[http://kabilsapworld.blogspot.com/search/label/HANA%20SQL#archive-page-4](http://kabilsapworld.blogspot.com/search/label/HANA%20SQL" \l "archive-page-4)

<http://www.hanaexam.com/p/sap-hana-tutorials.html>

# **Oracle Datatype to HANA Datatype Mapping**

|  |  |
| --- | --- |
| **Oracle data type** | **SAP HANA data type** |
| INTEGER | DECIMAL |
| NUMBER | DECIMAL |
| NUMBER(19)-NUMBER(38) | DECIMAL |
| NUMBER(10)-NUMBER(18) | BIGINT |
| NUMBER(5)-NUMBER(9) | INTEGER |
| NUMBER(2)-NUMBER(4) | SMALLINT |
| NUMBER(1) | TINYINT |
| NUMBER(p,s) | DOUBLE (if s > p),                                        DECIMAL (if 0 < s <= p),                             SMALLINT (if s < 0 and p-s <= 4),               INTEGER (if s < 0 and 4 < p-s <= 9),           BIGINT (if s < 0 and 9 < p-s <= 18),           DECIMAL (if s< 0 and p-s > 18) |
|  | DECIMAL (if 0 < s <= P) |
|  | SMALLINT (if s < 0 and p-s <= 4) |
|  | INTEGER (if s < 0 and 4 < p-s <= 9) |
|  | BIGINT (if s < 0 and 9 < p-s <= 18) |
|  | DECIMAL (if s< 0 and p-s > 18) |
|  |  |
| FLOAT | DOUBLE |
| FLOAT(1)-FLOAT(24) | REAL |
| FLOAT(25)-FLOAT(126) | DOUBLE |
| BINARY\_FLOAT | REAL |
| BINARY\_DOUBLE | DOUBLE |
| DATE | TIMESTAMP |
| TIMESTAMP(n) | TIMESTAMP |
| CHAR | VARCHAR |
| NCHAR | NVARCHAR |
| VARCHAR2 | VARCHAR ( MAX LENGTH : 5000) |
| NVARCHAR2 | NVARCHAR |
| BLOB | BLOB |
| BFILE | BLOB |
| RAW | VARBINARY |
| LONG | CLOB |
| LONG RAW | BLOB |
| CLOB | CLOB/NCLOB |
| NCLOB | NCLOB |
| INTERVAL | VARCHAR |
| TIMESTAMP WITH TIME ZONE | VARCHAR |
| TIMESTAMP WITH LOCAL TIME ZONE | VARCHAR |
| ROWID | Not Supported.. Its Pseudocolumn $row\_id$ |
| UROWID | Not Supported |
| ANYDATA | Not Supported |
| VARRAY | Not Supported |
| NESTEDTAB | Not Supported |
| OBJECT | Not Supported |
| REF | Not Supported |
| XMLANY | CLOB |

# **SCHEMA**

# **What is Schema in SAP HANA?**

**A database schema is a way to logically group objects such as tables, views, stored procedures etc. Think of a schema as a container of objects.**

**Types of Schemas**

**There are 3 types of schemas.**

**1. User Defined Schema**

**2. System Defined Schema**

**3. SLT Derived Schema**

**User Defined Schema:**

**These are created by user (DBA or System Administrator)**

**Example:**

**Syntax: Create Schema <Schema\_name>;**

**Code:** **create** **schema** "KABIL\_PRACTICE";

**Note :**

**The Created Schema is available in under the “Catalog”**

**SLT Derived Schema:**

* **Schema which are created upon Data Provisioning by SLT process.**
* **When SLT is configured, it creates schema in HANA system. All the tables replicated into HANA system are contained in this schema**

**System Defined Schema:**

* **These schemas are delivered with the SAP HANA database and contains HANA system information. There are system schemas like \_SYS\_BIC, \_SYS\_BI, \_SYS\_REPO, \_SYS\_STATISTICS etc.**

**System Generated Schemas**

**SAP HANA System Schemas**

**\_SYS\_BIC:**

* **This schema contains all the columns views of activated objects. When the user activates the Attribute View/Analytic View/Calculation View/Analytic Privilege /Procedure, the respective run-time objects are created under \_SYS\_BIC/ Column Views.**

**\_SYS\_REPO:**

* **Whatever the objects are there in the system is available in repository. This schema contains the list of Activated objects, Inactive Objects, Package details and Runtime Objects information etc.**
* **Also \_SYS\_REPO user must have SELECT privilege with grant option on the data schama.**

**\_SYS\_BI:**

* **This schema stores all the metadata of created column Views. It contains the tables for created Variables, Time Data (Fiscal, Gregorian), Schema Mapping and Content Mapping tables.**

