DB2 UDB TO PostgreSQL Conversion Guide

DB2 UDB To PostgreSQL Migration

DRAFT VERSION: 1.0
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.16 Condition Handling</td>
<td>41</td>
</tr>
<tr>
<td>2.17 Print Output Messages</td>
<td>42</td>
</tr>
<tr>
<td>2.18 Implicit casting in SQL</td>
<td>42</td>
</tr>
<tr>
<td>2.18.1 Casting double to integer syntax</td>
<td>42</td>
</tr>
<tr>
<td>2.18.2 Casting double to integer (Round)</td>
<td>42</td>
</tr>
<tr>
<td>2.18.3 Casting double to integer (lower possible integer)</td>
<td>42</td>
</tr>
<tr>
<td>2.19 Select from SYSIBM.SYSDUMMY1</td>
<td>42</td>
</tr>
<tr>
<td>2.20 Variables declaration and assignment</td>
<td>42</td>
</tr>
<tr>
<td>2.21 Conditional statements and flow control (supported by PostgreSQL)</td>
<td>42</td>
</tr>
</tbody>
</table>

3 Summary ........................................................................................................... 44
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

<table>
<thead>
<tr>
<th>Data Types</th>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
<td></td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td>Some Valid Inputs: now, today, tomorrow, yesterday</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘now’::datetime</td>
</tr>
<tr>
<td>DECIMAL(m,n)</td>
<td>DECIMAL(m,n)</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
<td>Some Valid Inputs: now, today, tomorrow, yesterday</td>
</tr>
<tr>
<td>TIME</td>
<td>TIME</td>
<td>Some Valid Inputs: now</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>VARCHAR(n)</td>
<td></td>
</tr>
</tbody>
</table>

2.2.2 Special Data Types

<table>
<thead>
<tr>
<th>Special Data Types</th>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOB</td>
<td>TEXT (maximum of 1GB)</td>
<td></td>
</tr>
<tr>
<td>BLOB</td>
<td>BYTEA (max 1GB) (Binary data - byte array)</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
</tr>
</tbody>
</table>
| CURRENT_TIMESTAMP | CURRENT_TIMESTAMP  
**Example:**  
CREATE TABLE products (  
  ...  
  created_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
  ...  
) ; |
| CURRENT_TIME | CURRENT_TIME  
**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 | 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

<table>
<thead>
<tr>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
</table>
| CREATE TABLE <table> (  
  <column1>,  
  ...,  
  <columnX> CONSTRAINT <constrain name> CHECK (<Condition>)  
); | CREATE TABLE <table> (  
  <column1>,  
  ...,  
  <columnX> CONSTRAINT <constrain 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

<table>
<thead>
<tr>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
</table>
| CREATE TABLE <table> (  
  <column1> NOT NULL,  
  ...,  
  <columnX>  
); | CREATE TABLE <table> (  
  <column1> NOT NULL,  
  ...,  
  <columnX>  
); |

### Example Usage
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> (     | CREATE TABLE <table> (     |
|   <column1> NOT NULL,     |   <column1> NOT NULL,     |
|   ....,                   |   ....,                   |
|   <columnX>               |   <columnX>               |
|   CONSTRAINT <constraint name> UNIQUE (<column>) | CONSTRAINT <constraint name> UNIQUE (<column>) USING INDEX TABLESPACE <Index tablespace name> |
| ) DATA CAPTURE NONE IN <Data tablespace name> INDEX IN <index tablespace name>; | ) TABLESPACE <Data tablespace name> |

| Example Usage                |  |
|----------------------------|  |
| CREATE TABLE products (     | CREATE TABLE products (     |
|   product_no INTEGER NOT NULL, |   product_no INTEGER NOT NULL, |
|   name VARCHAR(30) NOT NULL, |   name TEXT NOT NULL, |
|   price INTEGER CONSTRAINT positive_price CHECK (price > 0), |   price INTEGER CONSTRAINT positive_price CHECK (price > 0), |
|   CONSTRAINT unq_prod_no UNIQUE (product_no) | CONSTRAINT unq_prod_no UNIQUE (product_no) USING INDEX TABLESPACE myindexspace |
| ) DATA CAPTURE NONE IN mydataspase INDEX IN myindexspace; | ) TABLESPACE mydataspase; |
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**

