DB2 UDB To PostgreSQL Conversion Guide

DB2 UDB To PostgreSQL Migration

Draft Version : 1.0

Table Of Contents

1. Introduction

Since migrating from DB2 UDB to PostgreSQL requires a certain level of knowledge in both environments, the purpose of this document is to identify the issues in the process involved migrating from DB2 UDB to PostgreSQL database.

This document also relates the required information on PostgreSQL equivalents of DB2 UDB and its syntax of usage.

1.1 Purpose

The intent of this document is to serve as a valid reference - in the near future - for the process of migrating the structure as well as data from IBM DB2 database to PostgreSQL database .

1.2 Scope

The scope of this document is limited to the extent of identifying the PostgreSQL equivalents of various SQL components, column / OLAP / Scalar functions, Order by / Group by / Having, Joins, Sub-queries, Union / Intersect / Except clauses that are currently defined for DB2 database.

2 Conversion Reference

This section briefly discusses the different steps involved in conversion process from DB2 UDB to PostgreSQL.

2.1 Tools

The following tools, could be used while migrating data from DB2 to PostgreSQL.

- Aqua Data Studio 4.5.2 and above – Mainly used for exporting DB2 data to csv format and importing csv format into postgreSQL.

2.2 SQL Components - DB2 Objects

2.2.1 Data Types

Data Types		
	IBM DB2	***PostgreSQL***
	CHAR(n)	CHAR(n)

	DATE	DATE
		Some Valid Inputs:
		`now, today, tomorrow, yesterday`
		`'now'::datetime`

	DECIMAL(m,n)	DECIMAL(m,n)
	INTEGER	INTEGER
	SMALLINT	SMALLINT
	TIMESTAMP	TIMESTAMP Some Valid Inputs: `now, today, tomorrow, yesterday`
	TIME	TIME Some Valid Inputs: `now`
	VARCHAR(n)	VARCHAR(n)

2.2.2 Special Data Types

Special Data Types		
	IBM DB2	*PostgreSQL*
	CLOB	TEXT (maximum of 1GB)

	BLOB	BYTEA (max 1GB) (Binary data - byte array)

	CURRENT TIMESTAMP	CURRENT_TIMESTAMP
		Example :
		`CREATE TABLE products (`
		`...`
		`created_date` **`TIMESTAMP DEFAULT CURRENT_TIMESTAMP`**`,`
		`...`
		`)`
		`;`

	CURRENT TIME	CURRENT_TME
		Example :
		`CREATE TABLE products (`
		`...`
		`reordered_time` **`TIMESTAMP DEFAULT CURRENT_TIME`**`,`
		`...`
		`);`

	CURRENT DATE	CURRENT_DATE
		Example :
		`CREATE TABLE products (`
		`...`
		`reordered_date` **`TIMESTAMP DEFAULT CURRENT_DATE`**`,`
		`...`
		`);`

	GENERATED BY DEFAULT AS IDENTITY	***Example :***
		`CREATE TABLE products (`
		`product_no` **`INTEGER nextval ('products_product_no_seq')`**`,`
		`...`
		`);`
		
		Using **SERIAL**
		`CREATE TABLE products (`
		`product_no` **`SERIAL`**`,`
		`...`
		`)`
		`;`

		refcursor This is special data type of CURSOR type. `DECLARE <cursor_name> refcursor;`

2.2.3 Table Constraints

2.2.3.1 Check Constraints

A check constraint is the most generic constraint type. It allows you to specify that the value in a certain column must satisfy a Boolean (truth-value) expression.

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	CREATE TABLE <table> (<column1>, ..., <columnX> **CONSTRAINT** <constraints name> **CHECK** (<Condition>));	CREATE TABLE <table> (<column1>, ..., <columnX> **CONSTRAINT** <constraints name> **CHECK** (<Condition>));
Example Usage		
	CREATE TABLE products (product_no INTEGER, name VARCHAR(30), price INTEGER, category INTEGER **CONSTRAINT** my_catg **CHECK** (category IN (1,2,3,4)));	CREATE TABLE products (product_no INTEGER, name TEXT, price INTEGER **CONSTRAINT** positive_price **CHECK** (price > 0), category INTEGER);

2.2.3.2 Not-Null Constraints

A not-null constraint simply specifies that a column must not assume the null value.

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	CREATE TABLE <table> (<column1> NOT NULL, ..., <columnX>);	CREATE TABLE <table> (<column1> NOT NULL, ..., <columnX>);
Example Usage		

	CREATE TABLE products (product_no INTEGER **NOT NULL**, name VARCHAR(30) **NOT NULL**, price INTEGER **CONSTRAINT** positive_price **CHECK** (price > 0));	CREATE TABLE products (product_no INTEGER **NOT NULL**, name TEXT **NOT NULL**, price INTEGER **CONSTRAINT** positive_price **CHECK** (price > 0));

2.2.3.3 Unique Constraints

Unique constraints ensure that the data contained in a column or a group of columns is unique with respect to all the rows in the table.

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	CREATE TABLE <table> (<column1> NOT NULL, ..., <columnX> CONSTRAINT <constraint name> UNIQUE (<column>)) DATE CAPTURE NONE IN <Data tablespace name> INDEX IN <index tablespace name> ;	CREATE TABLE <table> (<column1> NOT NULL, ..., <columnX> CONSTRAINT <constraint name> UNIQUE (<column>) USING INDEX TABLESPACE <Index tablespace name>) TABLESPACE <Data tablespace name> ;
Example Usage		
	CREATE TABLE products (product_no INTEGER **NOT NULL**, name VARCHAR(30) **NOT NULL**, price INTEGER **CONSTRAINT** positive_price **CHECK** (price > 0), **CONSTRAINT** unq_prod_no **UNIQUE** (product_no)) DATA CAPTURE NONE IN mydataspace INDEX IN myindexspace ;	CREATE TABLE products (product_no INTEGER **NOT NULL**, name TEXT **NOT NULL**, price INTEGER **CONSTRAINT** positive_price **CHECK** (price > 0), **CONSTRAINT** unq_prod_no **UNIQUE** (product_no) USING INDEX TABLESPACE myindexspace) TABLESPACE mydataspace ;

2.2.3.4 Primary Key Constraints

Technically, a primary key constraint is simply a combination of a unique constraint and a not-null constraint.