**\_SYS\_STATISTICS:**

* **This schema contains all the system configurations and parameters.**

**\_SYS\_XS:** **This schema is used for SAP HANA Extended Application Services.**

# **Procedures:**

# **Procedure in SAP HANA**

* **A procedure is a unit/module that performs a particular task. Procedures are reusable processing blocks, and describe a sequence of data transformations Procedures can have multiple inputs and output parameters (scalar or table types) DROP and CREATE statements are used to modify the definition of a procedure A procedure can be created as read only (without side-effects) or read-write (with side-effects)**
* **Stored Procedures can return data in the form of output parameters (integer or character) or a cursor variable. It can also result in a set of Select statements, which are used by other Stored Procedures.**
* **Stored Procedures are also used for performance optimization as it contains series of SQL statements and results from one set of statement determines next set of statements to be executed. Stored procedures prevent users to see the complexity and details of tables in a database. As Stored procedures contain certain business logic, so users need to execute or call the procedure name.**

**An SQL Procedure can be created at –**

**1.      At Schema Level(Catalog Node)**

**2.      At Package Level(Content Node)**

**Example:**

**Here I try to create a simple procedure to get the data by entering the order id.**

***create******procedure****"KABIL"."PROCEDURE\_SUPERSTORE\_\_ORDER\_ID"(****in****O\_id****integer****)****asbeginSELECT****\*****FROM****"KABIL"."SUPERSTORE\_SALES\_HISTORY"****where****"Order ID"= :O\_id;****end****;*

**-- To drop procedure**

***drop******procedure****"KABIL"."PROCEDURE\_SUPERSTORE\_\_ORDER\_ID";*

**-- To call procedure**

***call****"KABIL"."PROCEDURE\_SUPERSTORE\_\_ORDER\_ID"(613);*

**Primary Key**

# **Primary Key in SAP HANA**

* F**or some applications, it is useful to uniquely identify the rows in a table by one or more columns so that you can process them in a specified sequence, for example. You can do this by assigning a primary key. The column names that are to create the table key are represented by the keywords PRIMARY KEY. The input values of the key columns defined in this way must not be a NULL value.**
* **You can use the primary key to insert rows in a table in the same way as you insert rows in a base table when no primary key is defined. However, the system outputs an error message if you attempt to insert an existing value into the primary key column a second time since the uniqueness of the column is ensured by defining the primary key.**
* **A primary key can consist of multiple columns. However, it is unusual for a key to be constructed from more than five columns, since this makes it difficult for users to enter unique values. The arrangement of the columns behind the keywords PRIMARY KEY defines the key sequence.**
* **You can specify a primary key when you define the table or add it to an existing table at a later stage.**

**Examples:**

**Code:**

**create column table "KABIL\_PRACTICE"."PRODUCT\_PRIMARY\_KEY"**

**(**

**"Prod\_Id" integer primary key,**

**"Prod\_Name" varchar(25),**

**"Stock\_Qty" integer,**

**"Each\_Price" integer**

**);**

**(Or)**

**create column table "KABIL\_PRACTICE"."PRODUCT\_PRIMARY\_KEY"**

**(**

**"Prod\_Id" integer,**

**"Prod\_Name" varchar(25),**

**"Stock\_Qty" integer,**

**"Each\_Price" integer,**

**primary key ("Prod\_Id")**

**);**

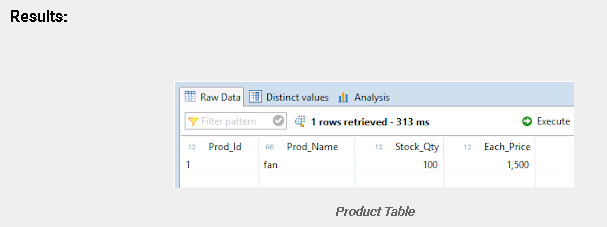
**Both statements allow to uniquely identify the rows in the PRODUCT\_PRIMARY\_KEY table on the basis of the product id “Prod\_Id”.**

**And insert some records into the table.**

**Code:**

**insert into "KABIL\_PRACTICE"."PRODUCT\_PRIMARY\_KEY" values**

**(1,'fan',100,1500);**

**

When I try to insert another record with the same “Prod\_Id” It through an error like this.



**And also it won’t allow any Null values.**

# **Foreign Key**

# **Foreign Key in SAP HANA**

* **A foreign key is a column (or columns) that references a column (most often the primary key) of another table. The purpose of the foreign key is to ensure referential integrity of the data. In other words, only values that are supposed to appear in the database are permitted.**

**Why should I use foreign keys?**

* **The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables. The FOREIGN KEY constraint also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.**

**Can a foreign key have a null value?**

* **Any number of rows in the child table can reference the same parent key value, so this model establishes a one-to-many relationship between the parent and foreign keys. However, each row in the child table must have a reference to a parent key value; the absence of a value (a null) in the foreign key is not allowed.**

**What is the difference between a primary key and a foreign key?**

* **A primary key is a column or a set of columns that uniquely identify a row in a table. ... Usually, a foreign key is in a table that is different from the table whose primary key it is required to match. A table can have multiple foreign keys. The primary key cannot accept null values.**
* **A foreign key must reference either the primary key or a unique key of the parent table. If the primary key has multiple columns, the foreign key must have the same number and order of columns. Therefore the foreign key references a unique row in the parent table; there can be no duplicates.**

