables specify PRUNE WHEN EMPTY in either the DDL or the query.) Copartitioning is explained in Subclause 12.7.9, "Virtual processors for Similarity".

A <copartition clause> has a list of <copartition specification>s. Each <copartition specification> is a parenthesized list of input tables to be copartitioned. Each input table is referenced by its range variable (correlation name, if any, otherwise the table name). Each input table listed in a <copartition specification> must be partitioned. They must all have the same number of partitioning columns, and corresponding partitioning columns must be comparable.

If there is more than one <copartition specification>, then the cross product is formed between the copartitionings. More than one <copartition specification> requires Feature B203, "More than one copartition specification".

8.14 Cross products of partitions

If there is more than one partitioned input table with set semantics, and they are not all copartitioned together in a single <copartition specification>, then execution of the PTF invocation will require the DBMS to form cross products of partitions. (See Subclause 11.1, "Partitions and virtual processors", for more information about the formation of partitions.) The DBMS can choose not to support cross products of partitions, with syntactic restrictions such as the following:

— Permit at most one table parameter (that is, do not support Feature B201, "More than one PTF generic table parameter").

— Permit at most one table parameter with set semantics. In this case, the DBMS will not support Feature B202, "PTF Copartitioning", since copartitioning is only possible if there are at least two input tables with set semantics.

— Permit more than one table parameter with set semantics, but allow at most one of them to be partitioned. In this case the DBMS will not support Feature B202, "PTF Copartitioning".

— Permit more than one table parameter with set semantics, and allow them all to be partitioned, but require that if there are at least two partitioned input tables, then all partitioned input tables must be listed in a single <copartition specification>.

If the DBMS supports cross products of partitions, then the DBMS can claim conformance to Feature B207, "Cross products of partitionings".

8.15 <descriptor argument>

```
<descriptor argument> ::=
  <descriptor value constructor>

<descriptor value constructor> ::=
  DESCRIPTOR <left paren> <descriptor column list> <right paren>

<descriptor column list> ::=
  <descriptor column specification>
      [ { <comma> <descriptor column specification> }... ]

<descriptor column specification> ::=
  <column name> [ <data type> ]
```

A <descriptor argument> is the keyword DESCRIPTOR followed by a parenthesized list of column names; each column name may optionally have a data type. If every column name has a data type, then the descriptor describes a row type. In the examples, CSVreader and Pivot use descriptor arguments that are just lists of column names; ExecR is an example that uses a descriptor to pass a complete row type.

9 Compilation

With an invocation written, it is time to compile the invocation. If you are the query author, this step corresponds to PREPARE in dynamic SQL. If you are using an embedded language preprocessor, then query compilation may occur when you compile the embedded program. Also, if a PTF is invoked in a DDL object, such as a view definition or the body of an SQL-invoked routine, then the invocation may be compiled once and executed many times. Using an interactive SQL interface, the query author is not aware of query compilation as a separate step from query execution, but the DBMS and the PTF do perceive query compilation and query execution as two separate steps. This section will talk about query compilation.

9.1 Calling the describe component procedure

Primary audience: DBMS developer

To compile a PTF invocation, the DBMS will invoke the PTF describe component procedure. The DBMS must do the following:

1) Create a private data area for the invocation. The private data area consists of one variable for each item in the PRIVATE declaration of the CREATE FUNCTION statement that created the PTF.

2) Create PTF descriptor areas for the input tables. Each input table has a full row type descriptor and a requested row type descriptor. The full row type descriptor describes SELECT * from the input table. The requested row type descriptor is empty so that the describe component procedure can populate it. For an input table with set semantics, two additional PTF descriptor areas are required, one for the partitioning and one for the ordering.

3) Create the PTF descriptor area for the initial result row type. If the PTF was created with a known list of proper result columns using <table function column list>, then the initial result row type descriptor describes the proper result columns specified in DDL. Otherwise, the initial result row type descriptor is empty.

4) Give all of these PTF descriptor areas PTF extended names in the PTF name space.

5) Create a status variable (type CHAR(5)) initialized to '00000'.

With this data in place, the DBMS is ready to assemble the argument list for the describe component procedure. The argument list must conform to the algorithm described in Subclause 5.2.4, “Component procedure signatures”. Any scalar arguments that are not compile-time constants (for example, columns) are treated as null.

9.2 Inside the describe component procedure

Primary audience: PTF author

The PTF author supplies the logic of the PTF describe component procedure. The PTF describe component procedure has four tasks:

1) Validate the input arguments. If the input arguments are not acceptable, then the PTF describe component procedure returns an error code in the SQL status argument. Returning an error to the DBMS will cause a syntax error.

2) If the input arguments are acceptable, populate the requested row type descriptor area for each input table.

3) If the PTF was not created with either a <table function column list> or RETURNS ONLY PASS THROUGH, then the PTF describe component procedure must populate the initial result row type descriptor area.

4) If there is private data, the PTF describe component procedure can set values in the private data that will be passed to the later run-time PTF component procedures.

9.3 Using the result of describe

Primary audience: DBMS developer

The PTF describe component procedure returns control to the DBMS. If the status code is not success, then the DBMS quits with a syntax error. Otherwise, the DBMS inspects the requested row type descriptors and the initial result row type descriptor for validity. Requested row type descriptors must satisfy the following constraints:

1) The number of columns must be a positive integer.

2) The NAME components of the top-level item descriptors must be mutually distinct and equal to column names of the input table.

The initial result row type descriptor must satisfy the following:

1) The number of columns in the result row type must be a positive integer, unless the PTF was created with RETURN ONLY PASS THROUGH (in which case there are no proper result columns).

2) Every column must have a name (not null or a zero-length string).

3) There must be no duplicate column names.

4) Every column must have a valid data type, that is, the components of the column's SQL item descriptor area must describe a data type.

If the DBMS does not detect any errors, then the DBMS saves the private data area and the PTF descriptor areas for use during execution.

10 Optimization

Primary audience: DBMS developer, PTF author

The standard does not specify how a DBMS optimizes a PTF invocation, but this subject is naturally important to all three audiences. The query author wants good performance, the DBMS wants to provide good performance, and the PTF author also wants the query author (the PTF author's customer) to obtain good performance.

Although the standard does not specify how to optimize a PTF invocation, the important thing to note here is that the standard has left flexibility for the DBMS and the PTF author to achieve this.

First, if a PTF invocation is partitioned, then each partition is executed in a separate virtual processor. This is the standard's way of saying that these partitions can be executed in parallel on separate processors.

Second, if the SQL-implementation supports Feature B204, “PRUNE WHEN EMPTY”, then this feature can be used to eliminate empty partitions. The PTF author can do this if the PTF does not produce results on empty partitions. Even if the PTF does produce results on an empty partition, the query author can still specify PRUNE WHEN EMPTY if the query author is not interested in results from an empty partition. Pruning empty partitions can be especially significant if there is copartitioning, because it reduces full outer joins to single sided or inner joins.

Third, the interface to the describe component procedure supports column projection in two ways:

1) The describe component procedure can eliminate undesired input columns using the requested row type descriptors.

2) The DBMS can optimize pass-through columns using column projection. For example, if the pass-through surrogate is a compressed value, only the columns that are actually referenced need to be compressed.

Fourth, the DBMS can extend its PTF interface during compilation to support additional PTF component procedures for optimization. For example, query planning frequently attempts to estimate the cost of different query plans based on estimates of the size of intermediate tables. A DBMS might support a PTF component procedure for providing size estimates to be used in query planning. The standard permits such extensions to the PTF component procedure interface, but they are extensions of the standard and are not discussed further in this Technical Report.

(Blank page)

11 Execution

Now we come to the most complicated part of a PTF life cycle, the execution phase. We begin by looking at the execution model in the standard.

11.1 Partitions and virtual processors

Primary audience: DBMS developer

The DBMS begins the execution by partitioning the input tables. Each partition is assigned to a separate virtual processor. There are nuances depending on the number of input tables, whether they have row semantics or set semantics, whether they are copartitioned, and whether they are pruned when empty. Here are some of the cases that can arise:

1) If there are no input tables, then the DBMS must execute the PTF in a single virtual processor.

2) If there is one input table:

   a) If the input table has row semantics, then the DBMS can create an arbitrary number of virtual processors and assign each input row to any virtual processor using any algorithm of its choosing. At one extreme would be to create a virtual processor for each row; at the other extreme would be to process all input rows on the same virtual processor.

   b) If the input table has set semantics and is not partitioned, then the DBMS will first check to see if the input table is empty and PRUNE WHEN EMPTY was specified. In this case, the DBMS does not need to invoke the PTF at all; the result is empty. Otherwise (either there are rows, or KEEP WHEN EMPTY was specified), the DBMS must create a single virtual processor to process the entire input table. If the query has ordered the data, then the DBMS must sort the data before presenting it to the PTF fulfill component procedure.

   c) If the input table has set semantics and is partitioned, then again the DBMS will first check to see if the input table is empty and PRUNE WHEN EMPTY is specified. In this case, the DBMS does not need to invoke the PTF, since the result is empty. Otherwise, the DBMS partitions the data according to the partitioning columns, creating one virtual processor for each partition. The DBMS sorts each partition if requested by the query.

3) If there is one input table with row semantics and one table with set semantics, then essentially the DBMS must perform a cross product of the virtual processors called for by item 1) and item 2) above. For example,

   a) If the table with set semantics is not partitioned, then the DBMS could choose to use an implementation-dependent algorithm to partition the input table with row semantics, creating one virtual processor for each implementation-dependent partition, and "broadcast" the entire table with set semantics to each virtual processor.

   b) If the table with set semantics is partitioned, then the DBMS could set up virtual processors corresponding to the partitions of that table, broadcasting the entire table with row semantics to each virtual processor.

See Subclause 12.4.9, “Virtual processors for Score”, for examples of these scenarios.

4) If there are two input tables with set semantics, then by default the cross product of partitions of one table with partitions of the other table is formed, with one virtual processor for each combination of partitions. This default is overridden if copartitioning is specified. Copartitioning is best understood by looking at the example in Subclause 12.7.9, “Virtual processors for Similarity”.

5) If there is one input table with row semantics and two input tables with set semantics, then the DBMS must create virtual processors that are essentially the cross product of item 1) and item 4).

6) With more input tables, the possible configurations grow by generalizing the preceding points.

Note that the scenarios that involve cross products of partitions are permitted only if Feature B207, “Cross products of partitionings”, is supported; otherwise, these cross product scenarios cannot arise because the syntax that leads to them is prohibited. See Subclause 8.14, “Cross products of partitions”, for a discussion of some of the syntactic restrictions that a DBMS might adopt to avoid having to create cross products of partitions.

On each virtual processor, the DBMS instantiates the PTF private variables using the values that were output from the PTF describe component procedure. Note that the PTF private variables are local to each virtual processor, so they cannot be used to share information between virtual processors.

On each virtual processor, the DBMS also instantiates all the PTF descriptor areas that it had after the PTF describe component procedure exited. The DBMS must also create the following new PTF descriptor areas:

1) For each input table, the cursor row type descriptor. This is the same as the requested row type descriptor, plus one additional column for the pass-through input surrogate column if the input table has pass-through columns.

2) The intermediate result row type descriptor. This is the same as the initial result row type descriptor, plus one additional column (the pass-through output surrogate column) for each input table with pass-through columns.

Each PTF descriptor area receives a PTF extended name in the PTF namespace. The PTF extended names do not need to be the same as used with the describe component procedure, but they might as well be, and we assume that convention in our examples.

The DBMS also needs to allocate a CHAR(5) variable for a status code, initialized to '00000' for success. In our examples we let ST be the name of this status code variable.

11.2 Calling the start component procedure

Primary audience: DBMS developer

On each virtual processor, the DBMS invokes the start component procedure (if any). The technique for generating the parameter list is shown in Subclause 5.2.4, “Component procedure signatures”. In our examples, only two PTFs have start component procedures: CSVreader (see Subclause 12.2, “CSVreader”) and ExecR (see Subclause 12.6, “ExecR”).

11.3 Inside the start component procedure

Primary audience: PTF author

The typical task for the PTF start component procedure is to initialize a non-DBMS resource for the PTF fulfill component procedure. In our examples, CSVreader_start opens a file (Subclause 12.2.12, “Inside CSVreader_start”) and ExecR attaches to an R engine (see Subclause 12.6.11, “Inside ExecR_start”. If the PTF start component procedure is unable to access the resource, the procedure returns an error code to the DBMS. Otherwise, a “handle” in the private data is typically used to pass the resource from the PTF start component procedure to the PTF fulfill component procedure. It is also possible that a PTF start component procedure might generate output row(s), for example, if there are “header” rows for the complete output on a virtual processor. However, the PTF start component procedure cannot set any pass-through columns because it does not have a cursor to supply a valid value for a pass-through surrogate.

11.4 Calling the PTF fulfill component procedure

Primary audience: DBMS developer

On each virtual processor, if the start component procedure has been invoked, the DBMS checks the status code.

Assuming the status code was success, the DBMS opens a read-only nonscrollable cursor for each input table. The cursor ranges over just the rows of the partition assigned to the virtual processor. The DBMS gives the cursor an extended name in the PTF namespace.

11.5 Inside the PTF fulfill component procedure

Primary audience: PTF author

The task of the PTF fulfill component procedure is to generate the output rows. Each output row is passed to the DBMS using the PIPE ROW command, described in Subclause 7.6, “Outputting a row”. If there are input rows, the PTF fulfill component procedure will generally read them from the PTF dynamic cursors.

11.6 Closing cursors

Primary audience: DBMS developer

When the PTF fulfill component procedure finishes on a virtual processor, the DBMS can close the input cursors on that virtual processor.

11.7 Calling the PTF finish component procedure

Primary audience: DBMS developer

On each virtual processor, the DBMS invokes the finish component procedure (if any). In our examples, only two PTFs have finish component procedures: CSVreader and ExecR.

11.8 Inside the PTF finish component procedure

Primary audience: PTF author

The typical task for the PTF finish component is to perform any finishing activity on any non-DBMS resource that was obtained in the PTF start component procedure. In our examples, CSVreader_finishes closes a file (Subclause 12.2.12, “Inside CSVreader_start”) and ExecR releases an R engine (see Subclause 12.6.11, “Inside ExecR_start”). A PTF finish component procedure might also output rows, for example, “summary” or “footer” rows.

11.9 Collecting the output

Primary audience: DBMS developer

Whenever a start, fulfill, or finish component procedure performs a PIPE ROW command, the DBMS must place the row into the result of the <table primary>. If the virtual processors are running concurrently, this will require aggregating the output rows across the servers on which the virtual processors are running.

11.10 Cleanup on a virtual processor

Primary audience: DBMS developer

When a virtual processor has finished, the DBMS can deallocate all the PTF descriptor areas on that virtual processor.

11.11 Final result

Primary audience: DBMS developer

When all virtual processors have finished, the DBMS has the complete result of the PTF invocation, as the union of the rows that were output on each virtual processor by the PTF fulfill component procedure. If there are partitioning columns, the result rows are prefixed with the partitioning columns.

12 Examples

The examples were chosen to exemplify the following scenarios:

Example	Input scenario	Output scenario	Other characteristic
Projection	One input table with row semantics.	Row type provided by query author; pass-through columns.	Fully worked code examples.
CSVreader	No input tables.	Row type determined by reading a file.	Start and finish component procedures, using private data for a file handle.
Pivot	One input table with row semantics and pass-through columns.	Proper result columns determined from descriptor arguments supplied by query.	
Score	One input table with row semantics and pass-through columns, one input table with set semantics and no pass-through columns.	One proper result column declared in DDL.	
TopNplus	One input table with set semantics, sorted. The example does not use pass-through columns, but could be rewritten to use pass-through columns.	Result row type is same as input row type (if rewritten with pass-through columns, there is one proper result column, declared in DDL).	Private data to communicate between describe component procedure and fulfill component procedure.
ExecR	One input table with set semantics and no pass-through columns.	Row type provided by query (rather than inferred by PTF).	Start and finish component procedures, using private data for a handle to an R engine.
Similarity	Two input tables with set semantics and no pass-through columns, sorted.	Fixed row type declared in DDL.	
UDjoin	Two input tables with set semantics, both with pass-through columns.	Only pass-through columns.	
Nested PTF invocation			Nested PTF invocation.

12.1 Projection

12.1.1 Overview

This example shows how a PTF could perform a column projection of its input table. Of course, column projection is a basic capability of SQL, so there is no need to write such a PTF. The main point to this example is that it is fully worked, showing every line of code that the PTF author must write, and every descriptor that the DBMS or the PTF must generate.

The example also demonstrates the use of pass-through columns, which in this example will replicate every input column. Again, this is not an interesting use of pass-through columns, but it demonstrates the technique, including the handling of the input and output surrogate columns.

Using Projection, the query author will be able to write a query such as the following:

```
SELECT P.Empno, E.Empno, E.Ename
FROM TABLE ( Projection (
               Input => TABLE (Emp) AS E,
               Columns => DESCRIPTOR (Empno)
     ) AS P
```

In this query, the input table is Emp; let us assume that it has four columns (Empno INTEGER, Ename VARCHAR(30), Salary INTEGER, Manager INTEGER). The input table has correlation name E. Because the input table has pass-through columns, all columns of Emp are available in the output of Projection, qualified by E. The query has chosen to access E.Empno and E.Ename. Projection also has one proper result column, which is simply a copy of Empno. The proper result column is qualified by the correlation name P, seen as P.Empno in the SELECT list.

12.1.2 Functional specification of Projection

The PTF author decides that Projection will have two parameters:

1) An input table.

2) A descriptor that lists the columns of the input table to be projected as proper result columns of Projection.

The PTF can operate on a row-by-row basis; therefore, the input table will have row semantics. In addition, the PTF author decides to permit pass-through columns.

These decisions lead to the following skeleton DDL for Projection:

```
CREATE FUNCTION Projection (
    Input TABLE PASS THROUGH WITH ROW SEMANTICS,
    Columns DESCRIPTOR
) RETURNS TABLE
DETERMINISTIC
READS SQL DATA
```


12.1.3 Design specification for Projection

The design specification provides the details that are private to the PTF (not visible to the query author). For the design specification, the PTF author decides:

1) Whether start and finish component procedures are required.

   Projection does not require any resources outside the DBMS, so start and finish component procedures are not required.

2) The names for the PTF component procedures.

   The PTF author decides to name the describe component procedure Projection_describe and the fulfill component procedure Projection_fulfill.

3) Whether the PTF needs any private data.

   There is no information to pass from compile time to run-time, other than the information that will be captured in the descriptors that the DBMS will build and pass to the fulfill component procedure. This fact will become clear when we look at the logic of the fulfill component procedure. (In actual PTF development, the PTF author may revisit this decision as the development unfolds.)

As a result of these decisions, the PTF author can enhance the skeleton DDL as follows:

```
CREATE FUNCTION Projection (
    Input TABLE PASS THROUGH WITH ROW SEMANTICS,
    Columns DESCRIPTOR
) RETURNS TABLE
DETERMINISTIC
READS SQL DATA
DESCRIBE WITH PROCEDURE Projection_describe
FULFILL WITH PROCEDURE Projection_fulfill
```

12.1.4 Projection component procedures

The DBMS should provide a tool for the PTF author that will generate the signatures of the PTF component procedures from the skeleton PTF definition.

A key decision for the DBMS is the maximum length of descriptor and cursor names. These names will be automatically generated and can be meaningless, other than the fact that they must be unique. As explained in Subclause 7.2, “PTF extended names”, these names can be short. Using just the uppercase Latin letters, a one-character name can support up to 26 different descriptor names, and up to 26 different cursor names. A two-character name using Latin letters or digits in the second character can support up to 26*36 = 936 different names. The examples in this Technical Report assume two-character names, which is more than adequate for the length of parameter lists in the examples.

The DBMS tool will generate parameter definitions for the PTF component procedures that are derived from PTF parameters. The DBMS should document its conventions for generating the parameters of the PTF component procedures. The conventions used in this Technical Report are as follows:

1) Scalar parameters are simply copied from the PTF parameter definition to the corresponding PTF component parameter definition.

2) A descriptor parameter of a PTF generates a VARCHAR(2) parameter of the PTF component procedures. To highlight that the parameter is a descriptor name, "_descr" is appended to the parameter name in the PTF component procedure.

3) A table parameter requires several descriptors and one cursor, as enumerated in Subclause 5.2.4, "Component procedure signatures". Each of these is a VARCHAR(2) parameter in the PTF component procedures. The names of these parameters are derived by appending specific strings to the PTF parameter name, as follows:

Descriptor or cursor	Suffix on the parameter name
Full row type	_row_descr
Partitioning (set semantics only)	_pby_descr
Ordering (set semantics only)	_order_descr
Requested row type	_request_descr
Cursor row type	_cursor_descr
Cursor	_cursor_name

4) There are two result row descriptors; this Technical Report calls these the Initial_result_row and the Intermediate_result_row, both of type VARCHAR(2).

5) There is a status parameter of type CHAR(5) named Status.

6) The describe component procedure is always DETERMINISTIC.

7) The other component procedures copy either DETERMINISTIC or NOT DETERMINISTIC from the PTF definition.

8) The fulfill component procedure copies the SQL-data access (either CONTAINS SQL or READS SQL) from the PTF definition.