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	CREATE TABLE <table> (<column1> NOT NULL, ..., <columnX> CONSTRAINT <constraint name> PRIMARY KEY (<column>)) DATE CAPTURE NONE IN <Data tablespace name> INDEX IN <index tablespace name> ;	CREATE TABLE <table> (<column1> NOT NULL, ..., <columnX> CONSTRAINT <constraint name> PRIMARY KEY (<column>) USING INDEX TABLESPACE <Index tablespace name>) TABLESPACE <Data tablespace name> ;
Example Usage		
	CREATE TABLE products (product_no INTEGER **NOT NULL**, name VARCHAR(30) **NOT NULL**, price INTEGER **CONSTRAINT** positive_price **CHECK** (price > 0), **CONSTRAINT** pk_prod_no **PRIMARY KEY** (product_no)) DATA CAPTURE NONE IN mydataspace INDEX IN myindexspace ;	CREATE TABLE products (product_no INTEGER **NOT NULL**, name TEXT **NOT NULL**, price INTEGER **CONSTRAINT** positive_price **CHECK** (price > 0), **CONSTRAINT** pk_prod_no **PRIMARY KEY** (product_no) USING INDEX TABLESPACE myindexspace) TABLESPACE mydataspace ;

2.2.3.5 Foreign Key Constraints

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table. We say this maintains the referential integrity between two related tables.

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*

	CREATE TABLE <table> (<column1> NOT NULL, ..., <columnX> CONSTRAINT <constraint name> FOREIGN KEY (<column>) REFERENCES <ref table name> (<column>)) DATE CAPTURE NONE IN <Data tablespace name> INDEX IN <index tablespace name> ;	CREATE TABLE <table> (<column1> NOT NULL, ..., <columnX> CONSTRAINT <constraint name> FOREIGN KEY (<column>) REFERENCES <ref table name>(<column>)) TABLESPACE <Data tablespace name> ;
Example Usage		
	CREATE TABLE products (product_no INTEGER **NOT NULL**, name VARCHAR(30) **NOT NULL**, price INTEGER **CONSTRAINT** positive_price **CHECK** (price > 0), **CONSTRAINT** pk_prod_no **PRIMARY KEY** (product_no)) DATA CAPTURE NONE IN mydataspace INDEX IN myindexspace ; CREATE TABLE orders (order_no INTEGER **NOT NULL**, product_no INTEGER, quantity DECIMAL(12,4), **CONSTRAINT** fk_prod_no **FOREIGN KEY** (product_no) REFERENCES products(product_no)) DATA CAPTURE NONE IN mydataspace INDEX IN myindexspace ;	CREATE TABLE products (product_no INTEGER **NOT NULL**, name TEXT **NOT NULL**, price INTEGER **CONSTRAINT** positive_price **CHECK** (price > 0), **CONSTRAINT** pk_prod_no **PRIMARY KEY** (product_no) USING INDEX TABLESPACE myindexspace) TABLESPACE mydataspace ; CREATE TABLE orders (order_no INTEGER **NOT NULL**, product_no INTEGER, quantity DECIMAL(12,4), **CONSTRAINT** fk_prod_no **FOREIGN KEY** (product_no) REFERENCES products(product_no)) TABLESPACE mydataspace ;

2.2.4 Sequence Number (Auto generated ID column)

The data types serial and bigserial are not true types, but merely a notational convenience for setting up unique identifier columns (similar to the AUTO_INCREMENT property supported by some other databases).

The <***sequence name***> should be unique for database level and it **minvalue n**, is the number at which the sequence starts.

Note: The sequence is always incremented by 1.

The tables created are later associated with the already created sequence, using **nextval** ('<***sequence_name***>') function.

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	CREATE TABLE <table> (<column1> NOT NULL GENERATED BY DEFAULT AS IDENTITY (START WITH n, INCREMENT BY x NO CACHE), ..., <columnX>) ;	CREATE SEQUENCE <sequence_name> MINVALUE n; CREATE TABLE <table> (<column1> DEFAULT nextval ('<sequence_name>'), ..., <columnX>) ;
Example Usage		
	CREATE TABLE products (product_no INTEGER NOT NULL GENERATED BY DEFAULT AS IDENTITY (START WITH 11, INCREMENT BY 1, NO CACHE), name VARCHAR(30) **NOT NULL**, price INTEGER) ;	CREATE SEQUENCE products_seq_prdno MINVALUE 1; CREATE TABLE products (product_no INTEGER nextval (' products_seq_prdno') name TEXT **NOT NULL**, price INTEGER **CONSTRAINT** positive_price **CHECK** (price > 0), **CONSTRAINT** pk_prod_no **PRIMARY KEY** (product_no) USING INDEX TABLESPACE myindexspace) TABLESPACE mydataspace ;

2.2.5 Special Objects

2.2.5.1 CLOB

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	CLOB(n) - n <= 2 GB	TEXT (max 1GB)
Example Usage		

	`CREATE TABLE orders`
	`....`
	`notes CLOB(1M),`
	`....`
	`);`
		`CREATE TABLE orders (`
		`...`
		`notes` **`TEXT(1M)`**`,`
		`...`
		`);`

2.2.5.2 BLOB

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	BLOB(n) - n <= 2 GB	BYTEA (maximum 1GB) binary data – byte array
Example Usage		

2.2.6 Views

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*

	`CREATE VIEW <view_name> AS`
	`sql statement`
	`;`
		`CREATE OR REPLACE VIEW <view_name> AS`
		`sql statement`
		`;`

Example Usage		

	CREATE VIEW products_v AS SELECT x,y,... FROM products .... ;	CREATE OR REPLACE VIEW products_v AS SELECT x,y,... FROM products .... ;