**Examples:**

**Here I already create a "KABIL\_PRACTICE"."PRODUCT\_PRIMARY\_KEY" Table.**

**Now I’m creating the another table with foreign Key.**

**Code:**

**CREATE column TABLE "KABIL\_PRACTICE"."SALES\_FOREIGN"**

**(**

**"Order\_code" integer,**

**"cust\_name" varchar(25),**

**"Prod\_Id"  integer  NOT NULL,**

**"Prod\_Name" varchar(25),**

**"Quantity" integer Not Null,**

**"Sales\_Price" integer,**

**FOREIGN KEY("Prod\_Id")**

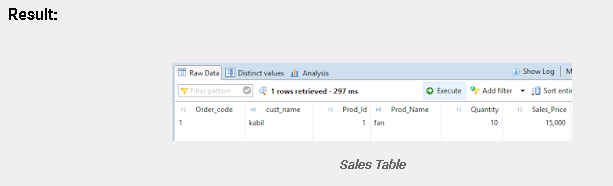
**REFERENCES "KABIL\_PRACTICE"."PRODUCT\_PRIMARY\_KEY" ("Prod\_Id")**

**) ;**

**And insert some records into the table.**

**Code:**

**insert into "KABIL\_PRACTICE"."SALES\_FOREIGN" values (1,'kabil',1,'fan',10,15000);**

**

**And now I try to insert another record into the table.**

**Code:**

**insert into "KABIL\_PRACTICE"."SALES\_FOREIGN" values (2,'arasan',2,'chair',5,7500);**

**If I try to insert another record which is not present in the product table it throws an error like this,**

**

# **Triggers**

# **SAP HANA : Triggers After**

* **This statement is used to triggers which is a set of statements that are executed when a given operation (INSERT/UPDATE/DELETE) takes place on a given subject table or subject view.**
* **A trigger is also a stored procedure that automatically executes when an event happens on a given table or view.**
* **The database users only having the TRIGGER privilege for the given <subject\_table\_name> are allowed to create a trigger for that table or view.**
* **The CREATE TRIGGER command defines a set of statements that are executed when a given operation (INSERT/UPDATE/DELETE) takes place on a given subject table or subject view.**

**Creates a trigger on a table or view.**

**Syntax**

**CREATE TRIGGER <trigger\_name> <trigger\_action\_time> <trigger\_event\_list>**

**ON <subject\_table\_name>**

**[REFERENCING <transition\_list>]**

**[<for\_each\_row>]**

**BEGIN**

**[<trigger\_decl\_list>]**

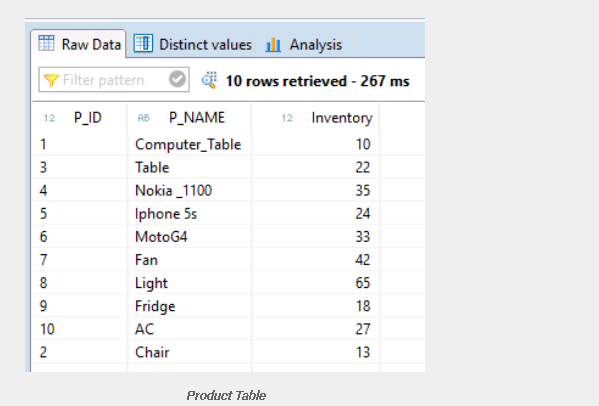
**[<proc\_handler\_list>]**

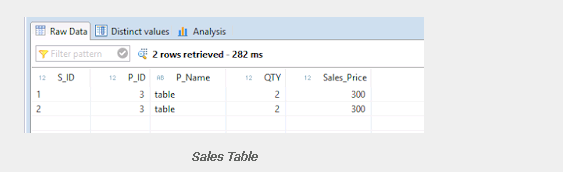
**<trigger\_stmt\_list>**

**END;**

**Ex:**

**Here I have two table named as Product table and Sales table:**

**

**

**Now, I’m Trying to create a trigger with the concept whenever I inserting a new record in a sales table the Product table should update is “Inventory” based on sales table  “quantity”**

**Code:**

**/\*---------------------------------------------------------\*/**

**create trigger "KABIL\_PRACTICE"."SALES\_TRIGGER"**

**after insert on "KABIL\_PRACTICE"."SALES" REFERENCING NEW ROW AS newrow for each row**

**begin**

**update "KABIL\_PRACTICE"."Inventory" set "Inventory" = "Inventory" - :newrow.QTY**

**where "P\_ID" = :newrow.P\_ID ;**

**end;**

**/\*-----------------------------------------------------------\*/**

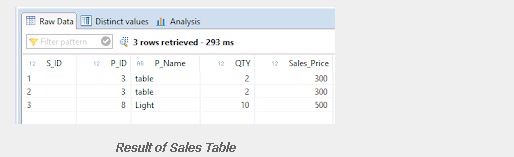
*][*

**Now Insert a record in sales table:**

**Code:**

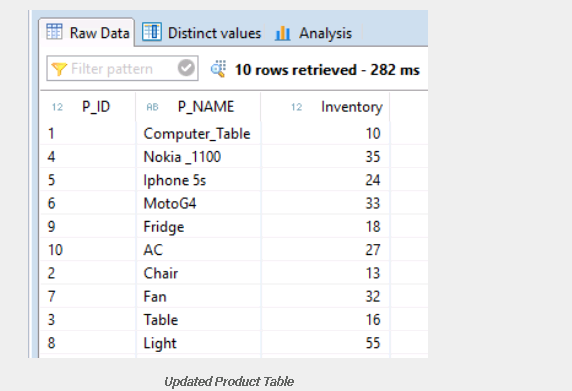
**insert into "KABIL\_PRACTICE"."SALES" values(3,8,'Light',10,500);**