9) The other component procedures have SQL-data access CONTAINS SQL.

Using the DBMS tool as specified above, the output of the tool might look like this:

```
CREATE PROCEDURE Projection_describe (
    IN Input_row_descr VARCHAR(2),
    IN Input_request_descr VARCHAR(2),
    IN Columns_descr VARCHAR(2),
    IN Initial_result_row VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL DETERMINISTIC CONTAINS SQL
SQL SECURITY DEFINER
BEGIN
END

CREATE PROCEDURE Projection_fulfill (
    IN Input_cursor_descr VARCHAR(2),
    IN Input_cursor_name VARCHAR(2),
    IN Columns_descr VARCHAR(2),
    IN Intermediate_result_row VARCHAR(2),
```



```
    INOUT Status CHAR(5)
) LANGUAGE SQL DETERMINISTIC READS SQL DATA
SQL SECURITY DEFINER
BEGIN
END
```

12.1.5 Invoking Projection

The query author writes the following query:

```
SELECT P.Empno, E.Empno, E.Ename
FROM TABLE ( Projection (
                Input => TABLE (Emp) AS E,
                Columns => DESCRIPTOR (Empno)
           ) AS P
```

12.1.6 Calling Projection_describe

Given the query, the DBMS must assemble the arguments to Projection_describe. There are five arguments:

1) Input_row_descr, the descriptor of the input table's row type. Emp has the following signature:

```
TABLE Emp (
    Empno INTEGER,
    Ename VARCHAR(30),
    Salary INTEGER,
    Manager INTEGER )
```

The DBMS builds a descriptor for Emp's row type, naming it 'I1', with the following contents:

	Content
Header	COUNT = 4 TOP_LEVEL_COUNT = 4 Other components unspecified
Item 1	NAME = 'EMPNO' LEVEL = 0 TYPE = 4 (for INTEGER) Other components unspecified

	Content
Item 2	NAME = 'ENAME' LEVEL = 0 TYPE = 12 (for VARCHAR) LENGTH = 30 CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the catalog, schema, and local name of the default character set. Other components unspecified
Item 3	NAME = 'SALARY' LEVEL = 0 TYPE = 4 (for INTEGER) Other components unspecified
Item 4	NAME = 'MANAGER' LEVEL = 0 TYPE = 4 (for INTEGER) Other components unspecified

2) Input_request_descr, the input table's requested row type. The DBMS assigns this descriptor the PTF extended name 'A1'. This is an empty descriptor, like this:

	Content
Header	COUNT = 0 TOP_LEVEL_COUNT = 0 Other components unspecified

3) Columns_descr, the descriptor generated from the query's argument. The DBMS assigns this the name 'Q', with the following contents:

	Content
Header	COUNT = 1 TOP_LEVEL_COUNT = 1 Other components unspecified
Item 1	NAME = 'EMPNO' LEVEL = 0 TYPE = 0 (for unspecified type) Other components unspecified

4) Initial_result_row, the descriptor of the proper result columns. The DBMS assigns this the name 'R'. This is another empty descriptor:

	Content
Header	COUNT = 0 TOP_LEVEL_COUNT = 0 Other components unspecified

5) Status, the status argument. The DBMS allocates a CHAR(5), initializes it '00000' (success).

After assembling these arguments, the DBMS can call Projection_describe like this:

```
CALL Projection_describe (
    Input_row_descr => 'I1',
    Input_request_descr => 'A1',
    Columns_descr VARCHAR => 'Q',
    Initial_result_row => 'R',
    Status => ST
)
```

12.1.7 Inside Projection_describe

The tasks for Projection_describe are:

1) Examine the arguments for validity, returning an error if they are not valid.

2) Populate the requested row type descriptor with the names of the columns that Projection wishes to receive during execution.

3) Populate the initial result row type with the names and data types of the proper result columns.

The code for Projection_describe might be:

```
CREATE PROCEDURE Projection_describe (
    IN Input_row_descr VARCHAR(2),
    IN Input_request_descr VARCHAR(2),
    IN Columns_descr VARCHAR(2),
    IN Initial_result_row VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL DETERMINISTIC CONTAINS SQL
SQL SECURITY DEFINER
BEGIN
  /* local variables used to scan Columns_descr
  ** (Columns1, C1, Name1) and Input_row_descr
  ** (Columns2, C2, Name2)
  */
  DECLARE Columns1, Columns2 INTEGER;
  /* number of columns */
  DECLARE C1, C2 INTEGER; /* current column number */
  DECLARE Name1, Name2 VARCHAR(128);
  /* current column name */
  DECLARE Found BOOLEAN; /* flag set to TRUE if a
                         ** matching column name is found */
  /* Copy Column_descr to Input_request_descr.
```

```
  ** Note that the DBMS only requires the column
  ** names in Input_request_descr, not their types.
  ** It is an error if a column name in
  ** Input_request_descr is not a column in
  ** Input_row_descr, but the DBMS will perform
  ** that check so Projection_describe does not
  ** need to */

  COPY DESCRIPTOR PTF Columns_descr
  TO PTF Input_request_descr;

  /* Populate Initial_result_row. Each column
  ** named in Columns_descr provides the name
  ** of a proper result column, because this is
  ** implementing a projection operation. We need
  ** to find the column in Input_row_descr and
  ** copy the type information to Initial_result_row.
  */

  GET DESCRIPTOR PTF Column_descr
    Columns1 = TOP_LEVEL_COUNT;
  GET DESCRIPTOR PTF Input_row_descr
    Columns2 = TOP_LEVEL_COUNT;

  /* Outer loop scans the column names
  ** in Columns_descr
  */
  SET C1 = 1;
  Outerloop: WHILE (C1 <= Columns1) DO
    GET DESCRIPTOR PTF Column_descr VALUE C1
      Name1 = NAME;

    /* Inner loop scans the column names
    ** in Input_row_descr
    */
    SET C2 = 1;
    SET Found = FALSE;
    Innerloop: WHILE (C2 <= Column2) DO
      GET DESCRIPTOR PTF Input_row_descr VALUE C2
        Name2 = NAME;

      /* Check for a match */
      IF (Name1 = Name2) THEN
        COPY DESCRIPTOR PTF Input_row_descr
          VALUE C2 (NAME, TYPE)
          TO PTF Initial_result_row VALUE C1;
        SET Found = TRUE;
        LEAVE Innerloop;
      END IF;
      SET C2 = C2+1;
    END WHILE Innerloop;

/* If no match was found, return an error */
    IF (Found = FALSE) THEN
      SET Status = '42000'; /* syntax error */
      RETURN;
    END IF;
```



```
/* go to next proper result column */
    SET C1 = C1+1;
  END WHILE Outerloop;

/* Success! just return; Status is already success */
RETURN;
END
```

12.1.8 Result of Projection_describe

In this example, Projection_describe populates the Input_request_descr descriptor as follows:

	Content
Header	COUNT = 1 TOP_LEVEL_COUNT = 1 Other components unspecified
Item 1	NAME = 'EMPNO' LEVEL = 0 TYPE = 0 (for unspecified type) Other components unspecified

The DBMS checks that this is a non-empty list of distinct column names of the input table Emp.

Projection_describe populates the Initial_result_row descriptor as follows:

	Content
Header	COUNT = 1 TOP_LEVEL_COUNT = 1 Other components unspecified
Item 1	NAME = 'EMPNO' LEVEL = 0 TYPE = 4 (for INTEGER) Other components unspecified

The DBMS checks this descriptor for validity as an output row type: at least one column, acceptable distinct column names (not zero-length or null), and valid data types.

12.1.9 Virtual processors for Projection

There is a single input table with row semantics; therefore, the DBMS is free to create any number of virtual processors and assign rows to them in an implementation-dependent fashion (round robin, random, *etc*.)

Prior to starting any virtual processors, the DBMS can determine the row type of the input table cursor, since that will be the same on all virtual processors. Since the input table has pass-through columns, the cursor row type is the requested row type plus one additional column, the pass-through input surrogate column. The DBMS gives this column an implementation-dependent name and data type.

12.1.10 Calling Projection_fulfill

On each virtual processor, the DBMS calls Projection_fulfill. Recall that its signature is:

```
PROCEDURE Projection_fulfill (
    IN Input_cursor_descr VARCHAR(2),
    IN Input_cursor_name VARCHAR(2),
    IN Columns_descr VARCHAR(2),
    IN Intermediate_result_row VARCHAR(2),
    INOUT Status CHAR(5)
)
```

This requires three descriptors, a cursor, and a status variable, as follows:

1) Input_cursor_descr describes the input cursor's row type. This consists of the columns that Projection_describe requested in using Input_request_descr, plus one column for the pass-through input surrogate. The DBMS names the pass-through input surrogate column '$surr1' (this name is implementation-dependent, but must not be equivalent to any column name of the requested row type or the intermediate result row type). The DBMS builds a descriptor and names it 'CR'; the contents look like this:

	Content
Header	COUNT = 2 TOP_LEVEL_COUNT = 2 Other components unspecified
Item 1	NAME = 'EMPNO' LEVEL = 0 TYPE = 4 (for INTEGER) Other components unspecified
Item 2	NAME = '$surr1' LEVEL = 0 TYPE = implementation-dependent

2) The DBMS opens a cursor and gives it the PTF extended name 'CN'.

3) The DBMS builds a descriptor from the Columns argument in the PTF invocation, naming it 'Q'. This has the same contents as previously seen in Subclause 12.1.6, “Calling Projection_describe”.

4) The DBMS builds a descriptor of the intermediate result row type, calling it 'MR'. The intermediate result row consists of the initial result row plus the pass-through output surrogate column. The name and type of the pass-through output surrogate column must be the same as the name of the pass-through input surrogate column.

5) The DBMS allocates a CHAR(5) variable called ST for the status variable.

After creating and naming these things, the DBMS calls Projection_fulfill like this:

```
CALL Projection_fulfill (
    Input_cursor_descr => 'CR',
    Input_cursor_name => 'CN',
    Columns_descr => 'Q',
    Intermediate_result_row => 'MR',
    Status => ST
)
```

12.1.11 Inside Projection_fulfill

The task for Projection_fulfill is to read the input cursor and write output rows. Note that the input cursor's row type is precisely the same as the intermediate output row. This means that Projection_fulfill's logic can be very simple: read a row from the cursor; test for end of data; if data was read, then copy the input row to the intermediate output row and repeat.

```
CREATE PROCEDURE Projection_fulfill (
    IN Input_cursor_descr VARCHAR(2),
    IN Input_cursor_name VARCHAR(2),
    IN Columns_descr VARCHAR(2),
    IN Intermediate_result_row VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL DETERMINISTIC CONTAINS SQL
SQL SECURITY DEFINER
BEGIN

  /* condition handler for the subsequent loop.
  ** When the input cursor is exhausted this
  ** handler will be invoked, causing
  ** Projection_fulfill to exit.
  ** There is no other task for the condition handler
  ** so a no-op would be sufficient. SQL does not
  ** have a no-op, so we set Status to success */
  DECLARE EXIT HANDLER FOR NOT FOUND
  SET Status = '00000';

  /* Loop until there is no more data */
  Loop: WHILE TRUE DO
    FETCH PTF Input_cursor_name
      INTO DESCRIPTOR PTF Input_cursor_row;
    COPY DESCRIPTOR PTF Input_cursor_row (DATA)
      TO PTF Intermediate_result_row;
    PIPE ROW PTF Intermediate_result_row;
  END WHILE Loop;
END
```

12.1.12 Collecting the results

Each time that Projection_fulfill executes a PIPE ROW statement, the DBMS builds a row of output. The intermediate result row that is delivered to the DBMS has two columns:

1) The proper result column EMPNO. This column can be copied into the complete result row, where it is qualified by the correlation name P.

2) The pass-through output surrogate column $surr1. This column must be expanded into the columns of Emp, qualified by E.

12.1.13 Cleanup

When Projection_fulfill finishes on a virtual processor, the DBMS can destroy the virtual processor.

12.2 CSVreader

This example was introduced in Subclause 3.2.1, “CSVreader”.

12.2.1 Overview

A spreadsheet can generally output a comma-separated list of values. Generally, the first line of the file contains a list of column names, and subsequent lines of the file contain data. The data in general can be treated as a large VARCHAR. However, some of the fields may be numeric or datetime. The PTF author has provided a PTF called CSVreader designed to read a file of comma-separated values and interpret this file as a table.

The distinguishing feature of this example is that there are no input tables.

12.2.2 Functional specification of CSVreader

The PTF author decides that CSVreader have the following inputs:

1) The file name, a character string.

2) An optional list of column names to be treated as REAL.

3) An optional list of column names to be treated as DATE.

Thus the signature that is visible to the query author will be:

```
FUNCTION CSVreader (
    File VARCHAR(1000),
    Floats DESCRIPTOR DEFAULT NULL,
    Dates DESCRIPTOR DEFAULT NULL )
RETURNS TABLE
NOT DETERMINISTIC
CONTAINS SQL
```

Note that this example has no input table. This example is non-deterministic because the results will vary depending on the contents of the file. The SQL-data access is CONTAINS SQL because there are no table parameters and no side tables that are read by the PTF.

12.2.3 Design specification for CSVreader

For the design specification, the PTF author decides:

1) Whether PTF start and/or finish component procedures are required.

   The PTF author decides to have a PTF start component procedure to open the input file, and a PTF finish component procedure to close the input file.

An alternative design would be to simply open and close the file in the PTF fulfill component procedure. This design has been chosen to illustrate the technique, without recommending or discouraging this technique.

2) The names of the PTF component procedures.

   The PTF author decides to name the PTF component procedures CSVreader_describe, CSVreader_start, CSVreader_fulfill and CSVreader_finish.

3) The private data for the PTF component procedures..

   The PTF start component procedure will open the file and pass a handle to subsequent runtime stages.

After making these decisions, the PTF author writes the following skeleton definition of CSVreader:

```
CREATE FUNCTION CSVreader (
    File VARCHAR(1000),
    Floats DESCRIPTOR DEFAULT NULL,
    Dates DESCRIPTOR DEFAULT NULL )
RETURNS TABLE
NOT DETERMINISTIC
CONTAINS SQL
PRIVATE DATA (
    FileHandle INTEGER )
DESCRIBE WITH PROCEDURE CSVreader_describe
START WITH PROCEDURE CSVreader_start
FULFILL WITH PROCEDURE CSVreader_fulfill
FINISH WITH PROCEDURE CSVreader_finish
```

12.2.4 CSVreader component procedures

The DBMS should provide a tool that takes the preceding skeleton DDL for CSVreader and generates the following skeleton signatures for the PTF component procedures. We are assuming that the PTF author will implement the PTF in SQL/PSM.

```
CREATE PROCEDURE CSVreader_describe (
    INOUT FileHandle INTEGER,
    IN File VARCHAR(1000),
    IN Floats_descr VARCHAR(2),
    IN Dates_descr VARCHAR(2),
    IN Initial_result_descr VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL DETERMINISTIC CONTAINS SQL
SQL SECURITY DEFINER
BEGIN
END

CREATE PROCEDURE CSVreader_start (
    INOUT FileHandle INTEGER,
    IN File VARCHAR(1000),
    IN Floats_descr VARCHAR(2),
    IN Dates_descr VARCHAR(2),
    IN Intermediate_result_descr VARCHAR(2),
    INOUT Status CHAR(5)
```



```
) LANGUAGE SQL NOT DETERMINISTIC CONTAINS SQL
SQL SECURITY DEFINER
BEGIN
END

CREATE PROCEDURE CSVreader_fulfill (
    INOUT FileHandle INTEGER,
    IN File VARCHAR(1000),
    IN Floats_descr VARCHAR(2),
    IN Dates_descr VARCHAR(2),
    IN Intermediate_result_descr VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL NOT DETERMINISTIC CONTAINS SQL
SQL SECURITY DEFINER
BEGIN
END

CREATE PROCEDURE CSVreader_finish (
    IN FileHandle INTEGER,
    IN File VARCHAR(1000),
    IN Floats_descr VARCHAR(2),
    IN Dates_descr VARCHAR(2),
    IN Intermediate_result_descr VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL NOT DETERMINISTIC CONTAINS SQL
SQL SECURITY DEFINER
BEGIN
END
```

NOTE 2 —

1) DBMS tool output should go to one or two files. If the PTF author will be writing in SQL/PSM, then the output can go to a single file with the preceding contents. If the PTF author will be writing in an external language such as C, then the output should go to two files, one for the SQL DDL similar to the above and another with a skeleton procedure header in the external language.

2) The names of the parameters are arbitrary as far as the DBMS is concerned; actually, they are not even required for the DDL to accompany an external language. However, the PTF author will want the parameter names of the PTF component procedures to be similar to the parameter names of the PTF itself. The DBMS tool should have some predictable algorithm for generating the parameter names. In this example, the DBMS tool has added _descr to the names of the PTF descriptor area parameters.

3) The penultimate parameter is the extended name of the descriptor area for the result row type of the PTF. For the describe component procedure, this is the initial result row type descriptor, and for the run-time component procedures this is the intermediate result row type descriptor.

4) The final parameter is an SQLSTATUS code. This is an INOUT parameter which the DBMS initializes to success (‘00000'); the PTF component procedure must change this value to indicate failure. Later, we will consider how to write the routine body of the PTF component procedures.

12.2.5 Implementation of CSVreader

Next the PTF author must write the bodies of the PTF component procedures. We discuss the logic of each PTF component procedure below at the point where the procedure is invoked, to provide context to understand the logic of each component procedure.

12.2.6 Invoking CSVreader

The query author can see the following signature:

```
FUNCTION CSVreader (
    File VARCHAR(1000),
    Floats DESCRIPTOR DEFAULT NULL,
    Dates DESCRIPTOR DEFAULT NULL )
RETURNS TABLE
NOT DETERMINISTIC
CONTAINS SQL
```

The PTF author has documented to the query author the appropriate use of the parameters:

— File: a character string containing the name of a file. The file should contain text formatted into lines. Each line is subdivided by commas into fields. In the first line, the fields are regarded as supplying column names. Each remaining line of the table produces one row of output.

— Floats: by default, a field of the input file is regarded as a character string. However, this argument can be used to declare the names of columns that are to be interpreted as floating point.

— Dates: this argument can be used to declare the names of columns that are to be interpreted as dates according to some format.

Using this information, the query author writes the following invocation of CSVreader PTF:

```
SELECT *
FROM TABLE ( CSVreader (
    File => 'abc.csv',
    Floats =>
      DESCRIPTOR ("principle", "interest")
    Dates => DESCRIPTOR ("due_date")
     ) ) AS S
```

To run successfully, there must be an operating system file named abc.csv. The first line of the file must have a comma-separated list of column names, among which must be principle, interest, and due_date. Each remaining line of the file must be a comma-separated list of values; the fields for the principle and interest columns must be formatted numerically; the field for due_date must be formatted as a date.

12.2.7 Calling CSVreader_describe

In order to compile the preceding query, the DBMS calls the PTF describe component procedure, CSVreader_describe. As stated in Subclause 12.2.4, “CSVreader component procedures”, the parameter list of CSVreader_describe is:

```
PROCEDURE CSVreader_describe (
    INOUT FileHandle INTEGER,
    IN File VARCHAR(1000),
    IN Floats_descr VARCHAR(2),
    IN Dates_descr VARCHAR(2),
    IN Initial_result_descr VARCHAR(2),
INOUT Status CHAR(5)
)
```


The private data is:

```
PRIVATE DATA ( FileHandle INTEGER )
```

The DBMS must allocate the private variable shown above, initialized to null. This will be passed as the first argument of CSVreader_describe.

The next argument, File, is a scalar that is simply copied from the invocation of CSVreader.

The next two arguments, Floats_descr and Dates_descr, correspond to Floats and Dates, respectively, in the invocation of CSVreader. The query author has passed the following two DESCRIPTOR constructors in the invocation of CSVreader:

```
DESCRIPTOR ("principle", "interest")
DESCRIPTOR ("due_date")
```

The corresponding arguments in CSVreader_describe are character strings holding the PTF extended names of the PTF descriptor areas. The DBMS might name these PTF descriptor areas Q1 and Q2. Q1 has the following contents:

	Content
Header	COUNT = 2 TOP_LEVEL_COUNT = 2 Other components unspecified
Item 1	NAME = 'principle' LEVEL = 0 TYPE = 0 Other components unspecified
Item 2	NAME = 'interest' LEVEL = 0 TYPE = 0 Other components unspecified

Q2 has the following contents:

	Content
Header	COUNT = 1 TOP_LEVEL_COUNT = 1 Other components unspecified
Item 1	NAME = 'due_date' LEVEL = 0 TYPE = 0 Other components unspecified

The DBMS must also allocate an empty read-write PTF descriptor area for the initial result row type. Let this PTF descriptor area be named R. R has the following contents:

	Content
Header	COUNT = 0 TOP_LEVEL_COUNT = 0 Other components unspecified

Note that although R has no SQL item descriptor areas, the describe component procedure can (and must) add more, up to some implementation-defined maximum number of columns.

Finally the DBMS must allocate a CHAR(5) variable for the status code, initialized to '00000'. Let ST be the status code variable.