2.2.7 Trigger

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	CREATE TRIGGER <trigger name> AFTER INSERT ON <table name> REFERENCING NEW AS N FOR EACH ROW MODE DB2SQL BEGIN ATOMIC ..... END ;	CREATE TRIGGER <trigger name> AFTER INSERT ON <table name> FOR EACH ROW EXECUTE PROCEDURE function_name();
Example Usage		

	CREATE TABLE emp_audit(operation CHAR(1) NOT NULL, ...); CREATE TRIGGER process_emp_audit AFTER INSERT ON emp_audit REFERENCING NEW AS N FOR EACH ROW MODE DB2SQL BEGIN ATOMIC INSERT INTO emp_audit SELECT 'I', now(), user, N.*; END ;	CREATE TABLE emp_audit(operation CHAR(1) NOT NULL, ...); CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER LANGUAGE plpgsql AS emp_audit BEGIN INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*; RETURN NEW; END; emp_audit; CREATE TRIGGER emp_audit AFTER INSERT ON emp_audit FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();

2.2.8 Functions

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*

	CREATE FUNCTION <function_name> (parameter, ...) SPECIFIC <function_name> RETURNS <return_data_type> NO EXTERNAL ACTION DETERMINISTIC RETURN .... ;	CREATE OR REPLACE FUNCTION <function_name> (parameter, ...) RETURNS <return_data_type> LANGUAGE PLPGSQL AS $$ BEGIN .... END; $$;
Example Usage		
	CREATE FUNCTION GREATEROF (V1 INTEGER, V2 INTEGER) SPECIFIC GREATEROF RETURNS integer LANGUAGE sql NO EXTERNAL ACTION DETERMINISTIC RETURN CASE WHEN V1 > V2 THEN V1 ELSE V2 END; ;	CREATE OR REPLACE FUNCTION GREATEROF (V1 INTEGER, V2 INTEGER) RETURNS integer LANGUAGE plpgsql AS $$ BEGIN RETURN CASE WHEN V1 > V2 THEN V1 ELSE V2 END; END; $$;

2.2.9 Stored Procedures

When creating functions which handles or returns cursors, these points are to be remembered.

- All variable declaration should be done at the top, in other words should be the first

few statements.

- Any default values assigned to the variables can be done at the declaration statement.
- Any assigning of values to the variables should be done within the BEGIN and END statement.
- Any cursor declaration can be done out side the BEGIN and END statement.
- Any dynamic cursors using dynamic sqls, should be done within BEGIN and END statement.
- In all the cases OPEN <cursor_name> and returning the cursor RETURN <cursor_name>, is a must statement for functions returning **REFCURSOR**.
- The function body block, to be defined within $$ and $$.

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	CREATE PROCEDURE <procedure_name> (IN para1 VARCHAR(5), IN para2 INTEGER) SPECIFIC <procedure_name> DYNAMIC RESULT SETS <number> LANGUAGE SQL BEGIN DECLARE <cursor_name> CURSOR WITH RETURN TO CLIENT FOR <sql_statement>; OPEN <cursor_name>; END ;	CREATE OR REPLACE FUNCTION <function_name> (IN para1 VARCHAR(5), IN para2 INTEGER) RETURNS **REFCURSOR** LANGUAGE PLPGSQL AS $$ DECLARE <cursor_name> CURSOR FOR <sql_statement>; BEGIN .... OPEN <cursor_name>; RETURN <cursor_name>; END; $$;
Example Usage		

1	CREATE PROCEDURE list_orders (IN prd_no INTEGER) SPECIFIC list_orders DYNAMIC RESULT SETS 1 LANGUAGE SQL BEGIN DECLARE lstOrds CURSOR WITH RETURN TO CLIENT FOR SELECT * FROM orders WHERE product_no = prd_no; OPEN <cursor_name>; END ;	CREATE OR REPLACE FUNCTION list_orders (IN prd_no INTEGER) RETURNS **REFCURSOR** LANGUAGE plpgsql AS $$ DECLARE lstOrds CURSOR FOR SELECT * FROM orders WHERE product_no = prd_no; BEGIN OPEN lstOrds; RETURN lstOrds; END; $$;

2	**Dynamic Cursor**: CREATE PROCEDURE list_orders (IN prd_no INTEGER) SPECIFIC list_orders DYNAMIC RESULT SETS 1 LANGUAGE SQL BEGIN DECLARE selCur CURSOR WITH RETURN TO CLIENT FOR strPrepSelSql; DECLARE sqlString VARCHAR (200); SET sqlString = ' SELECT * FROM orders WHERE product_no = ' || prd_no; PREPARE strPrepSelSql FROM sqlString; OPEN selCur; END ;	**Dynamic Cursor**: CREATE OR REPLACE FUNCTION list_orders (IN prd_no INTEGER) RETURNS **refcursor** LANGUAGE plpgsql AS $$ DECLARE sqlString VARCHAR(200); selCur **refcursor**; BEGIN sqlString = 'SELECT * FROM orders WHERE product_no = ' \|\| prd_no; OPEN selCur FOR EXECUTE sqlString; RETURN selCur; END; $$;

2.3 SQL Predicates

2.3.1 BETWEEN Predicate

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*

	SELECT x, y FROM tab1 WHERE ..... column BETWEEN value1 AND value2 ..... ;	SELECT x, y FROM tab1 WHERE column1 ..... column2 BETWEEN value1 AND value2 .... ;
Example Usage		
		SELECT * FROM orders, WHERE quantity <= 100 AND order_date BETWEEN '2005-04-06' AND '2006-04-05'; **Note**: Both the dates are inclusive, as in DB2.