**Now the Sales table should be like this:**

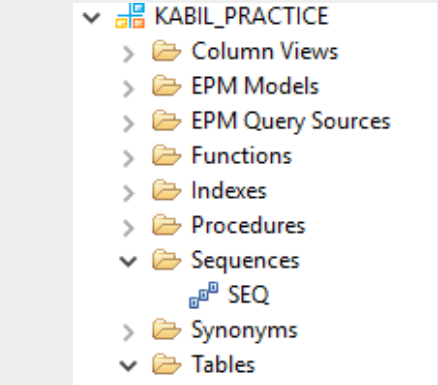
**

**Results:**

**When I inserting a new record in my “Sales Table” the trigger will execute automatically. From the above insert query will insert light product with the quantity of 10. so, the “Product table” inventory value of that light product will be reduced as per the quantity ordered in the sales table.**

**

# **Sequence**



# **Sequence in SAP HANA**

* **A sequence is a database object that generates an automatically incremented list of numeric values according to the rules defined in the sequence specification.**
* **The specification of numeric values is generated in an ascending or descending order at defined increment interval, and the numbers generated by a sequence can be used by applications, for example, to identify the rows and columns of a table.**
* **For example, to insert employee number automatically in column (EMPLOYEE\_NO) of Table, when a new record is inserted in the table, then we use sequence.**
* **Sequences are not associated with table; they are used by applications, which can use CURRVAL in a SQL statement to get the current value generated by a sequence and NEXTVAL to generate the next value in the defined sequence.**

**There are two values in sequence –**

**CURRVAL – Provide Current value of Sequence.**

**NEXTVAL – Provide Next value of sequence.**

**Syntax**

*CREATE SEQUENCE <sequence\_name> [<common\_sequence\_parameter\_list>] [RESET BY <subquery>]*

**Syntax Elements**

**<sequence\_name> ::= [<schema\_name>.]<identifier>**

**<schema\_name> ::= <identifier>**

**<common\_sequence\_parameter\_list> ::= <common\_sequence\_parameter>, ...**

**<common\_sequence\_parameter> ::= <sequence\_parameter\_start\_with>**

**| <basic\_sequence\_parameter>**

**<basic\_sequence\_parameter> ::= INCREMENT BY n**

**| MAXVALUE n**

**| NO MAXVALUE**

**| MINVALUE n**

**| NO MINVALUE**

**| CYCLE**

**| NO CYCLE**

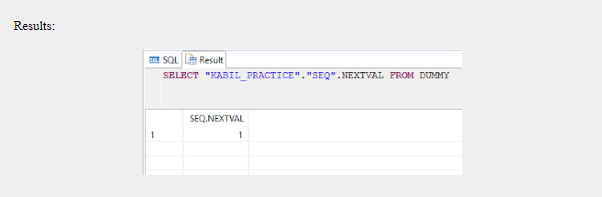
**<sequence\_parameter\_start\_with> ::= START WITH n**

**INCREMENT BY:  
Defines the amount the next sequence value is incremented from the last value assigned. The default is 1. Specify a negative value to generate a descending sequence. An error is returned if the INCREMENT BY value is 0  
  
START WITH:  
Defines the starting sequence value. If you do not specify a value for the START WITH clause, MINVALUE is used for ascending sequences and MAXVALUE is used for descending sequences.  
  
MAXVALUE:  
Defines the largest value generated by the sequence and must be between 0 and 4611686018427387903.  
  
NO MAXVALUE:  
When MAXVALUE is not specified, the maximum value for an ascending sequence is 4611686018427387903 and the maximum value for a descending sequence is -1.  
  
MINVALUE:  
The minimum value of a sequence can be specified after MINVALUE and is between 0 and 4611686018427387903.  
  
NO MINVALUE:  
When MINVALUE is not specified, the minimum value for an ascending sequence is 1 and the minimum value for a descending is -4611686018427387903.  
  
CYCLE:  
The sequence number will be reused after it reaches its maximum or minimum value.  
  
NO CYCLE:  
Default option. The sequence number will not be reused after it reaches its maximum or minimum value.  
  
RESET BY:  
During the restart of the database, the database automatically executes the RESET BY statement and the sequence value is restarted with the specified value from the statement after RESET BY.  
  