Now the DBMS makes the following invocation:

```
CALL CSVreader_describe (
    FileHandle => FileHandle,
    File => 'abc.csv',
    Floats_descr => 'Q1',
    Dates_descr => 'Q2',
    Initial_result_row => 'R'
    Status => ST
)
```

12.2.8 Inside CSVreader_describe

The basic objective of CSVreader_describe is to populate the PTF descriptor area whose name is passed in the Initial_result_row argument. If CSVreader_describe is unable to do this, for example, if the input argument File does not contain the name of a file that CSVreader_describe can open, then CSVreader_describe returns an error code in the argument Status. (Note that this argument has been initialized by the DBMS to indicate no error, so CSVreader_describe only needs to set ST in case an error is detected.)

The logic for CSVreader_describe may look something like this:

1) Open the file whose name is passed in the File argument. If the open operation fails, return an error in the Status argument.

2) Read the first line of the input file. If the file is empty, return an error.

3) Initialize a variable Colno = 0;

4) In a loop, parse the first line into tokens delimited by commas. For each token:

    a) Increment Colno.

    b) Increase the number of item descriptor areas in the result row type descriptor area:

    ```
    SET DESCRIPTOR Initial_result_row COUNT = Colno;
    ```

    This has the side effect of adding an empty SQL item descriptor area at the end of the result row descriptor area.

    c) Place the token in a variable, Colname.

d) Set the name of the new column to Colname:

```
SET DESCRIPTOR Initial_result_row
  VALUE Colno NAME = Colname;
```

e) Determine the data type of the column, as follows:

i) If the column name is found in the input PTF descriptor area Floats_descr, then the column type is REAL,

ii) If the column name is found in the input PTF descriptor area Dates_descr, then the column type is DATE.

iii) Otherwise the column type is VARCHAR(100).

f) Use SET DESCRIPTOR to set the type of the column. For example, if the column type is VARCHAR(100), this statement could be used:

```
SET DESCRIPTOR Initial_result_row
  VALUE Colno TYPE = 12, LENGTH = 100;
```

5) Close the input file.

If CSVreader_describe wished to communicate with the run-time PTF component procedures, it could do so by setting values in the private variables. In this example, there is no information to convey that won't be conveyed in the resuult row type descriptor area. Therefore, CSVreader can simply leave the private data (FileHandle) untouched.

12.2.9 Result of CSVreader_describe

The result of CSVreader_describe will depend on the first line in abc.csv. We are assuming that the first line is:

```
docno,name,due_date,principle,interest
```

Then CSVreader_describe will populate the initial result row type descriptor to describe five columns, as named above. The columns named "principle" and "interest" will be of type REAL and the column named "due_date" will be of type DATE. The other columns will be of type VARCHAR(100). Therefore, the descriptor looks like this:

	Content
Header	COUNT = 5 TOP_LEVEL_COUNT = 5 Other components unspecified

	Content
Item 1	NAME = 'docno' LEVEL = 0 TYPE = 12 LENGTH = 100 CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the catalog, schema, and local name of the default character set. Optionally, COLLATION_CATALOG, COLLATION_SCHEMA, and COLLATION_NAME may specify a collation. Other components unspecified
Item 2	NAME = 'name' LEVEL = 0 TYPE = 12 LENGTH = 100 CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the catalog, schema, and local name of the default character set. Optionally, COLLATION_CATALOG, COLLATION_SCHEMA, and COLLATION_NAME may specify a collation. Other components unspecified
Item 3	NAME = 'due_date' LEVEL = 0 TYPE = 9 DATETIME_INTERVAL_CODE = 1 Other components unspecified
Item 4	NAME = 'principle' LEVEL = 0 TYPE = 7 Other components unspecified
Item 5	NAME = 'interest' LEVEL = 0 TYPE = 7 Other components unspecified

Then the row type looks like this:

Correlation name	S				
Column name	docno	name	due_date	principle	interest

Data type	VARCHAR(100)	VARCHAR(100)	DATE	REAL	REAL

The query was:

```
SELECT *
FROM TABLE ( CSVreader (
                 File => 'abc.csv',
                 Floats => DESCRIPTOR ("principle", "interest")
                 Dates => DESCRIPTOR ("due_date")
           ) ) AS S
```

Based on the row type generated by CSVreader_describe, the SELECT * is equivalent to:

```
SELECT S."docno", S."name", S."due_date", S."principle", S."interest"
```

12.2.10 Virtual processor for CSVreader

To execute the invocation, the DBMS uses a single virtual processor, since there are no input tables.

The private data for CSVreader is:

```
PRIVATE DATA ( FileHandle INTEGER )
```

The DBMS must allocate memory on the virtual processor for this private variable, plus a CHAR(5) for the SQL status code. We will portray the private variable with the same name as shown in the PRIVATE DATA declaration above; we give the status code variable the name ST.

The DBMS must also instantiate copies of the SQL descriptor areas that were present after CSVreader_describe completed. We assume that they have been given the same PTF extended names as before; they were 'Q1' and 'Q2' for the two SQL descriptor areas provided by the query, and 'R' for the SQL descriptor area of the result row type. The contents of these SQL descriptor areas are found in Subclause 12.2.7, “Calling CSVreader_describe”, and Subclause 12.2.9, “Result of CSVreader_describe”.

12.2.11 Calling CSVreader_start

The signature for CSVreader_start is given in Subclause 12.2.4, “CSVreader component procedures”, as:

```
PROCEDURE CSVreader_start (
    INOUT FileHandle INTEGER,
    IN File VARCHAR(1000),
    IN Floats_descr VARCHAR(2),
    IN Dates_descr VARCHAR(2),
    IN Intermediate_result_row VARCHAR(2),
    INOUT Status CHAR(5)
)
```

Since there are no input tables, this is almost identical to the signature for CSVreader_describe, the one difference being that the describe component procedure populates the initial result row descriptor, whereas CSVreader_start has the intermediate result row descriptor as input. This example has no pass-through columns, so the interme-

diate result row descriptor is identical to the initial result row descriptor as it was output by CSVreader_describe. The other descriptors are the same as on input to CSVreader_describe. Assuming the PTF descriptor areas have the same names as during compilation, the DBMS calls CSVreader_start as follows:

```
CALL CSVreader_start (
    FileHandle => FileHandle,
    File => 'abc.csv',
    Floats_descr => 'Q1',
    Dates_descr => 'Q2',
    Intermediate_result_row => 'R'
    Status => ST
)
```

12.2.12 Inside CSVreader_start

CSVreader_start must initialize the processing on the virtual processor. The file named by the File argument (abc.csv) should be opened, and a file handle placed in the FileHandle argument. As a safety check, CSVreader_start should read the first line of the file and confirm that the column names are correctly described by the SQL descriptor area for the result row type, the one whose PTF extended name is passed in Intermediate_result_row argument ('R'). The reason for the safety check is that contents of the file may have changed. Note that it is possible to prepare the invocation at one time and execute it later. If any of these steps fail, then an error can be returned in the status code argument.

12.2.13 Calling CSVreader_fulfill

The DBMS checks the status code that was returned from CSVreader_start. If it is not '00000' (success), then the DBMS terminates the virtual processor. Otherwise, the DBMS proceeds to call the next stage, CSVreader_fulfill.

The signature for CSVreader_fulfill is given in Subclause 12.2.4, “CSVreader component procedures”, as:

```
PROCEDURE CSVreader_fulfill (
    INOUT FileHandle INTEGER,
    IN File VARCHAR(1000),
    IN Floats_descr VARCHAR(2),
    IN Dates_descr VARCHAR(2),
    IN Intermediate_result_row VARCHAR(2),
    INOUT Status CHAR(5)
)
```

Since there is no input table, the input arguments are the same as the preceding stage, CSVreader_start, and can simply be maintained by the DBMS without change.

12.2.14 Inside CSVreader_fulfill

CSVreader_start has already read the first line of the input file whose handle is in the parameter FileHandle. CSVreader_fulfill should now read the remaining lines of the input file. Each line is parsed by comma delimiters,

and the fields are mapped to columns of the output row. The values of the columns should be written (using SET DESCRIPTOR) to the DATA component of the PTF descriptor area whose name is passed in the Intermediate_result_row argument. After setting the DATA component for every column of a result row, CSVreader_fulfill uses PIPE ROW to send the row to the DBMS.

For example, suppose the following line is read:

```
123,Mary,01/01/2014,234.56,345.67
```

CSVreader_fulfill then performs SET DESCRIPTOR commands equivalent to the following:

```
SET DESCRIPTOR PTF Intermediate_result_row VALUE 1
  DATA = '123';
SET DESCRIPTOR PTF Intermediate_result_row VALUE 2
  DATA = 'Mary';
SET DESCRIPTOR PTF Intermediate_result_row VALUE 3
  DATA = CAST ('2014-01-01' AS DATE);
SET DESCRIPTOR PTF Intermediate_result_row VALUE 4
  DATA = CAST ('234.56' AS REAL);
SET DESCRIPTOR PTF Intermediate_result_row VALUE 5
  DATA = CAST ('345.67' AS REAL);
```

After setting all columns of the output row, CSVreader_fulfill sends the row to the DBMS using a PIPE ROW command:

```
PIPE ROW PTF Intermediate_result_row;
```

CSVreader_fulfill should do this repeatedly until the end of file is reached, calling PIPE ROW once for each input line. CSVreader_fulfill should also incorporate logic to check that the input is correctly formed; if an error is encountered, then an error code can be returned in the Status argument.

12.2.15 Collecting the output

The DBMS collects the output that it receives via PIPE ROW commands performed within CSVreader_fulfill.

12.2.16 Calling CSVreader_finish

When CSVreader_fulfill completes, the DBMS checks the status code.

Next the DBMS calls CSVreader_finish to perform final cleanup. The signature for CSVreader_finish is given in Subclause 12.2.4, “CSVreader component procedures”, as:

```
PROCEDURE CSVreader_finish (
    INOUT FileHandle INTEGER,
    IN File VARCHAR(1000),
    IN Floats_descr VARCHAR(2),
    IN Dates_descr VARCHAR(2),
    IN Intermediate_result_row VARCHAR(2),
    INOUT Status CHAR(5)
)
```

The input arguments are the same as the preceding stage, CSVreader_fulfill, and can simply be maintained by the DBMS without change. Therefore, the DBMS uses the following invocation:

```
CALL CSVreader_finish (
    FileHandle => FileHandle,
    File => 'abc.csv',
    Floats_descr => 'Q1',
    Dates_descr => 'Q2',
    Intermediate_result_row => 'R'
    Status => ST
)
```

12.2.17 Inside CSVreader_finish

CSVreader_finish closes the input file indicated by FileHandle.

12.2.18 Cleanup

After CSVreader_finish completes, the DBMS may do any final cleanup, such as deallocating the PTF descriptor areas.

12.3 Pivot

This example was introduced in Subclause 3.2.2, “Pivot”.

12.3.1 Overview

In general, a pivot is an operation that reads a row and outputs several rows. Generally, the input is denormalized and the output is normalized. For example, maybe an input table has six columns, forming three pairs of (phone type, phone number), and the user wishes to normalize this into a table with two columns.

12.3.2 Functional specification of Pivot

The functional specification specifies the interface that is visible to the query author.

Pivot needs the following inputs:

— An input table. Since a pivot can be performed on a single row, this input table has row semantics. This input table will use Feature B205, “Pass-through columns”, making all columns of the input table available in the output, qualified by the input table argument's range variable.

— A list of the input columns that will go into the first output row, a list for the second row, *etc*. Each of these suggests a PTF descriptor area. In general, we don't know how many pivots the query author will want to do, so the technique will be to just declare a large number of PTF descriptor areas, which can default to null. The query author will supply as many as desired.

— Since the columns to be pivoted will all have distinct names, such as (Phtype1, Phnumber1), (Phtype2, Phnumber2), ..., the PTF will not know what the desired output column names are for the pivoted columns. Therefore, the PTF will require a PTF descriptor area for these output column names.

The parameter list looks like this:

```
CREATE FUNCTION Pivot (
    Input TABLE PASS THROUGH WITH ROW SEMANTICS,
    Output_pivot_columns DESCRIPTOR,
    Input_pivot_columns1 DESCRIPTOR,
    Input_pivot_columns2 DESCRIPTOR DEFAULT NULL,
    Input_pivot_columns3 DESCRIPTOR DEFAULT NULL,
    Input_pivot_columns4 DESCRIPTOR DEFAULT NULL,
    Input_pivot_columns5 DESCRIPTOR DEFAULT NULL
) RETURNS TABLE
DETERMINISTIC
READS SQL DATA
```

This shows the capability to pivot at most 5 sets of columns. Of course, the PTF author could support many more.

12.3.3 Design specification for Pivot

The design specification specifies details that are private, that is, not visible to the query author. For the design specification, the PTF author decides:

1) Whether PTF start and/or finish component procedures are required.

   Pivot does not need any resources not provided by the DBMS, so there are no start or finish component procedures.

2) The names of the PTF component procedures.

   The PTF author decides to name the PTF component procedures Pivot_describe and Pivot_fulfill.

3) The private data for the PTF component procedures.

   Pivot does not need any private data.

This leads to the following skeleton DDL:

```
CREATE FUNCTION Pivot (
    Input TABLE PASS THROUGH WITH ROW SEMANTICS,
    Output_pivot_columns DESCRIPTOR,
    Input_pivot_columns1 DESCRIPTOR,
    Input_pivot_columns2 DESCRIPTOR DEFAULT NULL,
    Input_pivot_columns3 DESCRIPTOR DEFAULT NULL,
    Input_pivot_columns4 DESCRIPTOR DEFAULT NULL,
    Input_pivot_columns5 DESCRIPTOR DEFAULT NULL
) RETURNS TABLE
DETERMINISTIC
READS SQL DATA
DESCRIBE WITH PROCEDURE Pivot_describe
FULFILL WITH PROCEDURE Pivot_fulfill
```

12.3.4 Pivot component procedures

The DBMS tool should generate something like this:

```
CREATE PROCEDURE Pivot_describe (
    IN Input_row_descr VARCHAR(2),
    IN Input_request_descr VARCHAR(2),
    IN Output_pivot_columns_descr VARCHAR(2),
    IN Input_pivot_columns1_descr VARCHAR(2),
    IN Input_pivot_columns2_descr VARCHAR(2),
    IN Input_pivot_columns3_descr VARCHAR(2),
    IN Input_pivot_columns4_descr VARCHAR(2),
    IN Input_pivot_columns5_descr VARCHAR(2),
    IN Initial_result_row VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL DETERMINISTIC CONTAINS SQL
SQL SECURITY DEFINER
BEGIN
END
```



```
CREATE PROCEDURE Pivot_fulfill (
    IN Input_cursor_row VARCHAR(2),
    IN Input_cursor_name VARCHAR(2),
    IN Output_pivot_columns_descr VARCHAR(2),
    IN Input_pivot_columns1_descr VARCHAR(2),
    IN Input_pivot_columns2_descr VARCHAR(2),
    IN Input_pivot_columns3_descr VARCHAR(2),
    IN Input_pivot_columns4_descr VARCHAR(2),
    IN Input_pivot_columns5_descr VARCHAR(2),
    IN Intermediate_result_row VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL DETERMINISTIC READS SQL DATA
SQL SECURITY DEFINER
BEGIN
END
```

12.3.5 Invoking pivot

The DDL visible to the query author is this:

```
CREATE FUNCTION Pivot (
    Input TABLE PASS THROUGH WITH ROW SEMANTICS,
    Output_pivot_columns DESCRIPTOR,
    Input_pivot_columns1 DESCRIPTOR,
    Input_pivot_columns2 DESCRIPTOR DEFAULT NULL,
    Input_pivot_columns3 DESCRIPTOR DEFAULT NULL,
    Input_pivot_columns4 DESCRIPTOR DEFAULT NULL,
    Input_pivot_columns5 DESCRIPTOR DEFAULT NULL
) RETURNS TABLE
```

Here is an example of an invocation of Pivot:

```
SELECT D.Id, D.Name, P.Phonetype, P. Phonenumber
FROM TABLE (Pivot ( Input => TABLE (Joe.Data) AS D,
                    Output_pivot_columns => DESCRIPTOR (Phonetype, Phonenumber),
                    Input_pivot_columns1 => DESCRIPTOR (Phtype1, Phnumber1),
                    Input_pivot_columns2 => DESCRIPTOR (Phtype2, Phnumber2)
            ) ) AS P
```

To succeed, Joe.Data must be a table having columns called Id, Name, Phtype1, Phnumber1, Phtype2, and Phnumber2. The third, fourth, and fifth set of pivot columns are unused; these will default to null values.

12.3.6 Calling Pivot_describe

To compile the query, the DBMS calls Pivot_describe. The signature of the describe component procedure (see Subclause 12.3.4, “Pivot component procedures”) is as follows:

```
    PROCEDURE Pivot_describe (
    IN Input_row_descr VARCHAR(2),
    IN Input_request_descr VARCHAR(2),
    IN Output_pivot_columns_descr VARCHAR(2),
    IN Input_pivot_columns1_descr VARCHAR(2),
```

```
    IN Input_pivot_columns2_descr VARCHAR(2),
    IN Input_pivot_columns3_descr VARCHAR(2),
    IN Input_pivot_columns4_descr VARCHAR(2),
    IN Input_pivot_columns5_descr VARCHAR(2),
    IN Initial_result_row VARCHAR(2),
    INOUT Status CHAR(5)
)
```

The query has supplied Joe.Data as the input table. There are two descriptors associated with this table: the full row type, and the requested row type. The full row type descriptor describes every column of the input table. Let us suppose that the DBMS calls it I1. Let us also suppose that Joe.Data has the following columns:

```
TABLE Joe.Data (
    Id INTEGER PRIMARY KEY,
    Name VARCHAR(30),
    Phtype1 VARCHAR(5),
    Phnumber1 VARCHAR(15),
    Phtype2 VARCHAR(5),
    Phnumber2 VARCHAR(15)
)
```

This row type has the following PTF descriptor area (called I1):

	Content
Header	COUNT = 6 TOP_LEVEL_COUNT = 6 Other components unspecified
Item 1	NAME = 'ID' LEVEL = 0 TYPE = 4 (for INTEGER) Other components unspecified
Item 2	NAME = 'NAME' LEVEL = 0 TYPE = 12 (for VARCHAR) LENGTH = 30 CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the catalog, schema, and local name of the default character set. Other components unspecified
Item 3	NAME = 'PHTYPE1' LEVEL = 0 TYPE = 12 (for VARCHAR) LENGTH = 5 CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the catalog, schema, and local name of the default character set Other components unspecified

	Content
Item 4	NAME = 'PHNUMBER1' LEVEL = 0 TYPE = 12 (for VARCHAR) LENGTH = 15 CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the catalog, schema, and local name of the default character set Other components unspecified
Item 5	NAME = 'PHTYPE2' LEVEL = 0 TYPE = 12 (for VARCHAR) LENGTH = 5 CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the catalog, schema, and local name of the default character set Other components unspecified
Item 6	NAME = 'PHNUMBER2' LEVEL = 0 TYPE = 12 (for VARCHAR) LENGTH = 15 CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the catalog, schema, and local name of the default character set Other components unspecified

The DBMS must also create an empty descriptor area for the requested row type; let us suppose that the DBMS calls it A1. An empty descriptor area looks like this:

	Content
Header	COUNT = 0 TOP_LEVEL_COUNT = 0 Other components unspecified

There are three query-specified PTF descriptor areas; let them be named Q1, Q2, and Q3. The first (named Q1) is for this argument in the query invocation of Pivot:

```
Output_pivot_columns => DESCRIPTOR (Phonetype, Phonenumber),
```

The contents of Q1 are:

	Content
Header	COUNT = 2 TOP_LEVEL_COUNT = 2 Other components unspecified
Item 1	NAME = 'PHONETYPE' LEVEL = 0 TYPE = 0 Other components unspecified
Item 2	NAME = 'PHONENUMBER' LEVEL = 0 TYPE = 0

The second (named Q2) is for this argument in the invocation of Pivot:

```
Input_pivot_columns1 => DESCRIPTOR (Phtype1, Phnumber1)
```

The contents of Q2 are:

	Content
Header	COUNT = 2 TOP_LEVEL_COUNT = 2 Other components unspecified
Item 1	NAME = 'PHTYPE1' LEVEL = 0 TYPE = 0 Other components unspecified
Item 2	NAME = 'PHNUMBER1' LEVEL = 0 TYPE = 0

The third (named Q3) is for this argument in the invocation of Pivot:

```
Input_pivot_columns2 => DESCRIPTOR (Phtype2, Phnumber2)
```

The contents of Q3 are:

	Content
Header	COUNT = 2 TOP_LEVEL_COUNT = 2 Other components unspecified

	Content
Item 1	NAME = 'PHTYPE2' LEVEL = 0 TYPE = 0 Other components unspecified
Item 2	NAME = 'PHNUMBER2' LEVEL = 0 TYPE = 0

The DBMS must also allocate an empty read-write PTF descriptor area for the initial result row type. Let the initial result row type PTF descriptor area be named R. R has the following contents:

	Content
Header	COUNT = 0 TOP_LEVEL_COUNT = 0 Other components unspecified

Finally the DBMS must allocate a CHAR(5) variable for the status code, initialized to '00000'. Let ST be the status code variable.