2.3.2 EXISTS / NOT EXISTS Predicate

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	SELECT column(s), FROM <table_name> WHERE columnx = <value> AND NOT EXISTS (SELECT columnx FROM <table_name> ....) ;	SELECT column(s), FROM <table_name> WHERE columnx = <value> AND NOT EXISTS (SELECT columnx FROM <table_name> ....) ;
Example Usage		

		SELECT product_no FROM products WHERE name LIKE 'A%' AND category IN (1,2,3,4) AND NOT EXISTS (SELECT category_no FROM categorys WHERE status = 'D');

2.3.3 IN / NOT IN Predicate

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	SELECT * FROM <table_name> WHERE ..... <column> NOT IN ('C','S') ..... ;	SELECT * FROM <table_name> WHERE ..... <column> NOT IN ('C','S') ..... ;
Example Usage		
		SELECT product_no, name, FROM products WHERE category NOT IN (3,4);

2.3.4 LIKE Predicate

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*

	SELECT x, y FROM <table_name> WHERE ..... tab1.my_name LIKE LCASE (strName) ;	Same as DB2.
Example Usage		
		SELECT * FROM products WHERE product_no > 125 AND UPPER(name) LIKE 'M%' ;

2.3.5 IS NULL / IS NOT NULL Predicate

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	SELECT x, y FROM tab1 WHERE ..... column IS NOT NULL ;	Same as DB2.(IS NULL & IS NOT NULL)
Example Usage		
		SELECT * FROM products WHERE product_no > 125 AND category IS NOT NULL;

2.4Temporary Tables

2.4.1 Using WITH phrase at the top of the query to define a common table expression

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*

	`WITH TEMP(`
	`  name,`
	`  ....`
	`) AS (`
	`  SELECT`
	`  VALUE(id,0)`
	`  FROM`
	`  ....)`
	`;`
		Ref T121/T122. Yet to be implemented.

2.4.2 Full-Select in the FROM part of the query

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*

	`SELECT x, y`
	`FROM tab1`
	`  LEFT OUTER JOIN`
	`  (SELECT`
	`  ....`
	`  FROM`
	`  ....)`
	`WHERE`
	`...`
	`;`
		`SELECT x, y`
		`FROM tab1 A`
		`  LEFT OUTER JOIN`
		`  (SELECT *`
		`  FROM ....`
		`  ....) B`
		`ON  A.eid= B.eid`
		`WHERE B.eid < 3`
		`;`

Example Usage		

		SELECT SUM(tot_paid-tot_refund) AS tot_paid_amount, ... i.invoice_no FROM invoice i LEFT OUTER JOIN orders_pending o ON i.invoice_no = o.invoice_no AND invoice_year = '20052006'

2.4.3 Full-Select in the SELECT part of the query

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	SELECT <column_name>, (SELECT <column_name> FROM <table> WHERE column = Value) FROM <table> WHERE <condition> ;	SELECT <column_name>, (SELECT <column_name> FROM <table> WHERE column = Value) FROM <table> WHERE <condition> ;
Example Usage		

		SELECT cust_id, TO_CHAR((SELECT MAX (cf.fund_recvd_date) FROM cust_funding cf WHERE cf.er_id = iCuID ...),'YYYY-MM-DD') AS fund_date ... FROM cust_funding WHERE cust_id = iCuID AND invoice_year = '20052006' GROUP BY cust_id, invoice_year ;

2.5 CASE Expression

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	CASE ctrlVar WHEN 1 THEN <statements>; ELSE <statements>; END CASE ;	**Note :** Case expression is not supported in PostgreSQL. It can used in SELECT statements. As a workaround, use IF-ELSE construct.

2.6 Column Functions

Equivalents / Declaration		
Column / Aggregate Functions	*IBM DB2*	*PostgreSQL*

AVG	`SELECT emp_id, AVG(emp_pay) FROM emp_payments GROUP BY emp_id;`	Same as DB2
COUNT	`SELECT company_id, COUNT (emp_id) AS employee_count FROM employee GROUP BY company_id;`	Same as DB2
MAX	`SELECT emp_id, MAX (process_date) AS last_processed_date FROM emp_payments GROUP BY emp_id`	Same as DB2
MIN	`SELECT emp_id, MIN (process_date) AS first_processed_date FROM emp_payments GROUP BY emp_id`	Same as DB2
SUM	`SELECT emp_id, SUM(emp_pay) AS total_pay FROM emp_payments GROUP BY emp_id;`	Same as DB2

2.7 OLAP Functions

2.7.1 ROWNUMBER & ROLLUP

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	ROWNUMBER()	Not supported in PostgreSQL **Note** : Not used in application. Hence can be ignored.

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	ROLLUP()	There is no direct equivalent for ROLLUP in PostgreSQL database. This is could be achieved by using UNION clause. In some cases, we may end up using UNION clause along with a required VIEW.
Example Usage		

	SELECT 1 AS cur_row, cust_id, cust_name, fund_date, cust_funding AS Amount, invoice_date FROM customer c, invoice i WHERE c.cust_id = iCuID AND c.invoice_no = i.invoice_no AND c.invoice_year = '20052006' GROUP BY ROLLUP((cust_id, cust_name, cust_funding AS Amount, invoice_date)), fund_date ORDER BY cur_row, fund_date ;	SELECT * FROM (SELECT * FROM (SELECT 1 AS cur_row, cust_id, cust_name, fund_date, cust_funding AS Amount, invoice_date FROM customer c, invoice i WHERE c.cust_id = iCuID AND c.invoice_no = i.invoice_no AND c.invoice_year = '20052006') AS LST_RECS UNION SELECT COUNT(*) AS cur_row, NULL,NULL,NULL, SUM(cust_funding) AS Amount, NULL, FROM customer c,invoice i WHERE c.cust_id = iCuID AND c.invoice_no = i.invoice_no AND c.invoice_year = '20052006') AS TMP_TAB ORDER BY cur_row,fund_date ;

2.8 Scalar Functions

Scalar functions act on a single row at a time. This section lists all the IBM DB2 scalar functions that are used in Able Payroll project & their equivalents in PostgreSQL database.