If RESET BY is not specified, the sequence value is stored persistently in the database. During the restart of the database, the next value of the sequence is generated from the saved sequence value.  
  
  
You can use database sequences to perform the following operations:  
Generate unique, primary key values, for example, to identify the rows and columns of a table  
  
Coordinate keys across multiple rows or tables  
  
Example for creating a sequence:  
Sequence seq is created, then CURRVAL and NEXTVAL are used to get the values from the sequence:**

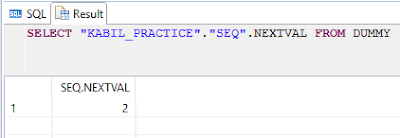
***create******sequence****"KABIL\_PRACTICE"."SEQ" start****with****1;*

**NEXTVAL returns 1:**

**SELECT "KABIL\_PRACTICE"."SEQ".NEXTVAL FROM DUMMY;**



**Note: Whenever I execute the above query it returns NEXT value. Such as when I execute it again it through the output like,**

[](https://4.bp.blogspot.com/-2dsKDDIWz38/WXcjAtsleRI/AAAAAAAABXg/3DBFHtC0WzYPV5t_9HdlDsKgqB-E1uPswCLcBGAs/s1600/execeute%2Bagain.PNG)

## **To Drop a SEQUENCE:**

***drop******sequence****"KABIL\_PRACTICE"."SEQ";*

## **Example For INCREMENT BY:**

## **This specifies the value to be incremented from the last value assigned for each time when new sequence value generated. The default is 1.**

## **CODE:**

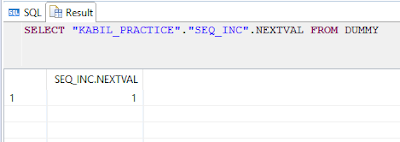
***create sequence "KABIL\_PRACTICE"."SEQ\_INC" start with 1 increment by 2;***

## **NOTE: Here I incremented it by 2. The default value is 1.**

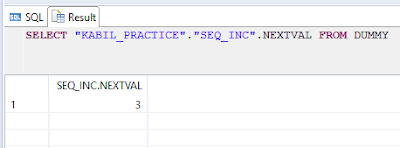
**Then Execute this code:**

***SELECT "KABIL\_PRACTICE"."SEQ\_INC".NEXTVAL FROM DUMMY;***

**Result:**

[](https://3.bp.blogspot.com/-9DLJ6MXsg-Q/WXcjrwgdo-I/AAAAAAAABXo/lFNeIZbUYb4SyjhmlMo0-mUEu9u9w3XywCLcBGAs/s1600/Initil%2Bvalue%2Bin%2Bincrement%2Bby%2B2.PNG)

**Again execute the above code:**

[](https://4.bp.blogspot.com/-PsJXZsDC7AY/WXci67821tI/AAAAAAAABXk/oxuIAtKrBpInBvHzXUcppyshU_Je0H0ngCEwYBhgL/s1600/again%2Bexecute%2Bincrement%2Bby%2B2.PNG)

**Here, you may note the value changed from “1” to “3” i.e. Incremented by 2.**

**Example for MAXVAL:**

**CODE:**

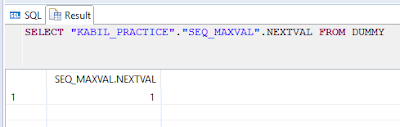
***create******sequence****"KABIL\_PRACTICE"."SEQ\_MAXVAL" start****with****1 increment****by****2 MAXVALUE 5;*

**NOTE: Here I incremented it by “2”. Default value is “1” and I mentioned that the MAXIMUM Value is “5”**

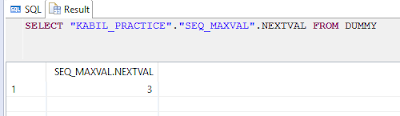
**Then Execute this code:**

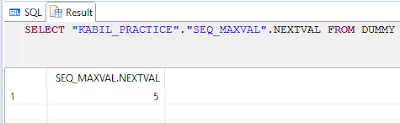
***SELECT****"KABIL\_PRACTICE"."SEQ\_MAXVAL".NEXTVAL****FROM****DUMMY;*

### **Result:**

[](https://3.bp.blogspot.com/-uz7zL0ZYJ8M/WXckKgdWTbI/AAAAAAAABXs/0DfK7XGHoe0JzpP7Wpu-oHupNKOOAqY6QCLcBGAs/s1600/Default%2Bvalue%2B1.PNG)

**Again execute the above code till it reaches the maximum value.**

[](https://1.bp.blogspot.com/-xFrd-U8IQLA/WXckUt0hnNI/AAAAAAAABX0/KTnCPWWYz9s9cQYtAwgO6Y9Cfc7CuMc_wCLcBGAs/s1600/MAXVAL%2B3.PNG)

[](https://4.bp.blogspot.com/-Jdfk0f0k-4o/WXckUmAAG6I/AAAAAAAABX4/Qd7SX9A_qLYCIS0X6UZ9PxYZotCBGdJ3wCEwYBhgL/s1600/max%2Bval%2B5.PNG)