Now the DBMS makes the following invocation:

```
CALL Pivot_describe (
    Input_row_descr => 'I1',
    Input_request_descr => 'A1',
    Output_pivot_columns_descr => 'Q1',
    Input_pivot_columns1_descr => 'Q2',
    Input_pivot_columns2_descr => 'Q3',
    Input_pivot_columns3_descr => NULL,
    Input_pivot_columns4_descr => NULL,
    Input_pivot_columns5_descr => NULL,
    Initial_result_row => 'R',
    Status => ST
)
```

12.3.7 Inside Pivot_describe

The tasks of Pivot_describe are:

1) To validate the input arguments.

2) To populate the requested row type descriptor (named A1).

3) To populate the initial result row type descriptor (named R).

In our example, the input table has the following row type:

```
TABLE Joe.Data (
    ID INTEGER PRIMARY KEY,
    Name VARCHAR(30),
    Phtype1 VARCHAR(5),
    Phnumber1 VARCHAR(15),
    Phtype2 VARCHAR(5),
    Phnumber2 VARCHAR(15),
)
```

The query author has written the following query:

```
SELECT D.Id, D.Name, P.Phonetype, P. Phonenumber
FROM TABLE ( Pivot ( Input => TABLE (Joe.Data) AS D,
                     Output_pivot_columns => DESCRIPTOR (Phonetype, Phonenumber),
                     Input_pivot_columns1 => DESCRIPTOR (Phtype1, Phnumber1),
                     Input_pivot_columns2 => DESCRIPTOR (Phtype2, Phnumber2)
           ) ) AS P
```

To satisfy the query, the PTF will need to read the columns Phtype1, Phnumer1, Phtype2, and Phnumber2; these are the columns that Pivot_describe must request by placing their names in the requested row type descriptor.

The columns of the initial result row type are:

```
PHONETYPE VARCHAR(5),
PHONENUMBER VARCHAR(15)
```

Note that the PTF is not responsible for placing any columns of the input table in the result row; this will be handled using pass-through columns. Thus, the PTF must only describe the two columns PHONETYPE and PHONENUMBER.

The logic might look like this:

1) Copy the column names from Input_pivot_columns1, ... Input_pivot_columns5 to Input_request_descr. This can be done by looking at each of Input_pivot_columns1 through Input_column_descr5 in turn. If the argument is null, there is nothing to do. Otherwise, get the number of columns and, in a loop, copy each column name, appending it to Input_request_descr with logic like this:

   ```
   SET rcol = rcol + 1;
   COPY DESCRIPTOR PTF Input_pivot_columns1 VALUE icol (NAME)
                TO PTF Input_request_row VALUE (rcol)
   ```

   Note that it is only necessary to set the name component in the requested row type descriptor, since these must all be columns of the input table and the DBMS already knows their data types.

2) Copy the PTF descriptor area whose name is passed in Output_pivot_columns_descr to the PTF area descriptor whose name is passed in Initial_result_row:

   ```
   COPY DESCRIPTOR PTF Output_pivot_columns_descr
                TO PTF Initial_result_row
   ```

   Note that the source PTF descriptor area only has the column names, so it is still necessary to set the column data types.

3) Determine the type of each result column as the union type of all corresponding columns in the PTF descriptor areas named by arguments Input_pivot_columns1_descr through Input_pivot_columns5_descr.

(Computing the union type for the general case can require some elaborate logic, so the PTF author might require that the pivot columns have the same type. The query author can work around this limitation by using casts to massage the input table.)

12.3.8 Result of Pivot_describe

Pivot_describe populates the requested row type descriptor, named A1, as follows:

	Content
Header	COUNT = 4 TOP_LEVEL_COUNT = 4 Other components unspecified
Item 1	NAME = 'PHTYPE1' LEVEL = 0 TYPE = 0 Other components unspecified
Item 2	NAME = 'PHNUMBER1' LEVEL = 0 TYPE = 0
Item 3	NAME = 'PHTYPE2' LEVEL = 0 TYPE = 0 Other components unspecified
Item 4	NAME = 'PHNUMBER2' LEVEL = 0 TYPE = 0

Pivot_describe populates the PTF descriptor area for the initial result row type, named R, as follows:

	Content
Header	COUNT = 2 TOP_LEVEL_COUNT = 2 Other components unspecified

	Content
Item 1	NAME = 'PHONETYPE' LEVEL = 0 TYPE = 12 (for VARCHAR) LENGTH = 5 CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the catalog, schema, and local name of the default character set Other components unspecified
Item 2	NAME = 'PHONENUMBER' LEVEL = 0 TYPE = 12 (for VARCHAR) LENGTH = 15 CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the catalog, schema, and local name of the default character set Other components unspecified

The columns of the result have correlation name P, so the row type can be portrayed like this:

Correlation name	Column name	Data type
D	ID	INTEGER
	NAME	VARCHAR(30)
	PHTYPE1	VARCHAR(30)
	PHNUMBER1	VARCHAR(15)
	PHTYPE2	VARCHAR(30)
	PHNUMBER2	VARCHAR(15)
P	PHONETYPE	VARCHAR(30)
	PHONENUMBER	VARCHAR(15)

Note that all columns of the input table are accessible using correlation name D. The example query has only asked for ID and NAME.

12.3.9 Virtual processors for Pivot

Pivot has one input table with row semantics. The DBMS can create an arbitrary number of virtual processors, and partition the input table arbitrarily among the virtual processors.

Prior to starting the virtual processors, the DBMS can do the following:

1) Determine the row type of the cursor, which can be described using this <cursor specification>:

```
SELECT PHTYPE1, PHNUMBER1, PHTYPE2, PHNUMBER2,
       EncodeSurrogate (ID, NAME) AS "$surr1"
FROM Joe.Data
```

   Note that this is the requested row type with one additional column, the pass-through input surrogate column. Here, EncodeSurrogate is an implementation-dependent function that encodes the columns ID and NAME in the pass-through input surrogate column named "$surr1". In this example the DBMS only needs to represent ID and NAME in the surrogate because those are the only columns of the table argument that the query asks for. The DBMS can pass a descriptor of this row type to each virtual processor.

2) Determine the intermediate result row type; this is the initial result row type plus one additional column, the pass-through output surrogate column. The data type and name of the output surrogate must be the same as for the input surrogate.

On each virtual processor, the DBMS does the following initialization:

1) The DBMS opens a PTF dynamic cursor that reads the partition assigned to that virtual processor; suppose that the PTF extended name of the cursor is CN (the same PTF extended name can be used on all virtual processors because each has its own address space).

2) The DBMS creates the requisite descriptors. The descriptors that were supplied by the query are the same as they were for Pivot_describe. We will assume that they have the same PTF extended names as they did during Pivot_describe (though this is not necessary). The cursor row descriptor and the intermediate result row descriptor are determined by the DBMS prior to starting the virtual processor; we assume that they are named 'CR' and 'MR' respectively.

3) Pivot has no private data to allocate on any virtual processor.

4) The DBMS must however allocate memory on each virtual processor for the SQL status code, a CHAR(5) variable initialized to '00000'. We portray this status code as a variable named ST.

12.3.10 Calling Pivot_fulfill

After initializing a virtual processor, the DBMS is ready to invoke Pivot_fulfill as follows:

```
CALL Pivot_fulfill (
    IN Input_cursor_row => 'CR',
    IN Input_cursor_name => 'CN',
    Output_pivot_columns_descr => 'Q1',
    Input_pivot_columns1_descr => 'Q2',
    Input_pivot_columns2_descr => 'Q3',
    Input_pivot_columns3_descr => NULL,
    Input_pivot_columns4_descr => NULL,
    Input_pivot_columns5_descr => NULL,
```

```
    Intermediate_result_row => 'MR',
    Status => ST
)
```

12.3.11 Inside Pivot_fulfill

The task of Pivot_fulfill is to process the rows of the input table and generate the output rows. Each input row results in multiple output rows. This task is distributed over the virtual processors, which each see a partition of the input table.

The logic of Pivot_fulfill might be:

1) Initialization:
   a) Locate the pass-through input surrogate column in the cursor row type descriptor. It is always the last column.
   b) Locate the pass-through output surrogate column in the intermediate result row descriptor. Since this example has only one pass-through table, the surrogate is the last column of the intermediate result row type.
2) Fetch a row of the input cursor, like this:

```
FETCH FROM PTF Input_cursor_name
      INTO DESCRIPTOR PTF Input_cursor_row;
```

3) If the FETCH encounters the end of the cursor, return with success in the status code.
4) If FETCH encounters an error, set the Status argument to that error code and return.
5) Copy the pass-through input surrogate column from the input cursor row to the pass-through output surrogate column in the intermediate result row.
6) If Input_pivot_columns1 is not null, then copy the input columns listed in this PTF descriptor area to the corresponding columns of the result PTF descriptor area.
7) Send the result row to the DBMS with:

```
PIPE ROW PTF Intermediate_result_row;
```

8) Process Input_pivot_columns 2, Input_pivot_columns3, ..., Input_pivot_columns5 the same way. Each one that is not null causes a separate output row from the same input row.
9) Loop back to step 2.

12.3.12 Collecting the results

The DBMS collects the result rows that are sent via PIPE ROW commands on all virtual processors. For each row, the DBMS must expand the output pass-through surrogate column to recover the values of ID and NAME. The union of these rows constitutes the overall result of the invocation of Pivot.

12.3.13 Cleanup

After a virtual processor completes, the DBMS closes the input cursor, deallocates its data structures (such as PTF descriptor areas), and terminates the virtual processor.

12.4 Score

This example was introduced in Subclause 3.2.3, “Score”.

12.4.1 Overview

Score has two input tables:

1) One input table contains rows to be scored according to some algorithm.

2) The other input table contains the parameters for the algorithm that is used to score a row.

The output of Score is the score computed for each input row according to the scoring algorithm as parameterized by the parameter table. The original input row is available as well, via the pass-through mechanism, as is the partitioning column of the parameter table.

12.4.2 Functional specification of Score

Score has two input tables:

1) One input table contains rows to be scored according to some algorithm. This table has row semantics since each row can be processed separately. The PTF will make the contents of an input row available to the query using the pass-through mechanism.

2) The other input table contains the parameters for the algorithm that is used to score a row. This table has set semantics since the entire table is required to specify the algorithm. An empty table does not specify an algorithm, so this input table is prunable (there is no result if the input table is empty). Since a result is not associated with a particular row of this table, the PTF does not provide pass-through columns for this input table.

```
CREATE FUNCTION Score (
    Data TABLE PASS THROUGH WITH ROW SEMANTICS,
    Model TABLE NO PASS THROUGH
      WITH SET SEMANTICS PRUNE WHEN EMPTY
) RETURNS TABLE (Score REAL)
DETERMINISTIC
READS SQL DATA
```

For each row of the first input table, Score uses the model supplied by the second input table to compute a value in a column called Score of type REAL. Since the proper result column is fixed, it can be specified in the CREATE FUNCTION statement as shown above.

12.4.3 Design specification for Score

The design specification specifies details that are private, that is, not visible to the query author. For the design specification, the PTF author decides:

1) Whether PTF start and/or finish component procedures are required.

   Score does not need any resources not provided by the DBMS, so there are no start or finish component procedures.

2) The names of the PTF component procedures.

   The PTF author decides to name the PTF component procedures Score_describe and Score_fulfill.

3) The private data for the PTF component procedures.

   Score does not need any private data.

This leads to the following skeleton DDL:

```
CREATE FUNCTION Score (
    Data TABLE PASS THROUGH WITH ROW SEMANTICS,
    Model TABLE NO PASS THROUGH
             WITH SET SEMANTICS PRUNE WHEN EMPTY
) RETURNS TABLE (Score REAL)
DETERMINISTIC
READS SQL DATA
DESCRIBE WITH PROCEDURE Score_describe
FULFILL WITH PROCEDURE Score_fulfill
```

12.4.4 Score component procedures

The DBMS tool should generate something like this:

```
CREATE PROCEDURE Score_describe (
    IN Data_row_descr VARCHAR(2),
    IN Data_request_descr VARCHAR(2),
    IN Model_row_descr VARCHAR(2),
    IN Model_pby_descr VARCHAR(2),
    IN Model_order_descr VARCHAR(2),
    IN Model_request_descr VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL DETERMINISTIC CONTAINS SQL
SQL SECURITY DEFINER
BEGIN
END

CREATE PROCEDURE Score_fulfill (
    IN Data_cursor_descr VARCHAR(2),
    IN Data_cursor_name VARCHAR(2),
    IN Model_cursor_descr VARCHAR(2),
    IN Model_pby_descr VARCHAR(2),
    IN Model_order_descr VARCHAR(2),
    IN Model_cursor_name VARCHAR(2),
    IN Intermediate_result_row VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL DETERMINISTIC READS SQL DATA
SQL SECURITY DEFINER
BEGIN
END
```

NOTE 3 —

1) The first table parameter has row semantics; therefore, it requires the following parameters in the component procedures:

   a) In the describe component procedure, the full row type (Data_row_descr), and the requested row type (Data_request_descr).

   b) In the fulfill component procedure, the cursor row type (Data_cursor_descr), and the cursor name (Data_cursor_name).

2) The second table parameter has set semantics, so it requires the following parameters in the component procedures:

   a) In the describe component procedure, the full row type (Model_row_descr), the partitioning (Model_pby_descr), the ordering (Model_order_descr), and the requested row type (Model_request_descr).

   b) In the fulfill component procedure, the cursor row type (Model_cursor_descr), the partitioning (Model_pby_descr), the ordering (Model_order_descr), and the cursor name (Model_cursor_name).

12.4.5 Invoking Score

The DDL visible to the query author is:

```
CREATE FUNCTION Score (
    Data TABLE PASS THROUGH WITH ROW SEMANTICS,
    Model TABLE NO PASS THROUGH
             WITH SET SEMANTICS PRUNE WHEN EMPTY
) RETURNS TABLE (Score REAL)
```

Here is the example invocation initially shown in Subclause 3.2.3, “Score”:

```
SELECT D.Id, D.S, D.T, M.Modelid, T.Score
FROM TABLE ( Score ( Data => TABLE (MyData) AS D
                     Model => TABLE (Models) AS M
                                PARTITION BY Modelid )
           ) AS T
```

12.4.6 Calling Score_describe

The signature of Score_describe, previously shown in Subclause 12.4.4, “Score component procedures”, is:

```
PROCEDURE Score_describe (
    IN Data_row_descr VARCHAR(2),
    IN Data_request_descr VARCHAR(2),
    IN Model_row_descr VARCHAR(2),
    IN Model_pby_descr VARCHAR(2),
    IN Model_order_descr VARCHAR(2),
    IN Model_request_descr VARCHAR(2),
    INOUT Status CHAR(5)
)
```

Before calling Score_describe, the DBMS must create PTF descriptor areas for the first six input parameters. As originally presented in Subclause 3.2.3, “Score”, the first input table Data has this row type: (ID INTEGER, S REAL, T REAL). Therefore, the DBMS can create a PTF descriptor area for the full row type (let us call it 'I1') as follows:

	Content
Header	COUNT = 3 TOP_LEVEL_COUNT = 3 Other components unspecified
Item 1	NAME = 'ID' LEVEL = 0 TYPE = 4 (for INTEGER) Other components unspecified
Item 2	NAME = 'S' LEVEL = 0 TYPE = 7 (for REAL) Other components unspecified
Item 3	NAME = 'T' LEVEL = 0 TYPE = 7 (for REAL) Other components unspecified

The DBMS also needs to create an empty PTF descriptor area for the requested row type of Data; let us call it 'A1'.

	Content
Header	COUNT = 0 TOP_LEVEL_COUNT = 0 Other components unspecified

The table called Models has this row type: (MODELID VARCHAR(10), PNAME VARCHAR(10), PVALUE REAL). The DBMS creates a PTF descriptor area (call it 'I2') as follows:

	Content
Header	COUNT = 3 TOP_LEVEL_COUNT = 3 Other components unspecified
Item 1	NAME = 'MODELID' LEVEL = 0 TYPE = 12 (for VARCHAR) LENGTH = 10 CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the catalog, schema, and local name of the default character set. Other components unspecified

	Content
Item 2	NAME = 'PNAME' LEVEL = 0 TYPE = 12 (for VARCHAR) LENGTH = 10 CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the catalog, schema, and local name of the default character set. Other components unspecified
Item 3	NAME = 'PVALUE' LEVEL = 0 TYPE = 7 (for REAL) Other components unspecified

The MODELS table is partitioned on MODELID. The DBMS creates a PTF descriptor area (call it 'P2') of the partitioning, listing just the names of the partitioning columns, as follows:

	Content
Header	COUNT = 1 TOP_LEVEL_COUNT = 0 Other components unspecified
Item 1	NAME = 'MODELID' LEVEL = 0 Other components unspecified

The MODELS table is unordered, so the DBMS creates an empty PTF descriptor area for this ordering (call it 'S2'):

	Content
Header	COUNT = 0 TOP_LEVEL_COUNT = 0 Other components unspecified

The DBMS must also create an empty PTF descriptor area for the requested row type of Models; let us call it 'A2'.

The proper result columns have been declared in the DDL as (Score REAL), which can be described in the initial result row type descriptor like this:

	Content
Header	COUNT = 1 TOP_LEVEL_COUNT = 0 Other components unspecified
Item 1	NAME = 'SCORE' LEVEL = 0 TYPE = 7 (for REAL) Other components unspecified

Since the initial result row type is fixed, this descriptor is not passed to the describe component procedure, and does not really need to be constructed.

Finally, the DBMS creates a variable for the status code, a CHAR(5) value initialized to '00000'; let us call it ST.

After creating and initializing the preceding, the DBMS is ready to call Score_describe like this:

```
CALL Score_describe (
    Data_row_descr => 'I1'
    Data_request_descr => 'A1'
    Model_row_descr => 'I2',
    Model_pby_descr => 'P2',
    Model_order_descr => 'S2',
    Model_request_descr => 'A2',
    Status => ST
);
```

12.4.7 Inside Score_describe

The tasks for Score_describe are:

1) Validate the input.

   Our example has not mentioned any requirements on the input data and model tables, though an actual implementation would have some. Any static requirements, such as column names and types, can be checked by using GET DESCRIPTOR, returning an error condition if the requirements are not met. Requirements based on the contents of the model data cannot be checked until run-time since there is no cursor open to read the data.

2) Request columns for the input tables by populating their requested row descriptors.

   In this example, let us assume that the PTF requests columns S and T from Data, and columns Pname and Pvalue from Models. Therefore, Score_describe must populate Data_request_descr like this:

	Content
Header	COUNT = 2 TOP_LEVEL_COUNT = 2 Other components unspecified
Item 1	NAME = 'S' LEVEL = 0 Other components unspecified
Item 2	NAME = 'T' LEVEL = 0 Other components unspecified

and populate Model_request_descr like this:

	Content
Header	COUNT = 2 TOP_LEVEL_COUNT = 2 Other components unspecified
Item 1	NAME = 'PNAME' LEVEL = 0 Other components unspecified
Item 2	NAME = 'PVALUE' LEVEL = 0 Other components unspecified

These descriptors can be set using techniques discussed in Subclause 7.4, “Writing a PTF descriptor area”.

12.4.8 Result of Score_describe

The DBMS first checks the status code variable for success; if not, the query has a syntax error.

Otherwise, the DBMS saves the requested row type descriptors for use at run-time (they will be used to construct the cursor row types later).

The DBMS can also save the initial result row type descriptor, or wait till run-time to build the intermediate result row type (this information is already saved in the metadata for Score, since it was declared in DDL).

At this point the DBMS can determine the complete result row type, since that is needed to finish analyzing the query. The query author has written the following query (initially presented in Subclause 3.2.3, “Score”):

```
SELECT D.Id, D.S, D.T, M.Modelid, T.Score
FROM TABLE ( Score ( Data => TABLE (MyData) AS D
             Model => TABLE (Models) AS M
```



```
                    PARTITION BY Modelid )
          ) AS T
```

The result row type of the PTF invocation has three correlation names: D, M, and T. Using D, the query can access all columns of the first input table, since it has pass-through columns. Using M, the query can access the partitioning column, Modelid, of the second input table. Finally, using T, the query can access the proper result column computed by the PTF, in the column called SCORE. Thus the row type of the PTF invocation looks like this:

Correlation name	D			M	T
Column name	ID	S	T	MODELID	SCORE
Data type	INTEGER	REAL	REAL	VARCHAR(10)	REAL

12.4.9 Virtual processors for Score

The query author has written the following query (initially presented in Subclause 3.2.3, “Score”:

```
SELECT D.Id, D.S, D.T, M.Modelid, T.Score
FROM TABLE ( Score ( Data => TABLE (MyData) AS D
                     Model => TABLE (Models) AS M
                              PARTITION BY Modelid )
          ) AS T
```

This example has one input table with row semantics and one with set semantics. The latter is partitioned. The sample data for the second input table is:

Modelid	Pname	Pvalue
wet	x	19
wet	y	28
wet	z	37
dry	x	4
dry	y	5
dry	z	6

This has two partitions when partitioned on Modelid, “wet” and “dry”.