<table>
<thead>
<tr>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
</table>
| 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 <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 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**

<table>
<thead>
<tr>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
</table>
| 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 <Data tablespace name>  
;  

| 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 mydataspace  
;
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 mydataspacespace 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 mydataspacespace 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 mydataspacespace  
;

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 mydataspacespace  
;

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 its `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

<table>
<thead>
<tr>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>`CREATE TABLE &lt;table&gt; (</td>
<td>`CREATE SEQUENCE &lt;sequence_name&gt; minvalue n;</td>
</tr>
<tr>
<td>&lt;column1&gt; NOT NULL GENERATED BY DEFAULT AS IDENTITY (START WITH n,</td>
<td>`CREATE TABLE &lt;table&gt; (</td>
</tr>
<tr>
<td>INCREMENT BY x NO CACHE),</td>
<td>&lt;column1&gt; DEFAULT nextval ('&lt;sequence_name&gt;',</td>
</tr>
<tr>
<td>....,</td>
<td>....,</td>
</tr>
<tr>
<td>&lt;columnX&gt;</td>
<td>&lt;columnX&gt;</td>
</tr>
<tr>
<td>)</td>
<td>)</td>
</tr>
<tr>
<td>;</td>
<td>;</td>
</tr>
</tbody>
</table>

### Example Usage

```sql
CREATE TABLE products (                                                  | CREATE SEQUENCE products_seqprdno minvalue 1;                                              |
   product_no INTEGER NOT NULL GENERATED BY DEFAULT AS IDENTITY (START | CREATE TABLE products (                                                                  |
   WITH 11, INCREMENT BY 1, NO CACHE),                                     |   product_no INTEGER nextval ('products_seqprdno')                                        |
   name VARCHAR(30) NOT NULL,                                              |   name TEXT NOT NULL,                                                                     |
   price INTEGER                                                           |   price INTEGER CONSTRAINT positive_price CHECK (price > 0),                              |
 )                                                                      | CONSTRAINT pk_prod_no PRIMARY KEY (product_no) USING INDEX TABLESPACE myindexspace       |
;                                                                        | )                                                                                         |
```
2.2.5 Special Objects

2.2.5.1 CLOB

<table>
<thead>
<tr>
<th>Equivalents / Declaration</th>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOB(n) - n &lt;= 2 GB</td>
<td>TEXT (max 1GB)</td>
<td></td>
</tr>
</tbody>
</table>

Example Usage

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

2.2.5.2 BLOB

<table>
<thead>
<tr>
<th>Equivalents / Declaration</th>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB(n) - n &lt;= 2 GB</td>
<td>BYTEA (maximum 1GB) binary data – byte array</td>
<td></td>
</tr>
</tbody>
</table>

Example Usage

2.2.6 Views

<table>
<thead>
<tr>
<th>Equivalents / Declaration</th>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE VIEW &lt;view_name&gt; AS sql statement ;</td>
<td>CREATE OR REPLACE VIEW &lt;view_name&gt; AS sql statement ;</td>
<td></td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
</table>
| 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()
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