Now it reached it maximum value. If you execute the above code it through an error like,

[https://4.bp.blogspot.com/-pBBC5Fm3Dh8/WXckUpoSOwI/AAAAAAAABXw/n_7kIcN0RxEsurZIvucioWvOih3jNjRvACEwYBhgL/s400/Error.PNG](https://4.bp.blogspot.com/-pBBC5Fm3Dh8/WXckUpoSOwI/AAAAAAAABXw/n_7kIcN0RxEsurZIvucioWvOih3jNjRvACEwYBhgL/s1600/Error.PNG)

**i.e. Sequence is exhausted.**

**NOTE: NO MAXVALUE**

**When MAXVALUE is not specified, the maximum value for an ascending sequence is 4611686018427387903 and the maximum value for a descending sequence is -1.**

**Example for MINVALUE:**

**CODE:**

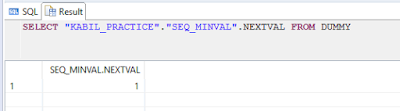
***create******sequence****"KABIL\_PRACTICE"."SEQ\_MINVAL" start****with****1 increment****by****-3 MAXVALUE 5 MINVALUE -5;*

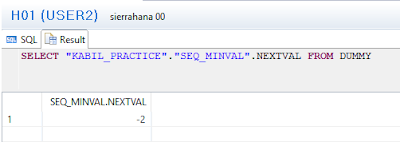
**NOTE: Here I incremented it by “-3”. Default value is “1” and I mentioned that the MAXIMUM Value is “5” and MINVALUE “-5”. Whenever we use MINVALUE in SEQUENCE we have to mention the MAXVALUE. Otherwise the sequence wont be create.**

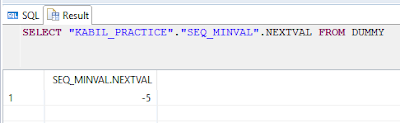
**Then Execute this code until it reaches it MINVALUE:**

***SELECT****"KABIL\_PRACTICE"."SEQ\_MAXVAL".MINVAL****FROM****DUMMY;*

**Result:**

[](https://2.bp.blogspot.com/-qlSwdySM9I4/WXclAmOZc3I/AAAAAAAABYA/SZvp45xsf1AP_MmQUE41yimRvlYG7CCRACLcBGAs/s1600/default%2Bval.PNG)

[](https://3.bp.blogspot.com/-y3yuPKsqEFY/WXclAtZxcDI/AAAAAAAABYE/8CH-wxQ3t7UQ_B6NzgDr_pZf1fxQGd6eACLcBGAs/s1600/-2.PNG)

[](https://4.bp.blogspot.com/-hFGv_PnQ5ck/WXclAsdhApI/AAAAAAAABX8/jgAhVM5A_u4129c-UcmJGKhRCQQ4aiMyACLcBGAs/s1600/-5.PNG)

**Now it reaches it “MINIMUMVALUE”. if you execute the above code it through’s the error like,**

[https://1.bp.blogspot.com/-dGfctUI8LoQ/WXclA2XrLjI/AAAAAAAABYI/ZocRVAqJrGMxdqQlFVryD7GZtB5Yz-19QCEwYBhgL/s400/error.PNG](https://1.bp.blogspot.com/-dGfctUI8LoQ/WXclA2XrLjI/AAAAAAAABYI/ZocRVAqJrGMxdqQlFVryD7GZtB5Yz-19QCEwYBhgL/s1600/error.PNG)

**Note:**

**NO MINVALUE:**

**When the NO MINVALUE directive is used, the minimum value for an ascending sequence is 1 and the minimum value for a descending sequence is -4611686018427387903.**

**Example for CYCLE:**

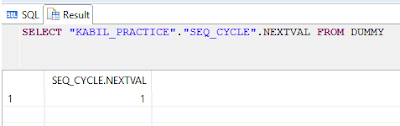
**CODE:**

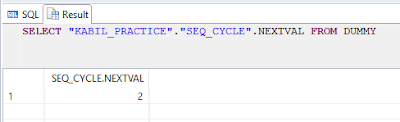
***create******sequence****"KABIL\_PRACTICE"."SEQ\_CYCLE" start****with****1 increment****by****1 MAXVALUE 3 cycle;*

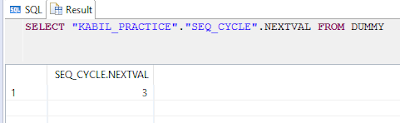
**Then Execute this code until it reaches it MAXVALUE:**

***SELECT****"KABIL\_PRACTICE"."SEQ\_CYCLE".NEXTVAL****FROM****DUMMY;*

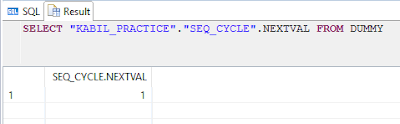
**RESULT:**

[](https://4.bp.blogspot.com/-A1TjCCbmNuo/WXclvrSfc-I/AAAAAAAABYY/-bZAZbZlpZw_2o8w1FXP3Mli9cNX8cPFgCLcBGAs/s1600/default%2Bval.PNG)