The DBMS must ensure that each partition is used to score each row of the first input table (since that table has row semantics). This might be done by creating a virtual processor for each partition and “broadcasting” the entire first input table to each virtual processor. It could also be done by subdividing the Data input table arbitrarily within each partition of Models. For example, the sample data for the first input table is :

Id	S	T
122	0.5	3.4
233	8.4	6.5
344	10.2	9.3
455	11.0	8.8

Then the DBMS might create four virtual processors, with cursors to read each input table as follows:

Virtual processor	Cursor for Data table	Cursor for Models table
1	SELECT ... FROM MyData WHERE ID = 122 OR ID = 233	SELECT ... FROM Models WHERE Modelid = 'wet'
2	SELECT ... FROM MyData WHERE ID = 344 OR ID = 455	SELECT ... FROM Models WHERE Modelid = 'wet'
3	SELECT ... FROM MyData WHERE ID = 122 OR ID = 455	SELECT ... FROM Models WHERE Modelid = 'dry'
4	SELECT ... FROM MyData WHERE ID = 233 OR ID = 344	SELECT ... FROM Models WHERE Modelid = 'dry'

(We have omitted the SELECT lists for now, which will be determined later.)

Virtual processors 1 and 2 handle the “wet” model, whereas virtual processors 3 and 4 handle the “dry” model. The rows of MyData are partitioned arbitrarily for the “wet” model, and arbitrarily for the “dry” model. Note that the same partitioning of MyData is not used in each model. This is a freedom that the DBMS has; however, the DBMS might also choose to use the same partitioning of MyData in each model.

Before starting the virtual processors, the DBMS can compute the following descriptors:

1) The cursor row type for MyData. Score_describe has requested columns named S and T. This input table has pass-through columns, so the DBMS adds a pass-through input surrogate column; let us suppose it is named "$surr1". Thus, the SELECT list for the cursors for MyData in every partition is SELECT S, T, "$surr1".

2) The cursor row type for Models. Score_describe has requested columns named Pname and Pvalue. This input table does not have pass-through columns; therefore, the SELECT list for the cursors for Models is SELECT Pname, Pvalue.

3) The partitioning and ordering descriptors for Models; these are the same as were input to Score_describe.

4) The intermediate result row type. This has two columns: the proper result columns declared in DDL as (Score REAL), plus the pass-through output surrogate column named "$surr1".

On each virtual processor, the DBMS does the following initialization:

1) For each input table, the DBMS opens a PTF dynamic cursor that reads the partition assigned to that virtual processor. There are two input tables, so there are two PTF cursors. Let the PTF extended names of these cursors be 'C1' and 'C2'.

2) The DBMS creates copies of the PTF descriptor areas mentioned above, and gives them PTF extended names.

   a) Cursor row type of MyData: I1.

   b) Cursor row type of Models: I2.

   c) Partitioning of Models: P2.

   d) Ordering of Models: S2.

   e) Intermediate result row: R.

3) Score has no private data to allocate on any virtual processor.

4) The DBMS must allocate memory on each virtual processor for the SQL status code, a CHAR(5) variable initialized to '00000'. We portray this status code as a variable named ST.

12.4.10 Calling Score_fulfill

On each virtual processor, the DBMS calls Score_fulfill like this:

```
CALL Score_fulfill (
    Data_cursor_descr => 'I1',
    Data_cursor_name => 'C1',
    Model_cursor_descr => 'I2',
    Model_pby_descr => 'P2',
    Model_order_descr => 'S2',
    Model_cursor_name => 'C2',
    Intermediate_result_row => 'R',
    Status => ST
);
```

12.4.11 Inside Score_fulfill

The logic for Score_fulfill might look like this:

1) Read all of the Model table into memory by performing the following in a loop until no more rows are available:

```
FETCH FROM PTF Model_cursor
      INTO DESCRIPTOR PTF Model_cursor_descr;
```

2) Build the model detemined by the rows that are read. If there is any error, return an error code in ST.

3) Get the number of columns in the cursor for MyData:

```
GET DESCRIPTOR PTF Data_cursor_descr
    Ncols = TOP_LEVEL_COUNT;
```

Note that the first (Ncols–1) columns are the requested data, and the last column (index Ncols) is the pass-through input surrogate column.

4) In a loop until the Data table is exhausted:

a) Read a row of the Data table:

```
FETCH FROM PTF Data_cursor
      INTO DESCRIPTOR PTF Data_cursor_descr;
```

b) Using the data model, compute the score in S.

c) Place the score in the result row:

```
SET DESCRIPTOR PTF Intermediate_result_row
    VALUE = 1 DATA = S;
```

d) Copy the pass-through input surrogate column to the Pass-through output surrogate column:

```
COPY DESCRIPTOR PTF Data_cursor_descr VALUE Ncols (DATA)
             TO PTF Intermediate_result_row VALUE 2;
```

e) Pipe the row to the DBMS:

```
PIPE ROW PTF Intermediate_result_row;
```

12.4.12 Collecting the output

On each virtual processor, the DBMS collects the output rows that are sent via PIPE ROW statements from Score_fulfill. Note that the PIPE ROW command only sends two columns to the DBMS (Score and "$surr1"), but the complete result row type has the following columns: D.Id, D.S, D.T, M.Modelid, and T.Score. The DBMS assembles the complete result row from the intermediate result row as follows:

1) D.Id, D.S and D.T are obtained by expanding the pass-through output surrogate column "$surr1".

2) M.Modelid is the partitioning key, which is an invariant on the virtual processor.

3) T.Score is derived from Score in the intermediate result row.

The union of the complete result rows is the result of the invocation of Score.

12.4.13 Cleanup

When a virtual processor completes, the DBMS does cleanup tasks such as closing the input cursors and deallocating the PTF descriptor areas.

12.5 TopNplus

This example was introduced in Subclause 3.2.4, “TopNplus”.

12.5.1 Overview

TopNplus takes an input table that has been sorted on a numeric column. It copies the first n rows through to the output table. (However, any partitioning columns are not copied, since those are available to the query through the range variable for the input table.) Any additional rows are summarized in a single output row in which the sort column has been summed and all other columns are null.

12.5.2 Functional specification of TopNplus

TopNplus requires two input parameters:

1) An input table. Since the algorithm is defined on a set of rows, this input table has set semantics. The algorithm could reasonably be specified to produce no rows on empty input, or it could produce a single summary row with a total of 0. We will show the PRUNE WHEN EMPTY choice.

2) The number of rows to be copied from input to output. (Any remaining input rows will be summarized in a single output row.)

The PTF author can write the following public DDL (visible to the query author via the Information Schema):

```
CREATE FUNCTION TopNplus (
    Input TABLE NO PASS THROUGH
      WITH SET SEMANTICS PRUNE WHEN EMPTY,
    Howmany INTEGER )
RETURNS TABLE
NOT DETERMINISTIC
READS SQL DATA
```

TopNplus is not deterministic, because there may be ties when an input partition is sorted. If a set of ties overlaps the cutoff specified by Howmany, then it is not deterministic which rows will be copied to the output and which rows will be summarized.

12.5.3 Design specification for TopNplus

The design specification specifies details that are private, that is, not visible to the query author. For the design specification, the PTF author decides:

1) Whether PTF start and/or finish component procedures are required.

   TopNplus does not need any resources not provided by the DBMS, so there are no start or finish component procedures.

2) The names of the PTF component procedures.

The PTF author decides to name the PTF component procedures TopNplus_describe and TopNplus_fulfill.

3) The private data for the PTF component procedures.

TopNplus will require that the input be ordered on a single column, which must be numeric. The describe component procedure will locate this column. The fulfillment component procedure will need to know which column is the ordering column. The fulfillment component procedure could figure this out on its own, but since the describe component procedure must look for it anyway, the describe component procedure can save the value to a private variable, thereby passing it to the fulfill component procedure.

Based on these decisions, we have the following skeleton DDL:

```
CREATE FUNCTION TopNplus (
    Input TABLE NO PASS THROUGH
            WITH SET SEMANTICS PRUNE WHEN EMPTY,
    Howmany INTEGER )
RETURNS TABLE
PRIVATE DATA (
    Order_col_no INTEGER )
NOT DETERMINISTIC
READS SQL DATA
DESCRIBE WITH PROCEDURE TopNplus_describe
FULFILL WITH PROCEDURE TopNplus_fulfill
```

12.5.4 TopNplus component procedures

The DBMS tool should generate something like this:

```
CREATE PROCEDURE TopNplus_describe (
    INOUT Order_col_no INTEGER;
    IN Input_row_descr VARCHAR(2),
    IN Input_pby_descr VARCHAR(2)
    IN Input_order_descr VARCHAR(2),
    IN Input_request_descr VARCHAR(2),
    IN Howmany INTEGER,
    IN Initial_result_row VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL DETERMINISTIC CONTAINS SQL
SQL SECURITY DEFINER
BEGIN
END

CREATE PROCEDURE TopNplus_fulfill (
    INOUT Order_col_no INTEGER;
    IN Input_row_descr VARCHAR(2),
    IN Input_pby_descr VARCHAR(2)
    IN Input_order_descr VARCHAR(2),
    IN Input_cursor VARCHAR(2),
    IN Howmany INTEGER,
    IN Intermediate_result_row VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL NOT DETERMINISTIC READS SQL DATA
SQL SECURITY DEFINER
```

```
BEGIN
END
```

NOTE 4 —

1) The parameter lists begin with the private data (Order_col_no).

2) Next come the parameters corresponding to the input table.

   a) For the describe component procedure, four PTF descriptor areas are required, one for the full row type (Input_row_type), one for the partitioning columns (Input_pby_descr), one for the ordering (Input_order_descr), and one for the requested row type (Input_request_descr). In the fulfill component procedure, there is also a parameter for the cursor (Input_cursor).

   b) For the fulfill component procedure, three PTF descriptor areas are required, one for the cursor row type (Input_row_type), one for the partitioning columns (Input_pby_descr), and one for the ordering (Input_order_descr). There is also a parameter for the cursor name (Input_cursor).

3) Next comes the scalar parameter Howmany, which is copied from the signature of TopNplus.

4) Next is the parameter for the PTF descriptor area for the result row type (called Initial_result_row in the describe component procedure and intermediate result row in the fulfill component procedure).

5) Finally there is a parameter for the SQL status code.

12.5.5 Invoking TopNplus

The DDL visible to the query author is:

```
CREATE FUNCTION TopNplus (
    Input TABLE WITH SET SEMANTICS PRUNE WHEN EMPTY,
    Howmany INTEGER
) RETURNS TABLE
```

Here is an example of an invocation:

```
SELECT S.Region, T.*
FROM TABLE ( TopNplus (
                Input => TABLE (My.Sales) AS S
                           PARTITION BY Region
                           ORDER BY Sales DESC,
                Howmany => 3 )
             ) AS T
```

Note that only the partitioning column can be accessed using correlation name S, because the input table has set semantics.

12.5.6 Calling TopNplus_describe

The signature of TopNplus_describe as stated in Subclause 12.5.4, “TopNplus component procedures”, is:

```
PROCEDURE TopNplus_describe (
    INOUT Order_col_no INTEGER;
    IN Input_row_descr VARCHAR(2),
    IN Input_pby_descr VARCHAR(2)
```



```
    IN Input_order_descr VARCHAR(2),
    IN Input_request_descr VARCHAR(2),
    IN Howmany INTEGER,
    IN Initial_result_row VARCHAR(2),
    INOUT Status CHAR(5)
)
```

The input table My.Sales has the following columns:

```
CREATE TABLE My.Sales (
    Region VARCHAR(20),
    Product VARCHAR(20),
    SalesmanID INTEGER,
    Sales FLOAT )
```

TopNplus has one private variable, an integer named Order_col_no. The DBMS must allocate this and initialize it to null.

The signature of TopNplus_describe will require that the DBMS create five PTF descriptor areas. The first is the full row type PTF descriptor area. Let it be named 'I1'; it has the following contents:

	Content
Header	COUNT = 3 TOP_LEVEL_COUNT = 3 Other components unspecified
Item 1	NAME = 'REGION' LEVEL = 0 TYPE = 12 (for VARCHAR) LENGTH = 20 CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the catalog, schema, and local name of the default character set. Other components unspecified
Item 2	NAME = 'PRODUCT' LEVEL = 0 TYPE = 12 (for VARCHAR) LENGTH = 20 CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the catalog, schema, and local name of the default character set. Other components unspecified
Item 3	NAME = 'SALES' LEVEL = 0 TYPE = 7 (for REAL) Other components unspecified

Next the DBMS must also create a PTF descriptor area of the partitioning; call this P1. The contents of P1 are:

	Content
Header	COUNT = 1 TOP_LEVEL_COUNT = 1 Other components unspecified
Item 1	NAME = 'REGION' LEVEL = 0 Other components unspecified

Since the input table has set semantics, the DBMS must also create a PTF descriptor area of the ordering. Call this PTF descriptor areas S1. The contents of S1 are:

	Content
Header	COUNT = 1 TOP_LEVEL_COUNT = 1 Other components unspecified
Item 1	NAME = 'SALES' LEVEL = 0 LENGTH = 0 TYPE = 0 SORT_DIRECTION = 1 (for ASC) NULL_ORDERING = 1 (for NULLS LAST) Other components unspecified

(This assumes an implementation-defined default to NULLS LAST).

The DBMS must also create an empty PTF descriptor area for the requested row type; let it be called A1:

	Content
Header	COUNT = 0 TOP_LEVEL_COUNT = 0 Other components unspecified

The DBMS must also allocate an empty read-write PTF descriptor area for the intermediate result row type. Let the intermediate result row type PTF descriptor area be named R. R has the following contents:

	Content
Header	COUNT = 0 TOP_LEVEL_COUNT = 0 Other components unspecified

Finally the DBMS must allocate a CHAR(5) variable for the status code, initialized to '00000'. Let ST be the status code variable.

Now the DBMS makes the following invocation:

```
CALL TopNplus_describe (
    Order_col_no => Order_col_no;
    Input_row_descr => 'I1',
    Input_pby_descr => 'P1',
    Input_order_descr => 'S1',
    Input_request_descr => 'A1',
    Howmany => 3,
    Initial_result_row => 'R',
    Status => ST
)
```

12.5.7 Inside TopNplus_describe

The tasks of TopNplus are:

1) Check the inputs for validity.

   The validity check is that there is a single ordering column of numeric type.

2) Populate the requested row type descriptor.

   The requested row descriptor is a copy of the full input row descriptor.

3) Populate the initial result row type descriptor.

   The initial result row descriptor is a copy of the full input row descriptor.

4) Set the value of the private data.

TopNplus can use the following logic:

1) Get the number of ordering columns in local variable OrderColumns:

   ```
   GET DESCRIPTOR PTF Input_order_descr
     OrderColumns = COUNT;
   ```

2) If OrderColumns is not 1, return an error in the status code parameter.

3) Get the name of the ordering column in local variable OrderColumnName:

   ```
   GET DESCRIPTOR PTF Input_order_descr
     VALUE 1 OrderColumnName = NAME;
   ```

4) Get the number of input columns in InputColumns:

   ```
   GET DESCRIPTOR PTF Input_row_descr
     InputColumns = COUNT;
   ```

5) In a loop setting I from 1 to InputColumns:

a) Get the input column name in ColumnName.

```
GET DESCRIPTOR PTF Input_row_descr
  VALUE I ColumnName = NAME;
```

b) If ColumnName equals OrderColumnName, exit the loop.

6) If no input column has the same name as OrderColumnName, return an error. (If this happens, it indicates a bug in the DBMS, since the ordering column must be a column of the input table.)

7) Get the data type of the ordering column:

```
GET DESCRIPTOR PTF Input_row_descr
  VALUE I OrderType = TYPE;
```

8) Check that OrderType is a numeric type (see Subclause 7.1.2, "SQL item descriptor areas for row types", for a list of all type codes). If the ordering column is not numeric, return an error.

9) Populate the PTF descriptor area for the initial result row type as a copy of the input table's row type, minus any partitioning columns.

a) Initialize OutputColumns to 0 and Found to 0.

b) In a loop, setting J from 1 through InputColumns:

i) Get the J-th column name:

```
GET DESCRIPTOR PTF Input_row_descr
  VALUE J ColumnName = NAME
```

ii) Search through the PTF descriptor area named by Input_pby_descr, looking for a match to ColumnName. If the column name matches a partitioning column, continue the loop at the next J.

iii) If ColumnName does not match any name in Input_pby_descr, then increment OutputColumns and append the J-th item descriptor from Input_row_descr to Initial_result_row:

```
OutputColumns = OutputColumns + 1;
SET DESCRIPTOR PTF Initial_result_row COUNT = J;
COPY DESCRIPTOR PTF Input_row_descr
  VALUE J (NAME, TYPE)
              TO PTF Initial_result_row VALUE OutputColumns;
```

10) The requested row descriptor area can be a copy of the initial result row:

```
COPY DESCRIPTOR PTF Initial_result_row
  TO Input_request_descr;
```

11) Search the requested row descriptor for a column with the same name as the ordering column. If there is no such column, return an error (the ordering column must have been a partitioning column, but ordering on a partitioning column will not order a partition). Otherwise, save the column number in the private variable Order_col_no.

12.5.8 Result of TopNplus_describe

The DBMS checks the status code for success. Then the DBMS saves the private data, requested row type descriptor, and the initial result row type descriptor for use at run time.

The complete result row type is:

Correlation name	S	T	
Column name	REGION	PRODUCT	SALES
Data type	VARCHAR(100)	VARCHAR(100)	VARCHAR(100)

12.5.9 Virtual processors for TopNplus

The query (first presented in Subclause 12.5.5, “Invoking TopNplus”) is:

```
SELECT S.Region, T.*
FROM TABLE ( TopNplus (
    Input => TABLE (My.Sales) AS S
               PARTITION BY Region
               ORDER BY Sales DESC,
    Howmany => 3 )
           ) AS T
```

This example has one input table; it has set semantics and is partitioned and ordered. Therefore, the DBMS creates one virtual processor for each partition of the input table. The DBMS must sort the data in each partition.

The sample data presented in Subclause 12.5.1, “Overview”, is:

Region	**Product**	**Sales**
East	A	1234.56
East	B	987.65
East	C	876.54
East	D	765.43
East	E	654.32
West	E	2345.67
West	D	2001.33
West	C	1357.99
West	B	975.35

Region	Product	Sales
West	A	864.22

There are two partitions; therefore, the DBMS must create two virtual processors, one for region “East” and one for region “West”. The data above has been sorted on Sales in descending order; this is the order required for the cursor in each partition.

Before creating the virtual processors, the DBMS can determine the following PTF descriptor areas:

1) The cursor row type descriptor. This example does not use pass-through columns; therefore, this is the same as the requested row type descriptor that was produced by TopNplus_describe.

2) The partitioning and ordering descriptors. These are the same as were input to TopNplus_describe.

3) The intermediate result row type descriptor. Since there are no pass-through columns, this is the same as the initial result row type descriptor populated by TopNplus_describe.

The DBMS creates the virtual processors, assigning each of them a partition of the input data. On each virtual processor, the DBMS opens a cursor to read the virtual processor's partition, with row type as described by the cursor row type descriptor. Let the cursor be named 'C1' in each partition (there is no name conflict because each virtual processor is its own name space). Thus, on region “East” the cursor is:

```
SELECT Product, Sales
FROM My.Sales
WHERE Region = 'East'
ORDER BY Sales DESC
```

and the cursor of the virtual processor for region “West” is:

```
SELECT Product, Sales
FROM My.Sales
WHERE Region = 'West'
ORDER BY Sales DESC
```

The private data for TopNplus is:

```
PRIVATE DATA (
    Order_col_no INTEGER )
```

This private data was initialized by TopNplus_describe and is simply maintained by the DBMS.

On each virtual processor, the DBMS does the following initialization:

1) The DBMS opens a PTF dynamic cursor that reads the partition assigned to that virtual processor, with SELECT list determined by the cursor row type descriptor. Suppose that the PTF extended name of the cursor is C1 (the same PTF extended name can be used on all virtual processors because each is its own address space).

2) The DBMS creates a copy of the PTF descriptor areas as determined above, and gives them names in the PTF extended name space. We will assume the following names:

    a) Cursor row type descriptor: I1

    b) Partitioning descriptor: P1

c) Ordering descriptor: S1

d) Intermediate result row descriptor: R

3) The DBMS instantiates a copy of the private data as it was output from TopNplus_describe. We show it as a variable named Order_col_no (the same as the parameter name).

4) The DBMS allocates memory on each virtual processor for the SQL status code, a CHAR(5) variable initialized to '00000'. We portray this status code as a variable named ST.