<table>
<thead>
<tr>
<th>Equivalents / Declaration</th>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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**

```sql
CREATE FUNCTION GREATEROF (  
    V1 INTEGER,  
    V2 INTEGER  
)  
SPECIFIC GREATEROF  
RETURNS integer  
LANGUAGE plpgsql  
NO EXTERNAL ACTION  
DETERMINISTIC  
RETURN  
    CASE  
    WHEN V1 > V2 THEN  
    V1  
    ELSE V2  
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 outside 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**

<table>
<thead>
<tr>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE PROCEDURE &lt;procedure_name&gt; (</td>
<td>CREATE OR REPLACE FUNCTION &lt;function_name&gt; (</td>
</tr>
<tr>
<td>IN para1 VARCHAR(5),</td>
<td>IN para1 VARCHAR(5),</td>
</tr>
<tr>
<td>IN para2 INTEGER</td>
<td>IN para2 INTEGER</td>
</tr>
<tr>
<td>)</td>
<td>)</td>
</tr>
<tr>
<td>SPECIFIC &lt;procedure_name&gt;</td>
<td>RETURNS REFCURSOR</td>
</tr>
<tr>
<td>DYNAMIC RESULT SETS &lt;number&gt;</td>
<td>LANGUAGE PLPGSQL</td>
</tr>
<tr>
<td>LANGUAGE SQL</td>
<td>AS</td>
</tr>
<tr>
<td>BEGIN</td>
<td>$$</td>
</tr>
</tbody>
</table>
| DECLARE <cursor_name> CURSOR WITH RETURN TO CLIENT FOR <sql_statement>; | DECLARE <cursor_name> CURSOR FOR <sql_statement>;
| OPEN <cursor_name>; | BEGIN |
| END | .... |
| ; | OPEN <cursor_name>;
| | RETURN <cursor_name>;
| | END;
| | $$ |
| | ; |

**Example Usage**
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;
$$
;
**Dynamic Cursor:**

```sql
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;  
```

```sql
CREATE OR REPLACE FUNCTION list_orders (  
    IN prd_no INTEGER  
)  
RETTURNS refcursor  
LANGUAGE plpgsql  
AS  
$$  
DECLARE sqlString VARCHAR(200);  
    selCur refcursor;  
BEGIN  
    sqlString = 'SELECT * FROM orders WHERE product_no = ' || prd_no;  
    PREPARE strPrepSelSql FROM sqlString;  
    OPEN selCur FOR EXECUTE  
    sqlString;  
    RETURN selCur;  
END;  
$$  
;  
```

---

### 2.3 SQL Predicates

#### 2.3.1 BETWEEN Predicate

<table>
<thead>
<tr>
<th>Equivalents / Declaration</th>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Example Usage

```sql
SELECT x, y
FROM tab1
WHERE
    ...
    column BETWEEN value1 AND value2
    ...
;

SELECT x, y
FROM tab1
WHERE
    ...
    column1
    ...
    column2 BETWEEN value1 AND value2
    ...
;
```

**Note:** Both the dates are inclusive, as in DB2.

### 2.3.2 EXISTS / NOT EXISTS Predicate

#### Equivalents / Declaration

<table>
<thead>
<tr>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>SELECT</td>
</tr>
<tr>
<td>column(s),</td>
<td>column(s),</td>
</tr>
<tr>
<td>FROM &lt;table_name&gt;</td>
<td>FROM &lt;table_name&gt;</td>
</tr>
<tr>
<td>WHERE</td>
<td>WHERE</td>
</tr>
<tr>
<td>columnx = &lt;value&gt;</td>
<td>columnx = &lt;value&gt;</td>
</tr>
<tr>
<td>AND NOT EXISTS (SELECT columnx</td>
<td>AND NOT EXISTS (SELECT columnx</td>
</tr>
<tr>
<td>FROM &lt;table_name&gt;</td>
<td>FROM &lt;table_name&gt;</td>
</tr>
<tr>
<td>....)</td>
<td>....)</td>
</tr>
<tr>
<td>;</td>
<td>;</td>
</tr>
</tbody>
</table>

**Example Usage**