[](https://2.bp.blogspot.com/-T213rRl5Gko/WXclvl7DdjI/AAAAAAAABYQ/mQ7H3_Gn-YoJKyl_pkrtc8r6UV3eu4BeACLcBGAs/s1600/2.PNG)

[](https://1.bp.blogspot.com/-U0a0AUK3vX4/WXclvs8_HYI/AAAAAAAABYU/vG7Wole0x906llJCus6s_kJwNoSLp8tpQCLcBGAs/s1600/3.PNG)

**Now it reaches its end value and again it starts from "1" i.e., Cycle.**

[](https://1.bp.blogspot.com/-A-iKvHz09VA/WXclv8znleI/AAAAAAAABYc/5ZlbQoY-U4M95bm-L67JJ7VninIedV4ywCLcBGAs/s1600/restart%2Bits%2Bcycle.PNG)

**NOTE:**

**NO CYCLE:**

* **Default option.**
* **When the NO CYCLE directive is used the sequence number will not be restarted after it reaches its maximum or minimum value.**
* **For example, to insert employee number automatically in column (EMPLOYEE\_NO) of Table, when a new record is inserted in the table, then we use sequence.**

**Example for Inserting a created sequence in an Existing table:**

**CODE:**

**1.  Create a table.**

***create******column******table****"KABIL\_PRACTICE"."EMP\_NAME\_SEQ"("EMP\_NAME"****VARCHAR****(10));*

**2. Insert a records into the table.**  
***INSERT******INTO****"KABIL\_PRACTICE"."EMP\_NAME\_SEQ"****VALUES****('KABIL');*

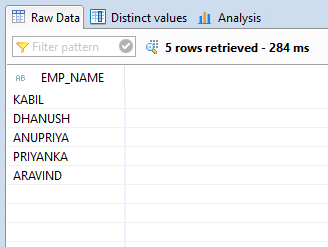
***INSERT******INTO****"KABIL\_PRACTICE"."EMP\_NAME\_SEQ"****VALUES****('DHANUSH')*

***INSERT******INTO****"KABIL\_PRACTICE"."EMP\_NAME\_SEQ"****VALUES****('ANUPRIYA');*

***INSERT******INTO****"KABIL\_PRACTICE"."EMP\_NAME\_SEQ"****VALUES****('PRIYANKA');*

***INSERT******INTO****"KABIL\_PRACTICE"."EMP\_NAME\_SEQ"****VALUES****('ARAVIND');*

**Results:**

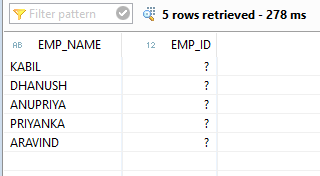
[](https://1.bp.blogspot.com/-TxKXmUmbEp8/WXcnDXiiaLI/AAAAAAAABYo/h0MwIC3uW3UwdnCVVzU98L3NuI-Ih7lkACLcBGAs/s1600/inserted%2Brecords.PNG)

**1. Create a sequence.**

**Code:**

***create******sequence****"KABIL\_PRACTICE"."INSERT\_SEQ" start****with****101;*

**2. Alter the table by adding One additional column named as “EMP\_ID”.**

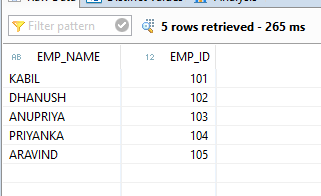
**Result:**  
[](https://1.bp.blogspot.com/-pewRkE1Fnko/WXcnuJO1OfI/AAAAAAAABYw/Baw9lb_OQvQFSJ-G_t7DiRCvrQSb6WWkACEwYBhgL/s1600/add%2Bcolumn%2BEMP_ID.PNG)

**3. Insert a created sequence into the table:**

**Code:**

**update "KABIL\_PRACTICE"."EMP\_NAME\_SEQ" set "EMP\_ID" = "KABIL\_PRACTICE"."INSERT\_SEQ".NEXTVAL;**

**Result:**

**[](https://1.bp.blogspot.com/-wo4UCWxrpeg/WXcn9fPVU8I/AAAAAAAABY0/VLVhmgsgM28HUt51UOMqhsVDaKchyaebwCLcBGAs/s1600/inserted%2Bsequence.PNG)**

**IMPORTANT POINTS:  
Description**

**A sequence generates unique integers for use by multiple users. Use CURRVAL to get the current value of the sequence and NEXTVAL to get the next value of the sequence. CURRVAL is only valid after calling NEXTVAL in a session.**

**You can use CURRVAL and NEXTVAL only in:**

* **The select list of a SELECT statement which is not contained in a subquery, or view**

**The select list of a subquery in an INSERT statement**

* **The VALUES clause of an INSERT statement**
* **The SET clause of an UPDATE statement**