12.5.10 Calling TopNplus_fulfill

The signature of TopNplus_fulfill is found in Subclause 12.5.4, "TopNplus component procedures", as follows:

```
PROCEDURE TopNplus_fulfill (
    INOUT Order_col_no INTEGER;
    IN Input_row_descr VARCHAR(2),
    IN Input_pby_descr VARCHAR(2)
    IN Input_order_descr VARCHAR(2),
    IN Input_cursor VARCHAR(2),
    IN Howmany INTEGER,
    IN Intermediate_result_row VARCHAR(2),
    INOUT Status CHAR(5)
)
```

Then the invocation looks like this:

```
CALL TopNplus_fulfill (
    Order_col_no => Order_col_no;
    Input_row_descr => 'I1',
    Input_pby_descr => 'P1',
    Input_order_descr => 'S1',
    Input_cursor => 'C1',
    Howmany => 3,
    Intermediate_result_row => 'R',
    Status => ST
)
```

12.5.11 Inside TopNplus_fulfill

1) In a loop:

a) Read at most Howmany rows of the input cursor:

```
FETCH FROM PTF Input_cursor
  INTO DESCRIPTOR PTF Input_row_descr;
```

b) Copy the nonpartitioning columns to Intermediate_result_row.

c) Output the row in Intermediate_result_row:

```
PIPE ROW PTF Intermediate_result_row;
```

2) Initialize local variable Sum to 0:

```
LET Sum = 0.0;
```

3) In a loop:

   a) Read all remaining rows of the input.

```
FETCH FROM PTF Input_cursor
      INTO DESCRIPTOR PTF Input_row_descr;
```

   b) Get the value of the sort column. Note that the position of this column in Input_row_descr was determined by TopNplus_describe and placed in Order_col_no:

```
GET DESCRIPTOR PTF Input_row_descr VALUE Order_col_no
  Val = DATA;
```

   c) Add the value to SUM:

```
LET Sum = Sum + Val;
```

4) If there were any remaining rows:

   a) Letting I range over all columns of the result, set the DATA component of the I-th column of the result row descriptor area to null:

```
SET DESCRIPTOR PTF Intermediate_result_row
  VALUE I DATA = NULL;
```

   b) Set the order column's DATA to Sum:

```
SET DESCRIPTOR PTF Intermediate_result_row
  VALUE Order_col_no DATA = Sum;
```

   c) Send the summary row to the DBMS:

```
PIPE ROW PTF Intermediate_result_row;
```

12.5.12 Collecting the output

On each virtual processor, the DBMS collects the rows that are sent from TopNplus_fulfill. The complete result row is formed from the intermediate result row (piped out of TopNplus_fulfill) plus the partitioning columns (invariant on the virtual processor). The overall result is the union of the complete result rows from each virtual processor.

12.5.13 Cleanup

After a virtual processor completes, the DBMS performs any cleanup for that virtual processor, such as closing the input cursor and deallocating the PTF descriptor areas.

12.5.14 TopNplus using pass-through columns

The preceding example can also be done using pass-through columns, if Feature B205, “Pass-through columns”, is supported by the DBMS, with a slight modification to the functional specification. The modification is that Feature B205, “Pass-through columns”, is used to provide the columns of the input table in the result. The only proper result column is a copy of the ordering column in the first Howmany rows, and the sum of the remaining rows in a summary row. Since there is no specific input row to associate with the summary row, the pass-through columns in the summary row are set to null.

The DDL becomes:

```
CREATE FUNCTION TopNplus (
    Input TABLE PASS THROUGH
            WITH SET SEMANTICS PRUNE WHEN EMPTY,
    Howmany INTEGER )
RETURNS TABLE
NOT DETERMINISTIC
READS SQL DATA
DESCRIBE WITH PROCEDURE TopNplus_describe
FULFILL WITH PROCEDURE TopNplus_fulfill
```

The differences from Subclause 12.5.3, “Design specification for TopNplus”, are:

1) PASS THROUGH instead of NO PASS THROUGH.

2) No private data (the requested row will only have the order column, so there is no problem finding it).

Consequently, the component procedure signatures are:

```
CREATE PROCEDURE TopNplus_describe (
    IN Input_row_descr VARCHAR(2),
    IN Input_pby_descr VARCHAR(2)
    IN Input_order_descr VARCHAR(2),
    IN Input_request_descr VARCHAR(2),
    IN Howmany INTEGER,
    IN Initial_result_row VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL DETERMINISTIC CONTAINS SQL
SQL SECURITY DEFINER
BEGIN
END

CREATE PROCEDURE TopNplus_fulfill (
    IN Input_cursor_row VARCHAR(2),
    IN Input_pby_descr VARCHAR(2)
    IN Input_order_descr VARCHAR(2),
    IN Input_cursor_name VARCHAR(2),
    IN Howmany INTEGER,
    IN Intermediate_result_row VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL NOT DETERMINISTIC READS SQL DATA
SQL SECURITY DEFINER
BEGIN
END
```

The tasks of TopNplus_describe are:

1) Validate the input (Input_order_descr must have a single column name, identifying a numeric column in Input_row_descr).

2) Populate Input_request_descr (with the name of the ordering column).

3) Populate Initial_row_descr (with a single column, whose description can be copied from the order column in Input_row_descr).

The DBMS constructs the cursor row descriptor from the requested row descriptor by appending the pass-through input surrogate column.

The DBMS constructs the intermediate result row descriptor from the initial result row descriptor by appending the pass-through output surrogate column.

Note that the cursor row descriptor and the intermediate result row descriptor are actually identical in contents, consisting of the sort column and the surrogate column.

The logic of TopNplus_fulfill does the following:

1) Fetch the first Howmany rows from the cursor into the cursor row descriptor. For each of these rows, copy the cursor row descriptor to the intermediate result row descriptor, and pipe the intermediate result row descriptor to the DBMS.

2) If there are any remaining rows, fetch them and sum the sort column in a variable.

3) After reading all rows, assign the sum to the first column of the intermediate result row descriptor, and a null to the second column.

The DBMS receives the intermediate result row descriptors that TopNplus_fulfill pipes out and expands them to obtain the complete result row, as follows:

1) The first column of the intermediate result row becomes the only proper result column in the complete result.

2) The partitioning columns are copied from invariant values on the virtual processor receiving the result row. The partitioning columns are available on the last row, even if the output surrogate is null.

3) The pass-through output surrogate column is expanded to obtain the non-partitioning columns of the input table. On the last row, when the output surrogate is null, this expands into null values for all the non-partitioning columns of the input table.

12.6 ExecR

This example was introduced in Subclause 3.2.5, “ExecR”.

12.6.1 Overview

R is a programming language used for analytic calculations. ExecR executes an R script on an input table. The PTF receives the R script as an input character string, but lacks the sophistication to analyze this R script to determine the row type of the result. Consquently, the query author bears the burden of specifying the output row type.

12.6.2 Functional specification of ExecR

ExecR executes an R script on an input table, resulting in an output table. Since the R script operates on an entire input table, the input table has set semantics. The R script might produce a result even for an empty input; therefore, the invocation cannot be pruned when the input table is empty. The PTF does not know the result's row type, so the query author must provide it in the invocation. Rows of the result have no known association with input rows; therefore, the input table does not use pass-through columns. Thus, the parameters are:

```
CREATE FUNCTION ExecR (
    Script VARCHAR(10000),
    Input TABLE NO PASS THROUGH
            WITH SET SEMANTICS KEEP WHEN EMPTY,
    Rowtype DESCRIPTOR )
RETURNS TABLE
NOT DETERMINISTIC
READS SQL DATA
```

Note that ExecR is non-deterministic because the script might be non-deterministic.

12.6.3 Design specification for ExecR

The design specification specifies details that are private, that is, not visible to the query author. For the design specification, the PTF author decides:

1) Whether PTF start and/or finish component procedures are required.

   The start component procedure will be used to attach to an R engine, and the finish component procedure will release the R engine.

2) The names of the PTF component procedures.

   The PTF author decides to name the PTF component procedures ExecR_describe, ExecR_start, ExecR_fulfill, and ExecR_finish.

3) The private data for the PTF component procedures.

   ExecR will use a private variable for the handle for the R engine.

Thus we have the following skeleton DDL:

```
CREATE FUNCTION ExecR (
    Script VARCHAR(10000),
    Input TABLE NO PASS THROUGH
            WITH SET SEMANTICS KEEP WHEN EMPTY,
    Rowtype DESCRIPTOR )
RETURNS TABLE
NOT DETERMINISTIC
READS SQL DATA
PRIVATE DATA (
    Handle INTEGER)
DESCRIBE WITH PROCEDURE ExecR_describe
START WITH PROCEDURE ExecR_start
FULFILL WITH PROCEDURE ExecR_fulfill
FINISH WITH PROCEDURE ExecR_finish
```

12.6.4 ExecR component procedures

The DBMS tool should generate something like the following:

```
CREATE PROCEDURE ExecR_describe (
    INOUT Handle INTEGER,
    IN Script VARCHAR(10000),
    IN Input_row_descr VARCHAR(2),
    IN Input_pby_descr VARCHAR(2),
    IN Input_order_descr VARCHAR(2),
    IN Input_request_descr VARCHAR(2),
    IN Rowtype_descr VARCHAR(2),
    IN Initial_result_row VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL DETERMINISTIC CONTAINS SQL
SQL SECURITY DEFINER
BEGIN
END

CREATE PROCEDURE ExecR_start (
    INOUT Handle INTEGER,
    IN Script VARCHAR(10000),
    IN Input_pby_descr VARCHAR(2),
    IN Input_order_descr VARCHAR(2),
    IN Rowtype_descr VARCHAR(2),
    IN Intermediate_result_row VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL NOT DETERMINISTIC CONTAINS SQL
SQL SECURITY DEFINER
BEGIN
END

CREATE PROCEDURE ExecR_fulfill (
    INOUT Handle INTEGER,
    IN Script VARCHAR(10000),
    IN Input_cursor_row VARCHAR(2),
    IN Input_pby_descr VARCHAR(2),
```



```
    IN Input_order_descr VARCHAR(2),
    IN Input_cursor_name VARCHAR(2),
    IN Rowtype_descr VARCHAR(2),
    IN Intermediate_result_row VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL NOT DETERMINISTIC READS SQL DATA
SQL SECURITY DEFINER
BEGIN
END

CREATE PROCEDURE ExecR_finish (
    IN Handle INTEGER,
    IN Script VARCHAR(10000),
    IN Input_pby_descr VARCHAR(2),
    IN Input_order_descr VARCHAR(2),
    IN Rowtype_descr VARCHAR(2),
    IN Intermediate_result_row VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL NOT DETERMINISTIC CONTAINS SQL
SQL SECURITY DEFINER
BEGIN
END
```

12.6.5 Invoking ExecR

Here is the DDL visible to the query author:

```
CREATE FUNCTION ExecR (
    Script VARCHAR(10000),
    Input TABLE WITH SET SEMANTICS KEEP WHEN EMPTY,
    Rowtype DESCRIPTOR )
RETURNS TABLE
```

Here is an example invocation:

```
SELECT D.Region, R.Name, R.Value
FROM TABLE ( ExecR ( Script => '...',
                     Input => TABLE (My.Data) AS D
                                PARTITION BY Region,
                     Rowtype =>
                       DESCRIPTOR (Name VARCHAR(100), Value REAL)
                   )
           ) AS R
```

12.6.6 Calling ExecR_describe

The signature for ExecR_describe is:

```
PROCEDURE ExecR_describe (
    INOUT Handle INTEGER,
    IN Script VARCHAR(10000),
    IN Input_row_descr VARCHAR(2),
```

```
    IN Input_pby_descr VARCHAR(2),
    IN Input_order_descr VARCHAR(2),
    IN Input_request_descr VARCHAR(2),
    IN Rowtype_descr VARCHAR(2),
    IN Initial_result_row VARCHAR(2),
    INOUT Status CHAR(5)
)
```

The first parameter is for the private variable, an integer called Handle. The DBMS must allocate memory for this private variable and initialize it to null.

The second parameter is the R script. This is a scalar parameter, so the DBMS can simply copy it from the query invocation when assembling the invocation of ExecR_describe.

The next six parameters pass the PTF extended names of six PTF descriptor areas. The first is the PTF descriptor area of the input table's row type; let this be called 'I1'. This will describe the table My.Data. The precise columns of My.Data will not matter going forward, so the contents of this PTF descriptor area is not shown.

The second PTF descriptor area describes the partitioning of the input table; let this be called 'P1'. This descriptor area will have a single item descriptor area, like this:

	Content
Header	COUNT = 1 TOP_LEVEL_COUNT = 2 Other components unspecified
Item 1	NAME = 'REGION' Other components unspecified

The third PTF descriptor area describes the ordering of the input table; let this be called 'S1'. Since there is no ordering, this PTF descriptor area has only a header with no item descriptor areas.

The fourth PTF descriptor area is the requested row type descriptor area; let this be called 'A1'. This is initially an empty descriptor. ExecR_describe must populate this descriptor to tell the DBMS which columns ExecR wants to receive.

The fifth PTF descriptor area is provided by the query author as:

```
DESCRIPTOR (Name VARCHAR(100),
            Val DOUBLE PRECISION)
```

Let this PTF descriptor area be called 'Q1'; its contents are:

	Content
Header	COUNT = 2 TOP_LEVEL_COUNT = 2 Other components unspecified

	Content
Item 1	NAME = 'NAME' LEVEL = 0 TYPE = 12 (for VARCHAR) LENGTH = 100 CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the catalog, schema, and local name of the default character set. Other components unspecified
Item 2	NAME = 'VAL' LEVEL = 0 TYPE = 8 (for DOUBLE) Other components unspecified

The sixth PTF descriptor area is for the intermediate result row type. Let the result row type PTF descriptor area be named 'IR'. 'IR' has the following contents:

	Content
Header	COUNT = 0 TOP_LEVEL_COUNT = 0 Other components unspecified

Finally the DBMS must allocate a CHAR(5) variable for the status code, initialized to '00000'. Let ST be the status code variable.

After assembling all the preceding, the DBMS makes the following invocation:

```
CALL ExecR_describe (
    Handle => Handle,
    Script => '...'
    Input_row_descr => 'I1',
    Input_pby_descr => 'P1',
    Input_order_descr => 'S1',
    Input_request_descr => 'A1',
    Rowtype_descr => 'Q1',
    Initial_result_row => 'IR',
    Status => ST
)
```

12.6.7 Inside ExecR_describe

ExecR_describe has the following tasks:

1) Validate the input. In this example, since the R engine is a black box to ExecR, there are probably no checks that ExecR_describe can make.

2) Populate the input request descriptor area, whose name is passed in Input_request_descr. This must be a subset of the full row type, whose name is passed in Input_row_descr. Since ExecR does not know what columns the R script will require, ExecR_describe can simply copy the full input row type to the requested row type, like this:

```
COPY DESCRIPTOR PTF Input_row_descr
            TO PTF Input_request_descr;
```

3) Populate the initial result row type descriptor, whose name is passed in Initial_result_row. Since ExecR is not designed to deduce the output row type, it is up to the query author to supply it in the argument Rowtype. This argument is a PTF descriptor area initialized by the query author using the DESCRIPTOR built-in function in SQL. The DBMS has given this PTF descriptor area the PTF extended name Q1, which is what is actually passed in to ExecR_describe. In this example, ExecR_describe only needs to copy the row type descriptor provided by the query author to the Initial_result_row., like this:

```
COPY DESCRIPTOR PTF Rowtype_descr
            TO PTF Initial_result_row;
```

12.6.8 Result of ExecR_describe

The query author wrote the following query:

```
SELECT D.Region, R.Name, R.Value
FROM TABLE ( ExecR (
    Script => '...',
    Input => TABLE (My.Data) AS D
             PARTITION BY Region,
    Rowtype => DESCRIPTOR (Name VARCHAR(100), Value REAL) )
            ) AS R
```

The row type resulting from the invocation of ExecR has one partitioning column, plus the two columns generated by ExecR itself, like this:

Correlation name	D	R	
Column name	REGION	NAME	VAL
Data type	VARCHAR(100)	VARCHAR(100)	DOUBLE PRECISION

ExecR has one private variable, called Handle. The DBMS initialized it to null before invoking ExecR, and ExecR_describe left the value null. In this example, the private variable is not used to communicate from the PTF describe component procedure to the run-time component procedures; however, in general, that is a possibility. The DBMS must save this private variable, as well as all the PTF descriptor areas for run time, when they will be replicated on every virtual processor as input to the run-time PTF component procedures.

12.6.9 Virtual processors for ExecR

The invocation (first presented in Subclause 12.6.5, “Invoking ExecR”) is:


```
SELECT D.Region, R.Name, R.Value
FROM TABLE ( ExecR (
                Script => '...',
                Input => TABLE (My.Data) AS D
                           PARTITION BY Region,
                Rowtype =>
                    DESCRIPTOR (Name VARCHAR(100), Value REAL) )
           ) AS R
```

This example has one input table, having set semantics, partitioned but not ordered. The DBMS creates one virtual processor for each partition of the input table.

The private data for ExecR is:

```
PRIVATE DATA (
    Handle INTEGER )
```

Prior to starting virtual processors, the DBMS can compute the contents of the descriptor areas that they will use, as follows:

1) There are no pass-through columns, so the cursor row type descriptor is the same as the requested row type descriptor that ExecR_describe populated. We will use 'R1' as the name of this descriptor area on all virtual processors.

2) The partitioning and ordering descriptors are the same as they were for ExecR_describe; we will call them 'P1' and 'S1' respectively.

3) There are no pass-through columns, so the intermediate result row type descriptor is the same as the initial result row type descriptor. We will use 'MR' as the name of this descriptor area on all virtual processors.

On each virtual processor, the DBMS does the following initialization:

1) The DBMS opens a PTF dynamic cursor that reads the partition assigned to that virtual processor; suppose that the PTF extended name of the cursor is 'C1' (the same PTF extended name can be used on all virtual processors because each is its own address space).

2) The DBMS creates copies of the PTF descriptor areas mentioned above.

3) The DBMS allocates memory for the SQL status code, a CHAR(5) variable initialized to '00000'. We portray this status code as a variable named ST.

12.6.10 Calling ExecR_start

On each virtual processor, the DBMS begins by Calling ExecR_start, as follows:

```
CALL ExecR_start (
    Handle => Handle,
    Script => '...'
    Input_pby_descr => 'P1',
    Input_order_descr => 'S1',
    Rowtype_descr => 'Q1',
    Intermediate_result_row => 'MR',
    Status => ST
)
```

12.6.11 Inside ExecR_start

The purpose of ExecR_start is to allocate a resource that will be used for processing the R script. ExecR_start returns the handle for this resource in the Handle argument, thereby placing it in the private variable of ExecR.

12.6.12 Calling ExecR_fulfill

On each virtual processor, after ExecR_start, the DBMS calls ExecR_fulfill. There is one input table, which is partitioned. The DBMS effectively opens a PTF dynamic cursor to read the partition; let 'C1' be the PTF extended name of this cursor. Then the DBMS calls ExecR_fulfill as follows:

```
CALL ExecR_fulfill (
    Handle => Handle,
    Script => '...'
    Input_cursor_row => 'R1',
    Input_pby_descr => 'P1',
    Input_order_descr => 'S1',
    Input_cursor_name => 'C1',
    Rowtype_descr => 'Q1',
    Intermediate_result_row => 'MR',
    Status => ST
)
```

12.6.13 Inside ExecR_fulfill

The purpose of ExecR_fulfill is to pass the input table and the script to the R processor, which returns a result table to ExecR_fulfill. ExecR_fulfill must then pass the result table to the DBMS.

ExecR_fulfill obtains the input table by performing:

```
FETCH FROM PTF Input_cursor
      INTO DESCRIPTOR PTF Input_row_descr;
```

until there are no more rows. ExecR_fulfill passes this input table to the R engine, using the interface provided by the R engine. The R engine computes the result, which is a table with row type (Name VARCHAR(100), Val DOUBLE PRECISION). The R engine passes this result back to ExecR_fulfill using its interface. For each row returned from the R engine, ExecR_fulfill populates the DATA components of the result row descriptor area named by Intermediate_result_row argument ('MR') using commands such as:

```
SET DESCRIPTOR PTF Intermediate_result_row VALUE I
    DATA = Variable;
```

Here, I is the column number being populated, and Variable holds the value of the column in the row returned by the R engine.

After populating every column of the result row, ExecR_fulfill sends the row to the DBMS with this command:

```
PIPE ROW PTF Intermediate_result_row;
```


12.6.14 Collecting the output

On every virtual processor, the DBMS collects the rows that are sent to it by PIPE ROW commands in ExecR_fulfill. These rows are prefixed with the value of the partitioning column. The union of all of these rows is the overall result of the invocation of ExecR.

12.6.15 Calling ExecR_finish

On each virtual processor, after ExecR_fulfill completes, the DBMS closes the input cursor and then calls ExecR_finish like this:

```
CALL ExecR_finish (
    Handle => Handle,
    Script => '...'
    Input_pby_descr => 'P1',
    Input_order_descr => 'S1',
    Rowtype_descr => 'Q1',
    Intermediate_result_row => 'MR',
    Status => ST
)
```

12.6.16 Inside ExecR_finish

The purpose of the PTF finish component procedure is to clean up on a virtual processor. In this example, ExecR_finish closes the connection to the R engine whose handle is in the Handle argument.

12.6.17 Cleanup

After ExecR_finish finishes on a virtual processor, the DBMS does any clean up on that virtual processor, such as deallocating the PTF descriptor areas.

12.7 Similarity

This example began in Subclause 3.2.6, “Similarity”.

12.7.1 Overview

Similarity performs an analysis on two data sets, which are both tables of two columns, treated as x and y axes of a graph. The analysis results in a number that indicates the degree of similarity between the two graphs, with 1 being perfectly identical and 0 being completely dissimilar. The numeric result is returned in a table with one row and one column. The result column is called Val and is of type REAL.