```sql
SELECT *
FROM orders,
WHERE
    quantity <= 100
    AND order_date BETWEEN '2005-04-06' AND '2006-04-05';
```
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

<table>
<thead>
<tr>
<th>Equivalents / Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IBM DB2</strong></td>
</tr>
</tbody>
</table>
| 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);
SELECT x, y
FROM <table_name>
WHERE
    ....
    tab1.my_name LIKE LCASE (strName)
;

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

<table>
<thead>
<tr>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT x, y</td>
<td>Same as DB2.(IS NULL &amp; IS NOT NULL)</td>
</tr>
<tr>
<td>FROM tab1</td>
<td></td>
</tr>
<tr>
<td>WHERE</td>
<td></td>
</tr>
<tr>
<td>....</td>
<td></td>
</tr>
<tr>
<td>column IS NOT NULL</td>
<td></td>
</tr>
<tr>
<td>;</td>
<td></td>
</tr>
</tbody>
</table>

Example Usage

SELECT *
FROM products
WHERE product_no > 125
    AND category IS NOT NULL;

2.4 Temporary Tables

2.4.1 Using WITH phrase at the top of the query to define a common table expression
### Equivalents / Declaration

<table>
<thead>
<tr>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>WITH TEMP (</td>
<td>Ref T121/T122. Yet to be implemented.</td>
</tr>
<tr>
<td>name,</td>
<td></td>
</tr>
<tr>
<td>....</td>
<td></td>
</tr>
<tr>
<td>) AS (</td>
<td></td>
</tr>
<tr>
<td>SELECT</td>
<td></td>
</tr>
<tr>
<td>VALUE(id,0)</td>
<td></td>
</tr>
<tr>
<td>FROM</td>
<td></td>
</tr>
<tr>
<td>....)</td>
<td></td>
</tr>
<tr>
<td>;</td>
<td></td>
</tr>
</tbody>
</table>

2.4.2 Full-Select in the FROM part of the query

### Equivalents / Declaration

<table>
<thead>
<tr>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT x, y</td>
<td>SELECT x, y</td>
</tr>
<tr>
<td>FROM tab1</td>
<td>FROM tab1 A</td>
</tr>
<tr>
<td>LEFT OUTER JOIN</td>
<td>LEFT OUTER JOIN</td>
</tr>
<tr>
<td>(SELECT</td>
<td>(SELECT *</td>
</tr>
<tr>
<td>....</td>
<td>FROM ....</td>
</tr>
<tr>
<td>FROM</td>
<td>....) B</td>
</tr>
<tr>
<td>....)</td>
<td>ON A.eid= B.eid</td>
</tr>
<tr>
<td>WHERE</td>
<td>WHERE B.eid &lt; 3</td>
</tr>
<tr>
<td>...</td>
<td>;</td>
</tr>
<tr>
<td>;</td>
<td></td>
</tr>
</tbody>
</table>

### Example Usage
2.4.3 Full-Select in the SELECT part of the query

<table>
<thead>
<tr>
<th>Equivalents / Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IBM DB2</strong></td>
</tr>
<tr>
<td>SELECT</td>
</tr>
<tr>
<td>&lt;column_name&gt;,</td>
</tr>
<tr>
<td>(SELECT &lt;column_name&gt;</td>
</tr>
<tr>
<td>FROM &lt;table&gt;</td>
</tr>
<tr>
<td>WHERE column = Value)</td>
</tr>
<tr>
<td>FROM</td>
</tr>
<tr>
<td>&lt;table&gt;</td>
</tr>
<tr>
<td>WHERE</td>
</tr>
<tr>
<td>&lt;condition&gt;</td>
</tr>
<tr>
<td>;</td>
</tr>
</tbody>
</table>

**Example Usage**