2.8.1 Scalar Functions - IBM DB2 vs PostgreSQL

Scalar Function	Return Type	IBM DB2	PostgreSQL	Description

CEIL or CEILING	Same as input	CEIL
		CEILING
		Example :
		`SELECT CEIL(123.89) FROM SYSIBM.SYSDUMMY1;`
		`SELECT CEILING(123.89) FROM SYSIBM.SYSDUMMY1;`
			CEIL
			CEILING
			Example :
			`SELECT CEIL(123.89);`
			`SELECT CEILING(123.89);`
				CEIL or CEILING returns the next smallest integer value that is greater than or equal to the input (e.g. *`CEIL(123.89)`* returns *`124,`* also *`CEIL(123.19)`* returns *`124`*)

CHAR	String / Text	CHAR
		Example :
		`SELECT CHAR(1) FROM SYSIBM.SYSDUMMY1;`
		`SELECT CHAR(DATE('2005-01-12'), EUR) FROM SYSIBM.SYSDUMMY1;`
			TO_CHAR(<timestamp / interval / int / double precision / numeric type>, text)
			Example :
			`SELECT TO_CHAR(-212.8, '999D99S');`
				Returns character String of the given input

COALESCE	Null or same as input	COALESCE(value [, ...]) **Example :** `SELECT COUNT(*), MIN(MAIL_ATTACH_ID) AS min_id, MAX(MAIL_ATTACH_ID) AS max_id, COALESCE(MIN (MAIL_ATTACH_ID), MAX (MAIL_ATTACH_ID)) FROM EMAIL_ATTACH_LOG;`	COALESCE(value [, ...]) **Example : (Same as DB2)** `SELECT COUNT(*), MIN(MAIL_ATTACH_ID) AS min_id, MAX(MAIL_ATTACH_ID) AS max_id, COALESCE(MIN (MAIL_ATTACH_ID), MAX (MAIL_ATTACH_ID)) FROM EMAIL_ATTACH_LOG;`	First non-null value in a list of (compatible) input expressions (read from left to right) is returned. VALUE is a synonym for COALESCE.
CONCAT or ||	String	**Example :** `SELECT 'A' || 'B' , CONCAT('A', 'B'), 'A' || 'B' || 'C', CONCAT (CONCAT('A', 'B'), 'C');`	**Note :** CONCAT is not available in PostgreSQL, only **||** works. A function CONCAT as given below can be created as a workaround. **Function :** `CREATE OR REPLACE FUNCTION "concat" (text,text) RETURNS text LANGUAGE sql AS $$ SELECT $1 \|\| $2; $$;` **Example :** `SELECT 'A' || 'B' , CONCAT('A', 'B'), 'A' || 'B' || 'C', CONCAT(CONCAT('A', 'B'), 'C');`	Joins two strings together. In IBM DB2, CONCAT function has both "infix" and "prefix" notations. In the former case, the verb is placed between the two strings to be acted upon. In PostgreSQL, CONCAT function needs to be created in order to use it.
DATE	Date	**Example :** `SELECT DATE ('2006-09-21') FROM SYSIBM.SYSDUMMY1;`	**Example :** `SELECT TO_DATE ('21-02-2006','DD-MM-YYYY');`	Converts the input to date value

DAY	Integer	**Usage :** DAY (<DATE_FIELD>) **Example :** SELECT DAY (DATE('2006-09-21')) FROM SYSIBM.SYSDUMMY1;	**Usage :** DATE_PART('day', <DATE_FIELD>) **Example :** SELECT DATE_PART('day', '2006-09-21'::date);	Returns the day (as in day of the month) part of a date (or equivalent) value. The output format is integer.
DAYS	Integer	**Usage :** DAYS (<DATE_FIELD>) **Example :** SELECT (DAYS (DATE('2006-09-25')) - DAYS(DATE('2006-09-21'))) FROM SYSIBM.SYSDUMMY1;	**Note :** DAYS is not available in PostgreSQL. **Example :** SELECT TO_DATE ('25-09-2006', 'DD-MM-YYYY') -TO_DATE('21-09-2006', 'DD-MM-YYYY'); A function DAYS can be created as a workaround. **Function :-** CREATE OR REPLACE FUNCTION DAYS (V1 DATE) RETURNS integer LANGUAGE plpgsql AS $$ BEGIN RETURN TO_DATE(V1,'YYYY-MM-DD') - TO_DATE('4712-01-01','YYYY-MM-DD'); END; $$;	Converts a date (or equivalent) value into a number that represents the number of days since the date "0001-01-01" inclusive. The output format is integer.
DECIMAL / DEC	Decimal	**Usage :** DECIMAL(<FIELD>) or DEC(<FIELD>) **Example :** SET l_sub4 = DECIMAL(l_absSub4);	No direct equivalent. Use TO_NUMBER instead. **Example :** SELECT TO_NUMBER(l_absSub4, <format_string>);	Converts either character or numeric input to decimal.
FLOOR	Same as input	**Usage :** FLOOR(<FIELD>) **Example :** SELECT FLOOR (5.945) FROM SYSIBM.SYSDUMMY1;	**Usage :** FLOOR(<FIELD>) **Example :** SELECT FLOOR (5.945);	Returns the next largest integer value that is smaller than or equal to the input (e.g. 5.945 returns 5.000).
IDENTITY_VAL_LOCAL	Integer	**Example :** SET iErID = IDENTITY_VAL_LOCAL();	**Example :** CURRVAL ('<<SEQUENCE_NAME>>') SELECT CURRVAL ('DummySeq');	Returns the most recently assigned value (by the current user) to an identity column.