12.7.2 Functional specification of Similarity

There are two input tables, both with set semantics. By definition an empty table is totally similar to another empty table, and totally dissimilar to a non-empty table. Since there is a result if an input table is empty, neither input table can be pruned when empty. The result is not associated with any particular row of either input table, so neither input table has pass-through columns.

```
CREATE FUNCTION Similarlity (
    Input1 TABLE NO PASS THROUGH
              WITH SET SEMANTICS KEEP WHEN EMPTY,
    Input2 TABLE NO PASS THROUGH
              WITH SET SEMANTICS KEEP WHEN EMPTY )
RETURNS TABLE (Val REAL)
NOT DETERMINISTIC
READS SQL DATA
```

Similarity is not deterministic because the result might depend on the order of the input rows.

12.7.3 Design specification for Similarity

The design specification specifies details that are private, that is, not visible to the query author. For the design specification, the PTF author decides:

1) Whether PTF start and/or finish component procedures are required.

   Similarity does not use any resources outside the DBMS, so no start or finish component procedures are needed.

2) The names of the PTF component procedures.

   The PTF author decides to name the PTF component procedures Similarity_describe and Similarity_fulfill.

3) The private data for the PTF component procedures.

   Similarity has no information to pass from the describe component procedure to the fulfill component procedure, so there is no private data.

These considerations give this skeleton DDL:

```
CREATE FUNCTION Similarlity (
    Input1 TABLE NO PASS THROUGH
              WITH SET SEMANTICS KEEP WHEN EMPTY,
    Input2 TABLE NO PASS THROUGH
              WITH SET SEMANTICS KEEP WHEN EMPTY )
RETURNS TABLE (Val REAL)
NOT DETERMINISTIC
READS SQL DATA
DESCRIBE WITH PROCEDURE Similarity_describe
FULFILL WITH PROCEDURE Similarity_fulfill
```

12.7.4 Similarity component procedures

The DBMS tool should generate something like the following:

```
CREATE PROCEDURE Similarity_describe (
    IN Input1_row_descr VARCHAR(2),
    IN Input1_pby_descr VARCHAR(2),
    IN Input1_order_descr VARCHAR(2),
    IN Input1_request_row VARCHAR(2),
    IN Input2_row_descr VARCHAR(2),
    IN Input2_pby_descr VARCHAR(2),
    IN Input2_order_descr VARCHAR(2),
    IN Input2_request_row VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL DETERMINISTIC CONTAINS SQL
SQL SECURITY DEFINER
BEGIN
END
```

Note that Similarity_describe does not have a parameter for the initial result row type, since this is specified in the CREATE FUNCTION statement as (Val REAL).

```
CREATE PROCEDURE Similarity_fulfill (
    IN Input1_cursor_row VARCHAR(2),
    IN Input1_pby_descr VARCHAR(2),
    IN Input1_order_descr VARCHAR(2),
    IN Input1_cursor_name VARCHAR(2),
    IN Input2_cursor_row VARCHAR(2),
    IN Input2_pby_descr VARCHAR(2),
    IN Input2_order_descr VARCHAR(2),
    IN Input2_cursor_name VARCHAR(2),
    IN Intermediate_result_row VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL NOT DETERMINISTIC READS SQL DATA
SQL SECURITY DEFINER
BEGIN
END
```

12.7.5 Invoking Similarity

Here is the DDL visible to the query author:

```
FUNCTION Similarlity (
    Input1 TABLE NO PASS THROUGH
             WITH SET SEMANTICS KEEP WHEN EMPTY,
    Input2 TABLE NO PASS THROUGH
             WITH SET SEMANTICS KEEP WHEN EMPTY )
RETURNS TABLE (Val REAL)
```

Here is an example invocation:

```
SELECT T1.Country, T2.Code, S.val
FROM TABLE ( Similarity ( Input1 => TABLE (Sales) AS T1
                                    PARTITION BY Country
                                    ORDER BY (Qtr, Revenue),
                          Input2 => TABLE (Countries) AS T2
                                    PARTITION BY Code
                                    ORDER BY (Quarter, GDP)
                          COPARTITION (T1, T2) )
           ) AS S
```

12.7.6 Calling Similarity_describe

To compile this, the DBMS will call Similarity_describe, whose signature is given in Subclause 12.7.4, “Similarity component procedures”, as:

```
PROCEDURE Similarity_describe (
    IN Input1_row_descr VARCHAR(2),
    IN Input1_pby_descr VARCHAR(2),
    IN Input1_order_descr VARCHAR(2),
    IN Input1_request_row VARCHAR(2),
    IN Input2_row_descr VARCHAR(2),
    IN Input2_pby_descr VARCHAR(2),
    IN Input2_order_descr VARCHAR(2),
    IN Input2_request_row VARCHAR(2),
    INOUT Status CHAR(5)
)
```

The signature for Similarity_describe requires eight PTF descriptor areas. The first is the PTF descriptor area of the full row type of Input1; let this be called 'I1'. This will describe the table S2. The PTF descriptor area might look like this:

	Content
Header	COUNT = 3 TOP_LEVEL_COUNT = 2 Other components unspecified

	Content
Item 1	NAME = 'COUNTRY' LEVEL = 0 TYPE = 1 (for CHAR) LENGTH = 3 CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the catalog, schema, and local name of the default character set. Other components unspecified
Item 2	NAME = 'QTR' LEVEL = 0 TYPE = 1 (for CHAR) LENGTH = 6 CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are the catalog, schema, and local name of the default character set. Other components unspecified
Item 3	NAME = 'REVENUE' LEVEL = 0 TYPE = 7 (for REAL) Other components unspecified

The second PTF descriptor area describes the partitioning of Input1; let this be called 'P1'. This PTF descriptor area looks like this:

	Content
Header	COUNT = 1 TOP_LEVEL_COUNT = 1 Other components unspecified
Item 1	NAME = 'COUNTRY' LEVEL = 0 Other components unspecified

The third PTF descriptor area describes the ordering of Input1; let this be called 'S1'. This PTF descriptor area looks like this:

	Content
Header	COUNT = 2 TOP_LEVEL_COUNT = 2 Other components unspecified

	Content
Item 1	NAME = 'QTR' LEVEL = 0 ORDER_DIRECTION = 1 (for ASC) NULLS_SORT_DIRECTION = 1 or –1, depending on DBMS default Other components unspecified
Item 2	NAME = 'REVENUE' LEVEL = 0 ORDER_DIRECTION = 1 (for ASC) NULLS_SORT_DIRECTION = 1 or –1, depending on DBMS default Other components unspecified

The fourth is an empty PTF descriptor area for the requested row type for Input1; let this be called 'A1'.

Next there are four PTF descriptor areas for the full row type, partitioning, ordering, and requested row type of Input2. Let these PTF descriptor areas be called 'I2', 'P2', 'S2', and 'A2', respectively. Their contents are very similar to the preceding PTF descriptor areas.

Finally, the DBMS must allocate a CHAR(5) for the SQL status code; let this be called ST. It is initialized to '00000' (success).

After allocating and populating the PTF descriptor areas and status code, the DBMS makes the following invocation:

```
CALL Similarity_describe (
    Input1_row_descr => 'I1',
    Input1_pby_descr => 'P1',
    Input1_order_descr => 'S1',
    Input1_request_row => 'A1',
    Input2_row_descr => 'I2',
    Input2_pby_descr => 'P2',
    Input2_order_descr => 'S2',
    Input2_request_row => 'A2',
    Status => ST
)
```

12.7.7 Inside Similarity_describe

Similarity_describe has three tasks:

1) Validate the input. The PTF author has specified to the query author that each input table must have two numeric columns and that each input table is ordered on those two columns. Similarity_describe can check that these conditions have been met by inspecting the relevant PTF descriptor areas using GET DESCRIPTOR. If the condition is not met, then Similarity_describe should exit with an error status.

2) Populate the requested row descriptor of Input1. Similarity only needs the two sort columns, so it can simply copy their names from Input1_order_descr to Input1_request_row like this:


```
    COPY DESCRIPTOR PTF Input1_order_descr (NAME)
                TO PTF Input1_request_row;
```

3) Populate the requested row descriptor of Input2, similarly to Input1.

12.7.8 Result of Similarity_describe

There are two partitioning columns, in addition to the one output column from Simillarity. There are no pass-through columns. Thus, the complete result row type is:

Correlation name	T1	T2	C
Column name	COUNTRY	CODE	VAL
Column type	VARCHAR(3)	VARCHAR(3)	REAL

12.7.9 Virtual processors for Similarity

The invocation (first presented in Subclause 12.7.5, “Invoking Similarity”, is:

```
SELECT T1.Country, T2.Code, S.val
FROM TABLE ( Similarity ( Input1 => TABLE (Sales) AS T1
                                    PARTITION BY Country
                                    ORDER BY (Qtr, Revenue),
                          Input2 => TABLE (Countries) AS T2
                                    PARTITION BY Code
                                    ORDER BY (Quarter, GDP)
                          COPARTITION (T1, T2) )
           ) AS S
```

This example has two copartitioned tables; both specify KEEP WHEN EMPTY. To process the copartitioning, the DBMS effectively creates a master list of every distinct value of country codes in either of the two input tables. This can be done with the following “master list” query:

```
SELECT *
FROM ( SELECT DISTINCT Country, 1 AS One
       FROM Sales ) AS S3
  FULL OUTER JOIN
     ( SELECT DISTINCT Code, 1 AS One
       FROM Countries ) AS T3
  ON ( S3.Country IS NOT DISTINCT FROM T3.Code )
```

(The IS NOT DISTINCT FROM predicate is *True* if the two comparands are equal or both null.)

For example, suppose that the distinct values of Sales.Country are 'CAN', 'JPN', and 'USA', whereas the distinct values of Countries.Code are 'CAN', 'JPN', and 'GBR'. The result of the preceding query is:

S3.Country	S3.One	T3.Code	T3.One
CAN	1	CAN	1
JPN	1	JPN	1
USA	1		
		GBR	1

Thus there are four copartitions (for 'CAN', 'JPN', 'USA', and 'GBR') and the DBMS must start a virtual processor for each of them.

In this example, both input tables are KEEP WHEN EMPTY. This leaves the option for the query to specify PRUNE WHEN EMPTY on either input table. For example, suppose the query is:

```
SELECT T1.Country, T2.Code, S.val
FROM TABLE ( Similarity ( Input1 => TABLE (Sales) AS T1
                                    PARTITION BY Country
                                    PRUNE WHEN EMPTY
                                    ORDER BY (Qtr, Revenue),
                          Input2 => TABLE (Countries) AS T2
                                    PARTITION BY Code
                                    KEEP WHEN EMPTY
                                    ORDER BY (Quarter, GDP)
                          COPARTITION (T1, T2) )
           ) AS S
```

Here the query author has requested pruning of empty partitions in the first input table but not in the second. This results in pruning the virtual processor for 'GBR', since this value is only found in the second table, not the first. This can be determined from the query above using:

```
SELECT *
FROM ( SELECT DISTINCT Country, 1 AS One
       FROM Sales ) AS S3
  FULL OUTER JOIN
     ( SELECT DISTINCT Code, 1 AS One
       FROM Countries ) AS T3
  ON ( S3.Country IS NOT DISTINCT FROM T3.Code )
WHERE S3.One IS NOT NULL
```

This is also equivalent to a right outer join:

```
SELECT *
FROM ( SELECT DISTINCT Country, 1 AS One
       FROM Sales ) AS S3
  RIGHT OUTER JOIN
     ( SELECT DISTINCT Code, 1 AS One
       FROM Countries ) AS T3
  ON ( S3.Country IS NOT DISTINCT FROM T3.Code )
```

Returning to the original query: the DBMS creates one virtual processor for each distinct value in the master list. On a virtual processor, the cursor for S2 fetches those rows of S2 that match the virtual processor's value of country (this might be empty). Similarly, the cursor for T2 fetches those rows of T2 that match the virtual

processor's value of country. Each cursor is ordered as shown in the invocation ORDER BY clauses. For example, on the copartition for 'CAN', the cursors might be:

```
SELECT Qtr, Revenue
FROM Sales
WHERE Country IS NOT DISTINCT FROM 'CAN'
ORDER BY Qtr, Revenue

SELECT Quarter, GDP
FROM Countries
WHERE Code IS NOT DISTINCT FROM 'CAN'
ORDER BY Quarter, GDP
```

(The cursors only fetch the sort columns because those are the columns requested by Similarity_describe.)

If an input table does not have any rows for a value in the master list, then an empty cursor should be used. For example, on the copartition for 'GBR', the cursors can be:

```
SELECT Qtr, Revenue
FROM Sales
WHERE Country IS NOT DISTINCT FROM 'GBR'
ORDER BY Qtr, Revenue

SELECT Quarter, GDP
FROM Countries
WHERE Code IS NOT DISTINCT FROM 'GBR'
ORDER BY Quarter, GDP
```

The first cursor above finds no rows, since there are no matches for 'GBR' in the first input table.

The sample data does not illustrate the tricky case of a null in the master list. It can happen that one input table has a null value in a partitioning column, and the other input table is never null in the corresponding partitioning column. The known not null column One can be used to distinguish nulls generated by null-extending a row *vs* nulls that are present in the input table. For example, suppose the master list had this result:

S3.Country	S3.One	T3.Code	T3.One
CAN	1	CAN	1
JPN	1	JPN	1
USA	1		
		GBR	1
			1

In the last row, the null in T3.Code reflects a null in the data because T3.One is not null, whereas the null in S3.Country is due to the outer join extension because S3.One is null. The cursors for this row might be:

```
SELECT Qtr, Revenue
FROM Sales
WHERE Country IS NOT DISTINCT FROM CAST (NULL AS VARCHAR(3))
ORDER BY Qtr, Revenue
```

```
SELECT Quarter, GDP
FROM Countries
WHERE Code IS NOT DISTINCT FROM CAST (NULL AS VARCHAR(3))
ORDER BY Quarter, GDP
```

The cursor for Sales will be empty because Sales does not in fact have any rows in which Country is null. The cursor for Countries is non-empty because there are rows with a null Code.

It is also possible to invoke Similarity without copartitioning, like this:

```
SELECT T1.Country, T2.Code, S.val
FROM TABLE ( Similarity ( Input1 => TABLE (Sales) AS T1
                                    PARTITION BY Country
                                    ORDER BY (Qtr, Revenue),
                          Input2 => TABLE (Countries) AS T2
                                    PARTITION BY Code
                                    ORDER BY (Quarter, GDP) )
           ) AS S
```

When the copartitioning clause is omitted, the DBMS forms the cross product of partitions. In that case, the master list is constructed using this query:

```
SELECT *
FROM ( SELECT DISTINCT Country FROM Sales) AS S3,
     ( SELECT DISTINCT Code FROM Countries ) AS T3
```

Note that this query does not add a known not null column, the reason being that there is no outer join, so any nulls in the result must arise from nulls in the data. The preceding query has the following results:

S3.Country	T3.Code
CAN	CAN
JPN	JPN
CAN	GBR
JPN	CAN
JPN	JPN
JPN	GBR
USA	CAN
USA	JPN
USA	GBR

In either the copartitioned or the non-copartitioned case, before starting the virtual processors, the DBMS can compute the descriptor areas that every virtual processor will need. They are:

1) The cursor row type descriptors for each input table; since there are no pass-through columns, these are identical to the requested row type descriptors that were populated by Similarity_describe. Let them be called 'I1' and 'I2'.

2) The partitioning and ordering descriptors for each input table, the same as they were for Similarity_describe. Let them be called 'P1', 'P2', 'S1', and 'S2'.

3) The intermediate result row type descriptor. Since there are no pass-through columns, this describes the initial result row, which was declared in DDL as (Val REAL). Let this descriptor area be named 'R'.

On each virtual processor, the DBMS does the following initialization:

1) The DBMS opens a PTF dynamic cursor that reads the partition assigned to that virtual processor; suppose that the PTF extended names of the cursors are C1 and C2 (the same PTF extended names can be used on all virtual processors because each is its own address space).

2) The DBMS creates copies of the PTF descriptor areas mentioned above.

3) The DBMS allocates memory for the SQL status code, a CHAR(5) variable initialized to '00000'. We portray this status code as a variable named ST.

12.7.10 Calling Similarity_fulfill

On each virtual processor, the DBMS calls Similarity_fulfill as follows:

```
CALL Similarity_fulfill (
    Input1_cursor_row => 'I1',
    Input1_pby_descr => 'P1',
    Input1_order_descr => 'S1',
    Input1_cursor_name => 'C1',
    Input2_cursor_row => 'I2',
    Input2_pby_descr => 'P2',
    Input2_order_descr => 'S2',
    Input2_cursor_name => 'C2',
    Intermediate_result_row => 'R',
    Status => ST
)
```

12.7.11 Inside Similarity_fulfill

Similarity_fulfill might have the following logic:

1) Read all rows of the first input table using:

```
FETCH FROM PTF Input1_cursor_name
      INTO DESCRIPTOR PTF Input1_cursor_row;
```

2) Read all rows of the second input table using:

```
FETCH FROM PTF Input2_cursor_name
      INTO DESCRIPTOR PTF Input2_cursor_row;
```

3) Compute the similarity of the two tables in a variable called Sim.

4) Place the result in the result row descriptor:

```
SET DESCRIPTOR PTF Intermediate_result_row VALUE 1
    DATA = Sim;
```

5) Pipe this row to the DBMS:

```
PIPE ROW PTF Intermediate_result_row;
```

12.7.12 Collecting the output

The DBMS collects the output supplied by PIPE ROW commands on each virtual processor. The complete row also includes the partitioning columns. For example, the complete output may look like this:

S2.Country	T2.Code	C.Val
CAN	CAN	0.5
JPN	JPN	0.7
USA		0.0
	GBR	0.0

12.7.13 Cleanup

When each virtual processor is finished, the DBMS closes the input cursors, deallocates all of its data structures, and closes the virtual processor.

12.8 UDjoin

This example began in Subclause 3.2.7, “UDjoin”.

12.8.1 Overview

UDjoin performs a user-defined join. It takes two input tables, *T1* and *T2*, and matches rows according to a join criterion. It is intended that *T2* is ordered on a timestamp. UDjoin will analyze this ordered data into “clusters” of related rows, where each cluster is interpreted as representing some “event”. If two rows are tied in the ordering, they are placed in the same cluster. Some rows may be interpreted as “noise”, not representing any event.

After analyzing *T2* into event clusters, rows from *T1* are matched to the most relevant event cluster. It is possible that some rows of *T1* have no matching event cluster. It is also possible that some event clusters have no match in *T1*.

The output resembles a full outer join. If a row *R* of *T1* matches an event cluster *EC* of *T2*, then, in the output, *R* is joined to every row of *EC*. If *R* has no matching event cluster, then *R* is output with a null-extended row in place of the event cluster. Conversely, if an event cluster *EC* is not matched, then every row of *EC* is output with nulls in the portion of the output corresponding to *T1*.

12.8.2 Functional specification of UDjoin

UDjoin has two input tables with set semantics. Because of the resemblance to a full outer join, there can be results if either input tables is empty. UDjoin uses pass-through columns on both input tables. There are no proper result columns; the only result columns are the pass-through columns. These considerations give the following skeleton DDL:

```
CREATE FUNCTION UDjoin (
    Candidates  TABLE PASS THROUGH
                  WITH SET SEMANTICS KEEP WHEN EMPTY,
    EventStream TABLE PASS THROUGH
                  WITH SET SEMANTICS KEEP WHEN EMPTY
) RETURNS ONLY PASS THROUGH
DETERMINISTIC
READS SQL DATA
```

UDjoin is deterministic because, although the Candidates input is sorted, ties are placed in the same cluster, so indeterminacy in the input ordering does not cause indeterminacy in the output.

12.8.3 Design specification for UDjoin

The design specification specifies details that are private, that is, not visible to the query author. For the design specification, the PTF author decides:

1) Whether PTF start and/or finish component procedures are required.

UDjoin does not use any resources outside the DBMS, so no start or finish component procedures are needed.

2) The names of the PTF component procedures.

   The PTF author decides to name the PTF component procedures UDjoin_describe and UDjoin_fulfill.

3) The private data for the PTF component procedures.

   UDjoin has no information to pass from the describe component procedure to the fulfill component procedure, so there is no private data.