```sql
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'
```
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

<table>
<thead>
<tr>
<th>IBM DB2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE ctrlVar</td>
</tr>
<tr>
<td>WHEN 1 THEN</td>
</tr>
<tr>
<td>&lt;statements&gt;;</td>
</tr>
<tr>
<td>ELSE &lt;statements&gt;;</td>
</tr>
<tr>
<td>END CASE</td>
</tr>
<tr>
<td>;</td>
</tr>
</tbody>
</table>

Note: Case expression is not supported in PostgreSQL. It can be used in SELECT statements. As a workaround, use IF-ELSE construct.

2.6 Column Functions

<table>
<thead>
<tr>
<th>Column / Aggregate Functions</th>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
</table>
### AVG

<table>
<thead>
<tr>
<th>SQL (DB2)</th>
<th>SQL (PostgreSQL)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT emp_id, AVG(emp_pay) FROM emp_payments GROUP BY emp_id;</td>
<td>Same as DB2</td>
<td></td>
</tr>
</tbody>
</table>

### COUNT

<table>
<thead>
<tr>
<th>SQL (DB2)</th>
<th>SQL (PostgreSQL)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT company_id, COUNT(emp_id) AS employee_count FROM employee GROUP BY company_id;</td>
<td>Same as DB2</td>
<td></td>
</tr>
</tbody>
</table>

### MAX

<table>
<thead>
<tr>
<th>SQL (DB2)</th>
<th>SQL (PostgreSQL)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT emp_id, MAX(process_date) AS last_processed_date FROM emp_payments GROUP BY emp_id;</td>
<td>Same as DB2</td>
<td></td>
</tr>
</tbody>
</table>

### MIN

<table>
<thead>
<tr>
<th>SQL (DB2)</th>
<th>SQL (PostgreSQL)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT emp_id, MIN(process_date) AS first_processed_date FROM emp_payments GROUP BY emp_id;</td>
<td>Same as DB2</td>
<td></td>
</tr>
</tbody>
</table>

### SUM

<table>
<thead>
<tr>
<th>SQL (DB2)</th>
<th>SQL (PostgreSQL)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT emp_id, SUM(emp_pay) AS total_pay FROM emp_payments GROUP BY emp_id;</td>
<td>Same as DB2</td>
<td></td>
</tr>
</tbody>
</table>

## 2.7 OLAP Functions

### 2.7.1 ROWNUMBER & ROLLUP

#### Equivalents / Declaration

<table>
<thead>
<tr>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROWNUMBER()</td>
<td>Not supported in PostgreSQL</td>
</tr>
<tr>
<td><strong>Note</strong>: Not used in application. Hence can be ignored.</td>
<td></td>
</tr>
</tbody>
</table>

#### Equivalents / Declaration

<table>
<thead>
<tr>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROLLUP()</td>
<td>There is no direct equivalent for ROLLUP in PostgreSQL database.</td>
</tr>
<tr>
<td>This is could be achieved by using UNION clause. In some cases, we may end up using UNION clause along with a required VIEW.</td>
<td></td>
</tr>
</tbody>
</table>

#### Example Usage
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
<table>
<thead>
<tr>
<th>Scalar Function</th>
<th>Return Type</th>
<th>IBM DB2</th>
<th>PostgreSQL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEIL or CEILING</td>
<td>Same as input</td>
<td>CEIL CEILING</td>
<td>CEIL CEILING</td>
<td>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)</td>
</tr>
<tr>
<td>CHAR</td>
<td>String / Text</td>
<td>CHAR</td>
<td>TO_CHAR(&lt;timestamp / interval / int / double precision / numeric type&gt;, text)</td>
<td>Returns character String of the given input</td>
</tr>
<tr>
<td>COALESCE</td>
<td>Null or same as input</td>
<td>COALESCE(value [...])</td>
<td>COALESCE(value [...])</td>
<td>First non-null value in a list of (compatible) input expressions (read from left to right) is returned. VALUE is a synonym for COALESCE.</td>
</tr>
<tr>
<td>CONCAT or</td>
<td></td>
<td></td>
<td>String</td>
<td>Example:</td>
</tr>
<tr>
<td>DATE</td>
<td>Date</td>
<td>Example:</td>
<td>Example:</td>
<td>Converts the input to date value</td>
</tr>
<tr>
<td>Type</td>
<td>Usage</td>
<td>Example</td>
<td>Notes</td>