INTEGER	Integer	Converts either a number or a valid character value into an integer. The character input can have leading and /or trailing blanks, and a sign indicator, but it cannot contain a decimal point. Numeric decimal input works just fine. **Example :** `SELECT INTEGER (234.8817) FROM SYSIBM.SYSDUMMY1;`	**Example :** `TO_NUMBER (<field>, <format>)` `SELECT TO_NUMBER(FLOOR (234.8817),'999999999'); => 234`	Converts input into an integer

LCASE or LOWER	String	**Usage :** `LOWER(<FIELD>)` (or)
		`LCASE(<FIELD>)`
		Example :
		`SELECT LCASE('LOWER CASE'), LOWER('LOWER CASE') FROM SYSIBM.SYSDUMMY1;`
			Usage : `LOWER(<FIELD>)`
			Example :
			`SELECT LOWER('LOWER CASE');`
				Converts the mixed or upper case input string to lower case

LENGTH	Integer	**Usage :** `LENGTH(<FIELD>)` **Example :** `SELECT LENGTH('LOWER CASE') FROM SYSIBM.SYSDUMMY1;`	**Usage :** `LENGTH(<FIELD>)` **Example :** `SELECT LENGTH('LOWER CASE');`	Returns an integer value with the internal length of the expression

LTRIM	String	**Usage :** `LTRIM(<FIELD>)`
		Example :
		`SELECT LTRIM('  ABC'), LENGTH(LTRIM('  ABC')),`
		`LTRIM(' ABC '), LENGTH (LTRIM(' ABC ')),`
		`LTRIM('ABC  '), LENGTH (LTRIM('ABC  ')) FROM SYSIBM.SYSDUMMY1;`
			Usage : `LTRIM(<FIELD>)`
			Example :
			`SELECT LTRIM('  ABC'), LENGTH(LTRIM('  ABC')),`
			`LTRIM(' ABC '), LENGTH (LTRIM(' ABC ')),`
			`LTRIM('ABC  '), LENGTH (LTRIM('ABC  '));`
				Removes leading blanks, but not trailing blanks, from the argument.

MOD	depends on input	**Usage :** `MOD(<FIELD_1>, <FIELD_2>)` **Example :** `SELECT MOD(-31,11) FROM SYSIBM.SYSDUMMY1;`	**Usage :** `MOD(<FIELD_1>, <FIELD_2)` **Example :** `SELECT MOD(-31,11);`	Returns the remainder (modulus) for the 1st argument divided by the 2nd argument.
MONTH	Integer	**Usage :** `MONTH (<DATE_FIELD>)` **Example :** `SELECT MONTH (DATE('2006-09-21')) FROM SYSIBM.SYSDUMMY1;`	**Usage :** `DATE_PART ('MONTH', <DATE_FIELD>)` **Example :** `SELECT DATE_PART ('month', '2006-09-21'::date);`	Returns the month part of the date value. The output format is integer.

POSSTR	Integer	**Usage :** `POSSTR (<FIELD_1>, <FIELD_2>)` **Example :** `SELECT POSSTR('Benefits and Expenses', 'and') FROM SYSIBM.SYSDUMMY1;`	**Usage :** `POSITION (<FIELD_1> IN <FIELD_2>)` **Example :** `SELECT POSITION('and' IN 'Benefits and Expenses');`	Returns the position of 2nd string (DB2) / 1st string (PostgreSQL) in 1st string (DB2) / 2nd string (PostgreSQL)
RAND	Floating point values	**Usage :** `RAND()` **Example :** `SELECT RAND() FROM SYSIBM.SYSDUMMY1;`	**Usage :** `RANDOM()` **Example :** `SELECT RANDOM();`	Returns a pseudo-random floating-point value in the range of zero to one inclusive.
ROUND	Integer	**Usage :** `ROUND(<FIELD>, <precision>)` **Example :** `SELECT ROUND(216.89, 1) FROM SYSIBM.SYSDUMMY1;`	**Usage :** `ROUND(<FIELD>, <precision>)` **Example :** `SELECT ROUND(216.89, 1);`	Rounds the rightmost digits of number (1st argument). If the second argument is positive, it rounds to the right of the decimal place. If the second argument is negative, it rounds to the left. A second argument of zero results rounds to integer.

RTRIM	String	**Usage :** `RTRIM (<TEXT_FIELD>)`
		Example :
		`SELECT RTRIM('  ABC'), LENGTH(RTRIM('  ABC')),`
		`RTRIM(' ABC '), LENGTH (RTRIM(' ABC ')),`
		`RTRIM('ABC  '), LENGTH (RTRIM('ABC  ')) FROM SYSIBM.SYSDUMMY1;`
			Usage : `RTRIM (<TEXT_FIELD>)`
			Example :
			`SELECT RTRIM('  ABC'), LENGTH(RTRIM('  ABC')),`
			`RTRIM(' ABC '), LENGTH (RTRIM(' ABC ')),`
			`RTRIM('ABC  '), LENGTH (RTRIM('ABC  '));`
				Removes trailing blanks, but not leading blanks, from the argument.

SMALLINT	Integer	Converts either a number or a valid character value into a smallint value. **Example :** `SELECT SMALLINT(219.89) FROM SYSIBM.SYSDUMMY1;`	**Example :** `TO_NUMBER (<field>, <format>)` `SELECT TO_NUMBER(FLOOR (234.8817),'999999999'); => 234`	
SUBSTR	String	**Usage :** `SUBSTR (<TEXT_FIELD>, <int_position>)` **Example :** `SELECT SUBSTR('This is a substring test', 9) FROM SYSIBM.SYSDUMMY1;`	**Usage :** `SUBSTR (<TEXT_FIELD>, <int_position>)` **Example :** `SELECT SUBSTR('This is a substring test', 9);`	Returns part of a string. If the length is not provided, the output is from the start value to the end of the string.