These considerations give this skeleton DDL:

```
CREATE FUNCTION UDjoin (
    Candidates  TABLE PASS THROUGH
                  WITH SET SEMANTICS KEEP WHEN EMPTY,
    EventStream TABLE PASS THROUGH
                  WITH SET SEMANTICS KEEP WHEN EMPTY
) RETURNS ONLY PASS THROUGH
DETERMINISTIC
READS SQL DATA
DESCRIBE WITH PROCEDURE UDjoin_describe
FULFILL WITH PROCEDURE UDjoin_fulfill
```

12.8.4 UDjoin component procedures

The DBMS tool should generate something like the following:

```
CREATE PROCEDURE UDjoin_describe (
    IN Candidates_full_row VARCHAR(2),
    IN Candidates_pby_descr VARCHAR(2),
    IN Candidates_oby_descr VARCHAR(2),
    IN Candidates_requested_row VARCHAR(2),
    IN EventStream_full_row VARCHAR(2),
    IN EventStream_pby_descr VARCHAR(2),
    IN Eventstream_oby_descr VARCHAR(2),
    IN EventStream_requested_row VARCHAR(2),
    INOUT Status CHAR(5)
) LANGUAGE SQL DETERMINISTIC CONTAINS SQL
```

Note that there is no descriptor parameter for the initial result row type, because the PTF declares RETURNS ONLY PASS THROUGH.

```
CREATE PROCEDURE UDjoin_fulfill (
    IN Candidates_cursor_row VARCHAR(2),
    IN Candidates_pby_descr VARCHAR(2),
    IN Candidates_oby_descr VARCHAR(2),
    IN Candidates_cursor_name VARCHAR(2),
    IN EventStream_cursor_row VARCHAR(2),
    IN EventStream_pby_descr VARCHAR(2),
    IN Eventstream_oby_descr VARCHAR(2),
    IN EventStream_cursor_name VARCHAR(2),
    IN Intermediate_result_row VARCHAR(2)
```



```
    INOUT Status CHAR(5)
) LANGUAGE SQL DETERMINISTIC READS SQL DATA
```

12.8.5 Invoking UDjoin

Here is a sample query to invoke UDjoin:

```
SELECT G.*, S.*
FROM TABLE ( UDjoin ( Candidates =>  TABLE (Goods) AS G,
                      EventStream => TABLE (TimeSeries) AS S
                          ORDER BY Tstamp ) )
```

It is also possible to partition either or both input tables. If both are partitioned with the same number of partitioning columns and corresponding partitioning columns are comparable, then copartitioning is possible (see Subclause 12.7, “Similarity”, for an example of copartitioning). If copartitioning is not specified, then the cross product of partitions would be formed. This example ignores the possibility of partitioning and focuses on the UDjoin's distinctive use of RETURNS ONLY PASS THROUGH. This requires that the DBMS support Feature B205, “Pass-through columns”.

The sample output in Subclause 3.2.7, “UDjoin”, posited the following input tables:

```
TABLE Goods (
    Gid INTEGER,
    Golly VARCHAR(20),
    Wiz VARCHAR(20)
)

TABLE TimeSeries (
    Tstamp INTEGER,
    Color VARCHAR(20),
    Shape VARCHAR(20)
)
```

We’ll assume that Timeseries (Color, Shape) is used to determine event clusters, which are then matched to Goods using the columns (Golly, Wiz).

12.8.6 Calling UDjoin_describe

The DBMS needs to create all the descriptor areas to pass to UDjoin, and assign them PTF extended names. Note that the requested row type descriptors for the two input tables are empty prior to invoking UDjoin_describe. The call to UDjoin_describe might look like this:

```
CALL UDjoin_describe (
    Candidates_full_row => 'F1',
    Candidates_pby_descr => 'P1',
    Candidates_oby_descr => 'S1',
    Candidates_requested_row => 'A1',
    EventStream_full_row => 'F2',
    EventStream_pby_descr => 'P2',
    Eventstream_oby_descr => 'S2',
    EventStream_requested_row => 'A2',
```

```
    St => St
)
```

There are no proper result columns because the PTF specifies RETURNS ONLY PASS THROUGH, so UDjoin_describe does not need to describe the initial result row.

12.8.7 Inside UDjoin_describe

The tasks for UDjoin_describe are:

1) Validate the inputs. We assume that the inputs in this example are acceptable. If not, UDjoin_describe would set the status code to some value other than '00000' (success).

2) Populate the Candidates_requested_row descriptor area. We assume the two columns of Goods in which UDjoin is interested are (Golly, Wiz).

3) Populate the EventStream_requested row descriptor area. We assume that the two columns of EventStream in which UDjoin is interested are (Color, Shape).

12.8.8 Result of UDjoin_describe

The DBMS checks that UDjoin_describe did not return an exception, and that both requested row type descriptors have been populated with names of columns of their respective input tables.

The DBMS saves the requested row type descriptor areas for use in creating the cursor row type descriptors at run-time.

Because the PTF specifies RETURNS ONLY PASS THROUGH, the complete result row type is the concatenation of the input table row types, like this:

Correlation name	G			S		
Column name	Gid	Golly	Wiz	Tstamp	Color	Shape

12.8.9 Virtual processors for UDjoin

This example has two input tables with set semantics, neither of them partitioned. Consequently, there is one partition for each of them, and the number of virtual processors is 1 * 1 = 1.

12.8.10 Calling UDjoin_fulfill

On the single virtual processor, the DBMS must do the following:

1) Determine the cursor row type for each input table, consisting of the requested row type plus one additional column for the pass-through input surrogate. The surrogate columns are of implementation-defined type and implementation-dependent name. The names must be distinct from the names of the other columns in the cursor rows, and from one another. We will suppose that the DBMS names the columns "$surr1" for the first input table and "$surr2" for the second input table. Thus, the row type for first table argument is (Golly, Wiz, "$surr1") and the row type for the second table argument is (Color, Shape, "$surr2").

2) Create the intermediate result row type descriptor, consisting of two columns, one for each table argument's pass-through output surrogate. These columns must have the same names as the corresponding pass-through input surrogate columns in the cursor row type descriptors. Thus the row type for the intermediate row type descriptor is ("$surr1", "$surr2").

3) Open cursors for each input table.

Then the DBMS is ready to call UDjoin fulfill.

12.8.11 Inside UDjoin_fulfill

UDjoin_fulfill logic might look like this:

1) Read the EventStream input cursor and identify event clusters. For each cluster, UDjoin_fulfill needs to buffer at least the pass-through input surrogate column, and probably additional columns as necessary to match event clusters to specific rows of the first table argument.

2) Read the Candidates input cursor. For each row, identify the best matching event cluster, if any.

3) After matching Candidates rows with event clusters, UDjoin_fulfill can produce its output. There are three kinds of output:

    a) If a Candidates row matches an event cluster, then UDjoin_fulfill copies the Candidates pass-through input surrogate column ("$surr1") to the pass-through output surrogate column column, also named "$surr1", in the intermediate result row type descriptor, and, for each row in the event cluster, copies the EventStream pass-through input surrogate column ("$surr2") to the pass-through output surrogate column, also named "$surr2", in the intermediate result row type descriptor.

    b) If a Candidates row does not have a matching event cluster, then UDjoin_fulfill copies the Candidates pass-through input surrogate column ("$surr1") to the pass-through output surrogate column, also named "$surr1", in the intermediate result row type descriptor, and places a null value in the other pass-through output surrogate column (the one named "$surr2") of the intermediate result row type descriptor.

    c) If an event cluster has no match in Candidates, then UDjoin_fulfill places a null in the pass-through output surrogate column (“$surr1") for the Candidates table and, for all rows in the event cluster, copies the pass-through input surrogate column ("$surr2") to the corresponding pass-through output surrogate column, also named "$surr2".

    After populating the intermediate result row type descriptor, UDjoin_fulfill performs a PIPE ROW statement to send the row to the DBMS.

12.8.12 Collecting the output

The DBMS receives rows that are passed via PIPE ROW statements. For each row that is received, the DBMS expands the pass-through surrogate output values back into the columns of the Candidates or EventStream input tables. When a pass-through output surrogate column is null, the result of the expansion is nulls in all columns.

12.8.13 Cleanup

When UDjoin_fulfill returns control to the DBMS, the DBMS can destroy all the descriptors, cursors, and whatever other control structures it created on the virtual processor.

12.9 Nested PTF invocation

As stated in Subclause 3.2.8, "MapReduce", the Map/Reduce paradigm for data processing can be implemented using nested table function invocations. This section does not consider the specifics of Map and Reduce, instead focusing on the issues that are distinctive to any nested PTF invocations. The main issues are how to handle pass-through columns and partitioning columns.

12.9.1 Nested PTF syntax and semantics

The general pattern to be considered in this section is the following nested PTF invocation:

```
SELECT S.*, R.*, E.*
FROM TABLE ( G ( F ( TABLE (Emp) AS E ) AS R ) ) AS S
```

Each PTF has a single table argument. There are variations in this pattern, depending on whether the table argument has pass-through columns or partitioning columns. To handle these variations clearly, the following PTFs are posited:

PTF	Table argument semantics	Name of proper result column
Gp	Pass-through columns	Gpo
Gs	Set semantics (allowing partitioning) but no pass-through columns	Gso
Fp	Pass-through columns	Fpo
Fs	Set semantics (allowing partitioning) but no pass-through columns	Fso

In addition, suppose that Emp has three columns: Empno, Ename, and Edept.

The first scenario to explore is pass-through nested within pass-through. The query is:

```
SELECT S.*, R.*, E.*
FROM TABLE ( Gp ( Fp ( TABLE (Emp) AS E ) AS R ) ) AS S
```

which is equivalent to the following:

```
SELECT S.Gpo, R.Fpo, E.Empno, E.Ename, E.Edept
FROM TABLE ( Gp ( Fp ( TABLE (Emp) AS E ) AS R ) ) AS S
```

The flow of data in the scenario can be pictured:

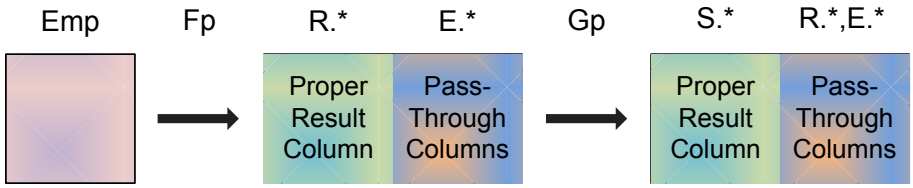

Figure 3 — Nested PTF data flow

That is,

1) Emp is processed by Fp,

2) resulting in a proper result column (R.Fpo) and pass-through columns (E.Empno, E.Ename, E.Edept),

3) which are processed by Gp,

4) resulting in a proper result column (S.Gpo) and pass-through columns (R.Fpo, E.Empno, E.Ename, E.Edept)

Thus the columns of Emp are passed through twice, and they retain their correlation name E when they emerge from the outer PTF invocation.

The next scenario is pass-through in the outer PTF invocation and partitioning in the inner PTF invocation:

```
SELECT S.*, R.*, E.*
FROM TABLE ( Gp ( Fs ( TABLE (Emp) AS E
                         PARTITION BY Edept
                     ) AS R
                )
           ) AS S
```

This scenario is equivalent to the following query:

```
SELECT S.Gpo, R.Fpo, E.Edept
FROM TABLE ( Gp ( Fs ( TABLE (Emp) AS E
                         PARTITION BY Edept
                     ) AS R
                )
           ) AS S
```

The difference from the first scenario — Gp(Fp) — is that now E.* only qualifies Edept, because Fs does not support pass-through columns, so only the partitioning column E.Edept is exported from Fs and visible to Gp. Gp has pass-through columns so E.Edept is passed through Gp to the outer query.

For the next variation, consider set semantics with partitioning in the outer PTF and pass-through columns in the inner PTF:

```
SELECT S.*, R.*, E.*
FROM TABLE ( Gs ( Fp ( TABLE (Emp) AS E ) AS R )
                         PARTITION BY E.Edept, R.Fpo
           ) AS S
```

This scenario is equivalent to:

```
SELECT S.Gso, R.Fpo, E.Edept
FROM TABLE ( Gs ( Fp ( TABLE (Emp) AS E ) AS R )
                          PARTITION BY E.Edept, R.Fpo
           ) AS S
```

In this scenario, Fp has result columns that are qualified by E and R. These columns are available for partitioning in the outer PTF. Note the use of the correlation names E and R in the PARTITION BY clause. Correlation names have not been shown in other examples of partitioning in this Technical Report, because the other examples only had a single correlation name that was applicable to the PARTITION BY clause. In the event of name conflicts between pass-through columns and proper result columns of Fp, the correlation names would be necessary to disambiguate (though this is not necessary in the example above).

Coming out of Gs, there are the proper result column S.Gso and the two partitioning columns R.Fpo and E.Edept. (By hypothesis, Gs does not have pass-through columns, so the other columns of Emp do not emerge out of Gs.)

The final scenario is set semantics with partitioning in both the inner and the outer PTF:

```
SELECT S.*, R.*, E.*
FROM TABLE ( Gs ( Fs ( TABLE (Emp) AS E
                          PARTITION BY Edept
                      ) AS R
                ) PARTITION BY E.Edept, R.Fso
           ) AS S
```

which is equivalent to:

```
SELECT S.Gso, R.Fso, E.Edept
FROM TABLE ( Gs ( Fs ( TABLE (Emp) AS E
                          PARTITION BY Edept
                      ) AS R
                ) PARTITION BY E.Edept, R.Fso
           ) AS S
```

In this example, the columns emerging from Fs are R.Fso (the proper result column) and E.Edept (the partitioning column of Fs). The columns emerging from G are S.Gso (the proper result column), R.Fso, and E.Edept (the partitioning columns of Gs).

If instead the query were:

```
SELECT S.*, R.*, E.*
FROM TABLE ( Gs ( Fs ( TABLE (Emp) AS E
                          PARTITION BY Edept ) AS R
                ) PARTITION BY R.Fso
           ) AS S
```

then the query is a syntax error, because there is no column in the final result that is qualified by E. The difference is the outer PARTITION BY clause, which now omits E.Edept. It was the presence of E.Edept in the outer PARTITION BY clause that exposed E.Edept to the main query. This is because Gs has set semantics but not pass-through columns.

To summarize, the syntax and semantics of nested PTF invocations can be understood by starting with the inner invocation and moving to the outer invocation. The complete result row type of the inner PTF becomes the input to the outer PTF. Visibility of pass-through or partitioning columns is determined by the properties of both PTFs, applied starting with the inner one and then the outer one.

12.9.2 Nested PTF compilation

This section considers how the DBMS compiles the scenarios shown in Subclause 12.9.1, "Nested PTF syntax and semantics". As just stated in summarizing that section, the query author can understand a nested PTF invocation by starting with the inner invocation and then moving to the outer invocation. This principle also governs the compilation of nested PTF invocations.

In both compilation and subsequently in execution, a number of descriptors are used. The master diagram for the flow of these descriptors is found in Subclause 4.8, "Flow of row types". The compilation phase is responsible for generating these descriptors. The DBMS can mostly manage the descriptors of each PTF separately, though the bridge from the inner PTF invocation to the outer is that the complete result row type of the inner PTF becomes the input row type of the outer PTF.

In this section we consider a single general case, with PTFs called G and F, both having a single table argument with set semantics and/or pass-through columns. Let the proper result column of G be Go, and the proper result column of F be Fo. The schematic query is:

```
SELECT S.*, R.*, E.*
FROM TABLE ( G ( F ( TABLE (Emp) AS E ) AS R ) ) AS S
```

with optionally some partitioning of the inner or the outer PTF invocation, or both. The DBMS begins by compiling the inner PTF invocation, obtaining the following flow of row types:

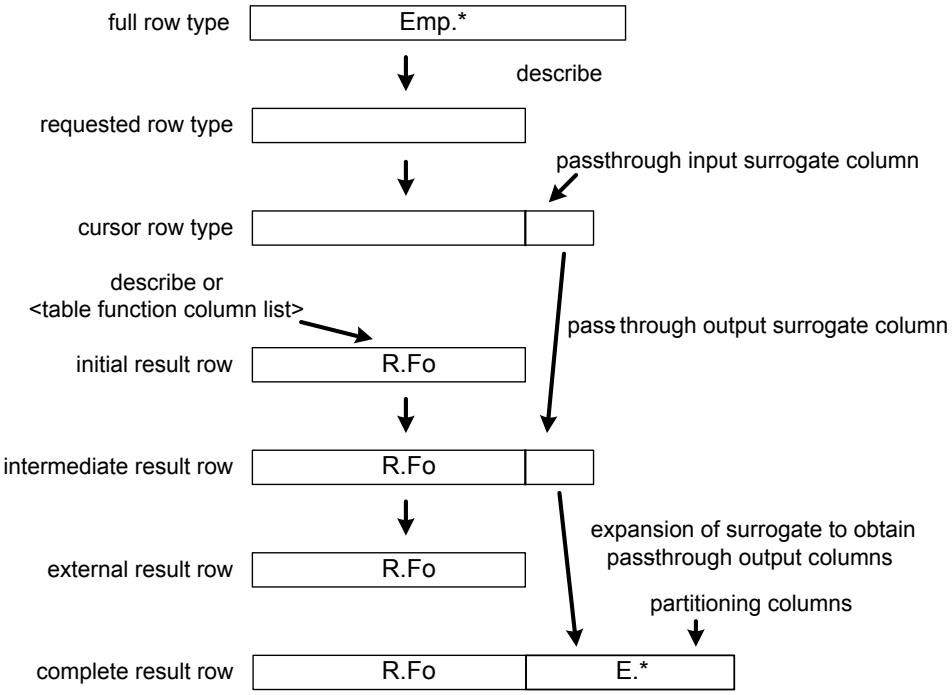

Figure 4 — Flow of row types

For the sake of syntax checking and type inferencing, the net effect can be simplified to this:

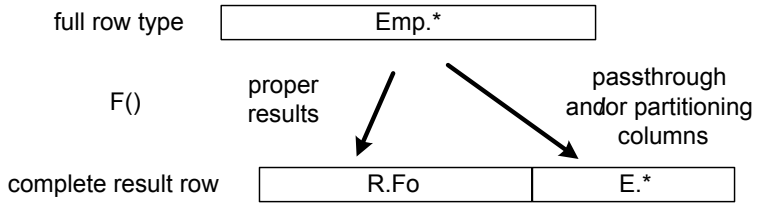

Figure 5 — Simplified flow of row types

After compiling the inner PTF invocation, the DBMS can compile the outer PTF invocation, using substantially the same diagrams. The key point is that the output of F (the complete result row) is the input to G. Thus the net effect of the complete compilation can be diagrammed:

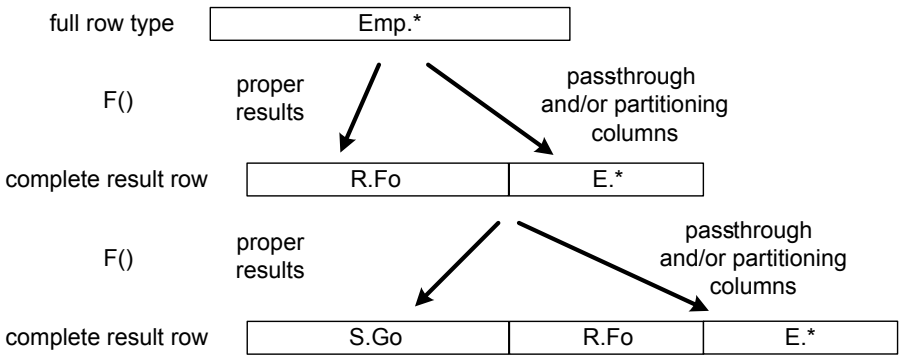

Figure 6 — Net effect of complete compilation

12.9.3 Nested PTF execution

Execution must also begin with the inner invocation and move to the outer. (With multiprocessing, the outer execution may be able to start before the inner execution completes, but the outer execution can only operate on rows already generated by the inner execution, so ultimately the outer execution must wait for the inner execution to supply rows.)

The inner execution can proceed like any other PTF invocation. As the rows are produced, they are accumulated and passed as input to the outer execution, which then produces the final result.

The distinctive thing to examine here is the pass-through columns. Suppose that both F and G have a table argument with pass-through columns. The schematic query is:

```
SELECT S.*, R.*, E.*
FROM TABLE ( G (F ( TABLE (Emp) AS E ) AS R ) ) AS S
```

The pass-through input surrogate column within F represents the columns of Emp. The pass-through input surrogate column within G represents the complete result row of F, which is the proper result columns of F plus the columns of Emp. Note that the input cursor to G has only one pass-through input surrogate column (it does not have one surrogate for R.* and another surrogate for E.*).

12.9.4 The PTF author's view of nested PTF invocations

The preceding sections have talked about the query author's and the DBMS's view of nested PTF invocation. What about the PTF author? The answer is that the PTF author does not need to be concerned with this at all. In fact, the PTF author cannot be concerned with this. The reason is that a PTF never knows anything about the table arguments, aside from the descriptors. In particular, a PTF does not know how its table arguments are formed, whether as a table name, a view name, an in-line view, or a nested PTF invocation. Consequently, there is nothing special for a PTF to do if its input is the output of a nested PTF invocation.

We have seen that an inner PTF invocation can attach more than one range variable to its output. This detail is hidden from the outer PTF invocation, because descriptors do not describe the correlation names associated with the columns being described. Managing the correlation names is handled by the DBMS, not the inner or the outer PTF invocation.

Bibliography

[Programming Language R] https://www.r-project.org/

(Blank page)

Index

Index entries appearing in **boldface** indicate the page where the word, phrase, or BNF nonterminal was defined; index entries appearing in *italics* indicate a page where the BNF nonterminal was used in a Format; and index entries appearing in roman type indicate a page where the word, phrase, or BNF nonterminal was used in a heading, Function, Syntax Rule, Access Rule, General Rule, Conformance Rule, Table, or other descriptive text.

— E —

— F —

— G —

— H —

— I —

— J —

— K —

— L —

— M —

— N —

— O —

— P —

— R —

ZONE • 55

ICS 35.060
Price based on 183 pages