<td>Output</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------</td>
<td>--------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>DAY</td>
<td><strong>Usage</strong>: DAY</td>
<td><strong>Example</strong>: SELECT DAY (DATE('2006-09-21')) FROM SYSIBM.SYSDUMMY1;</td>
<td>Returns the day (as in day of the month) part of a date (or equivalent) value. The output format is integer.</td>
<td></td>
</tr>
<tr>
<td>DAYS</td>
<td><strong>Usage</strong>: DAYS</td>
<td><strong>Example</strong>: SELECT (DAYS (DATE('2006-09-25')) - DAYS (DATE('2006-09-21'))) FROM SYSIBM.SYSDUMMY1;</td>
<td>A function DAYS can be created as a workaround.</td>
<td></td>
</tr>
<tr>
<td>DECIMAL / DEC</td>
<td><strong>Usage</strong>: DECIMAL(&lt;FIELD&gt;) or DEC(&lt;FIELD&gt;)</td>
<td><strong>Example</strong>: SET l_sub4 = DECIMAL(l_absSub4);</td>
<td>No direct equivalent. Use TO_NUMBER instead.</td>
<td></td>
</tr>
<tr>
<td>FLOOR</td>
<td><strong>Usage</strong>: FLOOR(&lt;FIELD&gt;)</td>
<td><strong>Example</strong>: SELECT FLOOR (5.945) FROM SYSIBM.SYSDUMMY1;</td>
<td>Returns the next largest integer value that is smaller than or equal to the input (e.g. 5.945 returns 5.000).</td>
<td></td>
</tr>
<tr>
<td>IDENTITY_VAL_LOCAL</td>
<td><strong>Example</strong>: SET iErID = IDENTITY_VAL_LOCAL();</td>
<td><strong>Example</strong>: CURRVAL ('&lt;&lt;SEQUENCE_NAME&gt;&gt;') SELECT CURRVAL ('DummySeq');</td>
<td>Returns the most recently assigned value (by the current user) to an identity column.</td>
<td></td>
</tr>
</tbody>
</table>
| **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**: `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')) FROM SYSIBM.SYSDUMMY1;` | **Usage**: `LTRIM(<FIELD>)`  
**Example**:  
`SELECT LTRIM(' ABC'), LENGTH(LTRIM(' ABC')), LTRIM(' ABC'), LENGTH(LTRIM(' ABC'));` | Removes leading blanks, but not trailing blanks, from the argument. |
| **MOD** | depend 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. |
<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Usage</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSSTR</td>
<td>Integer</td>
<td>Usage: POSSTR (&lt;FIELD_1&gt;, &lt;FIELD_2&gt;)</td>
<td>Example: SELECT POSSTR('Benefits and Expenses', 'and') FROM SYSIBM.SYSDUMMY1;</td>
<td>Returns the position of 2&lt;sup&gt;nd&lt;/sup&gt; string (DB2) / 1&lt;sup&gt;st&lt;/sup&gt; string (PostgreSQL) in 1&lt;sup&gt;st&lt;/sup&gt; string (DB2) / 2&lt;sup&gt;nd&lt;/sup&gt; string (PostgreSQL)</td>
</tr>
<tr>
<td>RAND</td>
<td>Floating point values</td>
<td>Usage: RAND()</td>
<td>Example: SELECT RAND() FROM SYSIBM.SYSDUMMY1;</td>
<td>Returns a pseudo-random floating-point value in the range of zero to one inclusive.</td>
</tr>
<tr>
<td>ROUND</td>
<td>Integer</td>
<td>Usage: ROUND(&lt;FIELD&gt;, &lt;precision&gt;)</td>
<td>Example: SELECT ROUND(216.89, 1) FROM SYSIBM.SYSDUMMY1;</td>
<td>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.</td>
</tr>
<tr>
<td>RTRIM</td>
<td>String</td>
<td>Usage: RTRIM (&lt;TEXT_FIELD&gt;)</td>
<td>Example: SELECT RTRIM(' ABC'), LENGTH(RTRIM(' ABC')), RTRIM(' ABC'), LENGTH(RTRIM(' ABC')) FROM SYSIBM.SYSDUMMY1;</td>
<td>Removes trailing blanks, but not leading blanks, from the argument.</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>Integer</td>
<td>Converts either a number or a valid character value into a smallint value.</td>
<td>Example: SELECT SMALLINT(219.89) FROM SYSIBM.SYSDUMMY1;</td>
<td></td>
</tr>
<tr>
<td>SUBSTR</td>
<td>String</td>
<td>Usage: SUBSTR (&lt;TEXT_FIELD&gt;, &lt;int_position&gt;)</td>
<td>Example: SELECT SUBSTR('This is a substring test', 9) FROM SYSIBM.SYSDUMMY1;</td>
<td>Returns part of a string. If the length is not provided, the output is from the start value to the end of the string.</td>
</tr>
</tbody>
</table>
### TIMESTAMP

**Usage**: `TIMESTAMP (<FIELD>)`  

**Example**:

```sql
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'),
FROM SYSIBM.SYSDUMMY1;
```

**Example**: (to get the default timestamp)

```sql
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**:

```sql
SELECT TO_TIMESTAMP('2006-01-31-22.44.55.000000', 'YYYY-MM-DD-HH.MI.MS.SS'),
      TO_TIMESTAMP('2006-01-31-22.44.55.000', 'YYYY-MM-DD-HH.MI.MS.SS'),
      TO_TIMESTAMP('2006-01-31-22.44.55', 'YYYY-MM-DD-HH.MI.SS'),
      TO_TIMESTAMP('20060131224455', 'YYYYMMDDHHMMSS'),
FROM SYSIBM.SYSDUMMY1;
```

**Example**: (to get the default timestamp)

```sql
SELECT CURRENT_TIMESTAMP;
```

### UPPER

**Usage**: `UPPER (<TEXT_FIELD>)`  

**Example**:

```sql
SELECT UCASE('upper case'), UPPER('upper case') FROM SYSIBM.SYSDUMMY1;
```

### VALUE

**Usage**: Same as `COALESCE`  

**Usage**: Same as `COALESCE`  

**Example**:

```sql
SELECT UPPER('upper case');
```

**Example**:

```sql
SELECT UCASE('upper case'), UPPER('upper case') FROM SYSIBM.SYSDUMMY1;
```

### YEAR

**Usage**: `YEAR (<DATE_FIELD>)`  

**Usage**: `DATE_PART ('YEAR',<DATE_FIELD>);`  

**Example**:

```sql
SELECT YEAR (DATE('2006-09-21')) FROM SYSIBM.SYSDUMMY1;
```

**Usage**: `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

<table>
<thead>
<tr>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>```</td>
<td></td>
</tr>
</tbody>
</table>
| SELECT 
|   .... 
|   <column> 
|   .... 
| FROM 
|   <table(s)> 
| WHERE 
|   <condition(s)> 
|   ....  
| ORDER BY 
|   <column(s)> 
| ;                                       | Same as DB2   |
| ```                                                                              |

### 2.9.2 GROUP BY

<table>
<thead>
<tr>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>```</td>
<td></td>
</tr>
</tbody>
</table>
| SELECT 
|   Aggregate_fun(column1),  
|   Aggregate_fun(column2),  
|   <column> 
| FROM 
|   <table(s)> 
| WHERE <condition(s)> 
| GROUP BY <column>  
| ;                                       | Same as DB2   |
| ```                                                                              |

### 2.9.3 HAVING

<table>
<thead>
<tr>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
</table>
SELECT
    Aggregate_fun(column1),
    Aggregate_fun(column2),
    <column>
FROM <table(s)>
WHERE <condition(s)>
GROUP BY <column>
HAVING <condition>
;