TIMESTAMP	Timestamp	**Usage :** `TIMESTAMP (<FIELD>)`
		Example :
		`SELECT TIMESTAMP('2006-01-31-22.44.55.000000'),`
		`TIMESTAMP('2006-01-31-22.44.55.000'),`
		`TIMESTAMP('2006-01-31-22.44.55'),`
		`TIMESTAMP ('20060131224455'),`
		`TIMESTAMP('2006-01-31','22.44.55')`
		`FROM SYSIBM.SYSDUMMY1;`
		Example : (to get the default timestamp)
		`SELECT CURRENT TIMESTAMP FROM SYSIBM.SYSDUMMY1;`
			Usage : `TO_TIMESTAMP (<FIELD>, <format>)`
			`When using as default to a column, in table definition.`
			Default Timestamp : `CURRENT_TIMESTAMP`
			Example :
			`SELECT TO_TIMESTAMP ('2006-01-31-22.44.55.000000', 'YYYY-MM-DD-HH.MI.SS.MS'),`
			`TO_TIMESTAMP('2006-01-31-22.44.55.000', 'YYYY-MM-DD-HH.MI.SS.MS'),`
			`TO_TIMESTAMP('2006-01-31-22.44.55', 'YYYY-MM-DD-HH.MI.SS'),`
			`TO_TIMESTAMP ('20060131224455', 'YYYYMMDDHHMISSMS');`
			Example : (to get the default timestamp)
			`SELECT CURRENT_TIMESTAMP;`
				Converts the input into a time value.

UPPER	String	**Usage :** `UPPER (<TEXT_FIELD>)` **Example :** `SELECT UCASE('upper case'), UPPER('upper case') FROM SYSIBM.SYSDUMMY1;`	**Usage :** `UPPER (<TEXT_FIELD>)` **Example :** `SELECT UPPER('upper case');`	Converts the mixed or lower case input string to upper case
VALUE	Null or same as input	**Usage :** Same as COALESCE	**Usage :** Same as COALESCE **Refer to COALESCE example usage**	In PostgreSQL, there is no direct equivalent for VALUE function. Use COALESCE instead
YEAR	Integer	**Usage :** `YEAR (<DATE_FIELD>)` **Example :** `SELECT YEAR (DATE('2006-09-21')) FROM SYSIBM.SYSDUMMY1;`	**Usage :** `DATE_PART ('YEAR',<DATE_FIELD>);` `SELECT DATE_PART('year', '2006-09-21'::date);`	Returns the year part of a date value. The output format is integer.

2.9 ORDER BY, GROUP BY & HAVING

2.9.1 ORDER BY

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	SELECT .... <column> .... FROM <table(s)> WHERE <condition(s)> ..... ORDER BY <column(s)> ;	Same as DB2

2.9.2 GROUP BY

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	SELECT Aggregate_fun(column1), Aggregate_fun(column2), <column> FROM <table(s)> WHERE <condition(s)> GROUP BY <column> ;	Same as DB2

2.9.3 HAVING

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*

	SELECT Aggregate_fun(column1), Aggregate_fun(column2), <column> FROM <table(s)> WHERE <condition(s)> GROUP BY <column> HAVING <condition> ;	Same as DB2

2.10 DYNAMIC Cursors

In case of defining a dynamic cursor, we need to use **refcursor** special data type object.

The sample declaration is as follows:

In this sample, we assume the below code is part of a function and the function returns **refcursor** special data type and have the following input parameters:

```
sYear VARCHAR(10),
iCuID INTEGER
```

```
....
$$
DECLARE
        sqlString VARCHAR(500);
        selCur refcursor;

BEGIN
        sqlString = 'SELECT product_no,name ' ||
                'FROM products ' ||
                'WHERE product_no IN (SELECT product_no ' ||
                        'FROM invoice WHERE cust_id = ' || iCuID || ') ' ||
                        'AND invoice_year = ''' || sYear || ''') ' ||
                'ORDER BY product_no';

        OPEN selCur FOR EXECUTE sqlString;
        RETURN selCur;
END
;
$$
```

2.11 Joins

2.11.1 Self-Join

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*

	`SELECT a.emp_id, a.company_id, b.user_id FROM employee a INNER JOIN employee b ON a.emp_id= b.emp_id;`
	`(or)`
	`SELECT a.emp_id, a.company_id, b.user_id FROM employee a, employee b WHERE a.emp_id= b.emp_id;`
		Same as DB2

2.11.2 Left-outer Join

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	`SELECT a.company_id, a.company_name, b.emp_id, b.company_id FROM company a LEFT OUTER JOIN employee b ON a.company_id= b.company_id;`	Same as DB2

2.11.3 Right-outer Join

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	`SELECT a.company_id, a.company_name, b.emp_id, b.company_id FROM company a RIGHT OUTER JOIN employee b ON a.company_id= b.company_id;`	Same as DB2

2.12 Sub-Query

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*

	SELECT title, fname, sname, forename FROM employee WHERE emp_id IN (SELECT emp_id FROM department WHERE company_id = iCID);	Same as DB2

2.13 Manipulating Resultset returned by Called Function (Associate..)

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	DECLARE result RESULT_SET_LOCATOR VARYING; CALL procedure(<params>); ASSOCIATE RESULT SET LOCATORS (result) WITH PROCEDURE procedure; ALLOCATE cursor CURSOR FOR RESULT SET result; FETCH FROM cursor INTO <var list>;	DECLARE cursor REFCURSOR; cursor := SELECT function_returning_cursor(); FETCH ALL IN cursor; or FETCH cursor INTO <var list>;
Example Usage		

1	DECLARE result1 RESULT_SET_LOCATOR VARYING; CALL SFT_STY_1 (strProcessTaxYear); ASSOCIATE RESULT SET LOCATORS (result1) WITH PROCEDURE SFT_STY_1; ALLOCATE rsCur CURSOR FOR RESULT SET result1; FETCH FROM rsCur INTO var1, var2; CLOSE rsCur;	CREATE OR REPLACE FUNCTION func_select() RETURNS refcursor; LANGUAGE plpgsql; AS $$ DECLARE ref refcursor; BEGIN OPEN ref FOR SELECT 'JOHN' AS name; RETURN ref; END; $$; CREATE OR REPLACE FUNCTION func_fectch() RETURNS refcursor; LANGUAGE plpgsql; AS $$ BEGIN DECLARE rsCur REFCURSOR; rsCur := SELECT func_select (); FETCH cursor INTO myname; ... CLOSE rsCur;. END; $$;