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

<table>
<thead>
<tr>
<th>Equivalents / Declaration</th>
<th>IBM DB2</th>
<th>PostgreSQL</th>
<th>Same as DB2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>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;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.11.2 Left-outer Join

<table>
<thead>
<tr>
<th>Equivalents / Declaration</th>
<th>IBM DB2</th>
<th>PostgreSQL</th>
<th>Same as DB2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>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;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.11.3 Right-outer Join

<table>
<thead>
<tr>
<th>Equivalents / Declaration</th>
<th>IBM DB2</th>
<th>PostgreSQL</th>
<th>Same as DB2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>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;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.12 Sub-Query

<table>
<thead>
<tr>
<th>Equivalents / Declaration</th>
<th>IBM DB2</th>
<th>PostgreSQL</th>
<th>Same as DB2</th>
</tr>
</thead>
</table>
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..)

<table>
<thead>
<tr>
<th>Equivalents / Declaration</th>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECLARE result</td>
<td>RESULT_SET_LOCATOR VARYING;</td>
<td>DECLARE cursor REFCURSOR;</td>
</tr>
<tr>
<td>CALL procedure(&lt;params&gt;);</td>
<td>CALL procedure(&lt;params&gt;);</td>
<td>cursor := SELECT function_returning_cursor();</td>
</tr>
<tr>
<td>ASSOCIATE RESULT SET LOCATORS</td>
<td>ASSOCIATE RESULT SET LOCATORS</td>
<td>FETCH ALL IN cursor;</td>
</tr>
<tr>
<td>(result) WITH PROCEDURE procedure;</td>
<td>WITH PROCEDURE procedure;</td>
<td>or</td>
</tr>
<tr>
<td>ALLOCATE cursor CURSOR FOR RESULT SET result;</td>
<td>ALLOCATE cursor CURSOR FOR RESULT SET result;</td>
<td>FETCH cursor INTO &lt;var list&gt;;;</td>
</tr>
<tr>
<td>FETCH FROM cursor INTO &lt;var list&gt;;</td>
<td>FETCH FROM cursor INTO &lt;var list&gt;;</td>
<td></td>
</tr>
</tbody>
</table>

Example Usage
| 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 refcursore; |
| LANGUAGE plpgsql; |
| AS |
| $$
| DECLARE ref refcursor;
| BEGIN
| OPEN ref FOR SELECT 'JOHN'
| AS name;
| RETURN ref;
| END;
| $$
| ; |

| CREATE OR REPLACE FUNCTION func_fetch() |
| RETURNS refcursore; |
| LANGUAGE plpgsql; |
| AS |
| $$
| BEGIN
| DECLARE rsCur REFCURSOR;
| rsCur := SELECT func_select
| (); |
| FETCH cursor INTO myname;
| ... |
| CLOSE rsCur; |
| END;
| $$
| ; |
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 ('funcursor');
    FETCH ALL IN funcursor;
COMMIT;
```
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
;
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

<table>
<thead>
<tr>
<th>Equivalents / Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IBM DB2</strong></td>
</tr>
</tbody>
</table>
### 2.14.2 UNION ALL

<table>
<thead>
<tr>
<th>Equivalents / Declaration</th>
<th>IBM DB2</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT emp_id, pay_amt FROM emp_payments UNION ALL SELECT emp_id, pay_amt FROM emp_absent_payments</td>
<td>Same as DB2</td>
<td>(duplicate rows also will be fetched)</td>
</tr>
</tbody>
</table>
2.15 Dynamic SQL

```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

```sql
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

2.18.1 Casting double to integer syntax

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

2.18.2 Casting double to integer (Round)

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

2.18.3 Casting 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 SYISBM.SYSDUMMY1

There is no "SYISBM.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 DEFAULT <<DEFAULT_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.