2		Using bound cursor name, that is cursor name specified. CREATE TABLE test (col text); INSERT INTO test VALUES (’123’); CREATE FUNCTION reffunc(refcursor) RETURNS refcursor LANGUAGE plpgsql AS $$ BEGIN OPEN $1 FOR SELECT col FROM test; RETURN $1; END; $$; BEGIN; SELECT reffunc (’funccursor’); FETCH ALL IN funccursor; COMMIT;

3		Using unbound cursor, that is cursor does not have a name, reference is automatically generated.. CREATE FUNCTION reffunc2() RETURNS refcursor LANGUAGE plpgsql AS $$ DECLARE ref refcursor; BEGIN OPEN ref FOR SELECT col FROM test; RETURN ref; END; $$; BEGIN; SELECT reffunc2(); ***on screen message:*** reffunc2 -------------------- <unnamed cursor 1> (1 row) FETCH ALL IN "<unnamed cursor 1>"; COMMIT ;

4		Function returning multiple cursors. CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor LANGUAGE plpgsql AS $$ BEGIN OPEN $1 FOR SELECT * FROM table_1; RETURN NEXT $1; OPEN $2 FOR SELECT * FROM table_2; RETURN NEXT $2; END; $$; -- need to be in a transaction to use cursors. BEGIN; SELECT * FROM myfunc('a', 'b'); FETCH ALL FROM a; FETCH ALL FROM b; COMMIT;

2.14 UNION & UNION ALL

2.14.1 UNION

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*

	SELECT emp_id, pay_amt FROM emp_payments UNION SELECT emp_id, pay_amt FROM emp_absent_payments	Same as DB2

2.14.2 UNION ALL

Equivalents / Declaration		
	IBM DB2	*PostgreSQL*
	SELECT emp_id, pay_amt FROM emp_payments UNION ALL SELECT emp_id, pay_amt FROM emp_absent_payments	Same as DB2 (duplicate rows also will be fetched)

2.15 Dynamic SQL

```
.....
RETURNS refcursor
LANGUAGE plpgsql
AS
$$
       DECLARE
              sqlString1 VARCHAR(500);
              sqlString2 VARCHAR(500);
              selCur refcursor;
       BEGIN
              sqlString1 = 'SELECT code, list_code, short_description,
description ' ||
                     'FROM department ' ||
                     'WHERE code = ''' || strCode || '''';


              sqlString2 = 'SELECT code, list_code, short_description,
description ' ||
                     'FROM payment_master ' ||
                     'WHERE code IN (''' || strCode || ''')';


              IF iwhichCursor = 1 THEN
                     OPEN selCur FOR EXECUTE sqlString1;
                     RETURN selCur;
              ELSEIF iwhichCursor = 2 THEN
                     OPEN selCur FOR EXECUTE sqlString2;
                     RETURN selCur;
              END IF;
       END;
$$
;
```

2.16 Condition Handling

```
EXCEPTION
       WHEN division_by_zero or UNIQUE_VIOLATION THEN
       RAISE NOTICE 'caught division_by_zero';
       RETURN x;
END;
```

Where division_by_zero is a condition which when occurs it comes to the exception block to execute it.

2.17 Print Output Messages

```
RAISE NOTICE 'Print any message ';
```

2.18 Implicit casting in SQL

2.18.1Casting double to integer syntax

```
SELECT Double_variable::INTEGER;
SELECT 235.22::INTEGER;
```

2.18.2Casting double to integer (Round)

```
SELECT 235.674::INTEGER;
```
This rounds the value to 236.

2.18.3Casting double to integer (lower possible integer)

To cast it to the lower possible integer, use Floor function.
```
SELECT FLOOR(235.22)::INTEGER;
```

2.19 Select from SYSIBM.SYSDUMMY1

There is no “SYSIBM.SYSDUMMY1” table equivalent in PostgreSQL. Unlike other RDBMS, PostgreSQL allows a “select” without the ”from” clause.

```
SELECT FLOOR(42.2);
```

2.20 Variables declaration and assignment

Syntax

```
DECLARE  <<Variable_name>> DATATYPE DEFUALT <<DEFUALT_VAL>>;
DECLARE  iMaxLen  INTEGER DEFAULT 0;
```

2.21 Conditional statements and flow control (supported by PostgreSQL)

2.21.1 IF – THEN – END IF

```
IF <boolean-expression> THEN
       <statements>
END IF;
```

2.21.2 IF – THEN – ELSE – END IF

```
IF <boolean-expression> THEN
       <statements>
ELSE
       <statements>
END IF;
```

2.21.3 IF – THEN – ELSE IF – END IF

IF statements can be nested, as in the following example:

```
IF temp.val = 'm' THEN
       gender := 'man';
ELSE
       IF temp.val = 'f' THEN
             gender := 'woman';
       END IF;
END IF;
```

2.21.4 IF – THEN – ELSIF – THEN – ELSE

```
IF <boolean-expression> THEN
       <statements>
[ ELSIF <boolean-expression> THEN
       <statements>
[ ELSIF <boolean-expression> THEN
       <statements>
...]]
[ ELSE
       <statements> ]
END IF;
```

2.21.5 IF – THEN – ELSEIF – THEN – ELSE

ELSEIF is an alias for ELSIF & the usage is same as mentioned under IF – THEN – ELSIF – THEN – ELSE clause

2.21.6 LOOP – statement – END LOOP

```
[ <<label>> ]
LOOP
statements
END LOOP [ label ];
```

2.21.7 WHILE condition – LOOP – END LOOP

```
[ <<label>> ]
WHILE expression LOOP
statements
END LOOP [ label ];
```

3 Summary

Based on the initial experiment, the above similarities & differences are observed between IBM DB2 & PostgreSQL. The scope of this initial exercise is restricted only to the extent of directly checking all the IBM DB2 features & their PostgreSQL equivalents at the database level. We may not though rule out the possibility of any unforeseen issues that can occur at the time of testing the same from the application layer.