**You are not allowed to use CURRVAL and NEXTVAL in:**

**The WHERE clause of a SELECT statement.**

* **A subquery in a DELETE, SELECT, UPDATE, REPLACE or UPSERT statement.**
* **A SELECT statement in a CREATE VIEW statement.**
* **A SELECT statement with the DISTINCT operator.**
* **A SELECT statement with a GROUP BY clause.**
* **A SELECT statement with the UNION, INTERSECT, or MINUS set operator.**
* **The DEFAULT value of a column in a CREATE TABLE or ALTER TABLE statement.**

# **UPSERT**

**UPSERT: The UPSERT is used to either updates rows in a table or inserts new rows if the record doesn’t exist with the specified condition.**

**Syntax: UPSERT(or REPLACE) <schema\_name>.<table\_name> VALUES (values) WHERE <condition> ;  
Points to remember:**

1. **Whenever this command is used without a subquery it functions in a similar way to the UPDATE statement. The difference with this command is that when the WHERE clause condition is false it will insert a new record into the table.**
2. **When this command is used with a table that has a PRIMARY KEY, the primary key column must be included in the column list. Columns defined with NOT NULL and without a default specification also have to be included in the column list. Columns that are not specified will be filled with a default value or NULL.**
3. **The UPSERT (or REPLACE) statement with a subquery functions like the INSERT statement. The exception with this command is that, if an existing row in the table has the same primary key value as a new row, the row will be updated with the returned record from the subquery. If the table does not have a primary key the command functions in an equivalent way to an INSERT statement, as an index cannot be used to check for row duplication.**

**Examples:**

1. **Insert a record into the table using UPSERT.**

**Ans: UPSERT "KABIL\_PRACTICE"."ORDERS"  
                    VALUES (1234,70)   — Columns are Order\_Number and Quantity**

1. **Insert the record if where clause if false or update the row if where clause is true.**

**Ans: UPSERT "KABIL\_PRACTICE"."ORDERS"  
                    VALUES (1235,120) WHERE ORD\_NO = 1235**

1. **Update the existing record with primary key constraint.**

**Ans: UPSERT "KABIL\_PRACTICE"."ORDERS"  
                    VALUES (1234,170) WITH PRIMARY KEY**

1. **Update or insert the records using sub query.**

**Ans: UPSERT "KABIL\_PRACTICE"."ORDERS"  
                    SELECT 1236,40  
                    FROM DUMMY**

**Examples:**

**Create Order table:**

**create column table "KABIL\_PRACTICE"."ORDERS"**

**(**

**"ID" integer,**

**"QTY" integer**

**)**

**;**

**Insert records in the table:**

**insert into "KABIL\_PRACTICE"."ORDERS" Values (1,100);**

**insert into "KABIL\_PRACTICE"."ORDERS" Values (2,150);**

**insert into "KABIL\_PRACTICE"."ORDERS" Values (3,160);**

**Result:**

**Now, Try this code 1:**

**upsert "KABIL\_PRACTICE"."ORDERS" values (4,180) where "ID" = 2;**

**Result:**

**ORDERS:**

|  |
| --- |
| *[https://1.bp.blogspot.com/-Zt8onqbyL_A/WW74w-GpSwI/AAAAAAAABTU/ZGCPYKcF7d8iBWI_sFbbWImM9c6mjnWIQCEwYBhgL/s320/update.PNG](https://1.bp.blogspot.com/-Zt8onqbyL_A/WW74w-GpSwI/AAAAAAAABTU/ZGCPYKcF7d8iBWI_sFbbWImM9c6mjnWIQCEwYBhgL/s1600/update.PNG)* |
|  |
| ***Updated Table*** |

**Note: It's just updated a record where “ID” = 2, because where the condition is True.**

**Code 2:**

**upsert "KABIL\_PRACTICE"."ORDERS" values (2,280) where "ID" = 2;**

**Results:**

|  |
| --- |
| *[https://3.bp.blogspot.com/-rZFcu68ns48/WW74wUV0kdI/AAAAAAAABTM/HL2wTPMYbQkcWfnMKs6S17iZpSrF2t6ywCEwYBhgL/s320/insert.PNG](https://3.bp.blogspot.com/-rZFcu68ns48/WW74wUV0kdI/AAAAAAAABTM/HL2wTPMYbQkcWfnMKs6S17iZpSrF2t6ywCEwYBhgL/s1600/insert.PNG)* |
| ***Inserted Table*** |

**Not just inserts a new record because where condition is false.**

**Code 3:**

**upsert "KABIL\_PRACTICE"."ORDERS" values (7,366);**

**Results:**

|  |
| --- |
| *[https://4.bp.blogspot.com/-ORbD0M5tkJ8/WW74wY2OxBI/AAAAAAAABTI/1PcLsQwpBH4bz0wb_ZY7nz83mOEHSudfQCEwYBhgL/s320/all%2Bvalues.PNG](https://4.bp.blogspot.com/-ORbD0M5tkJ8/WW74wY2OxBI/AAAAAAAABTI/1PcLsQwpBH4bz0wb_ZY7nz83mOEHSudfQCEwYBhgL/s1600/all%2Bvalues.PNG)* |
| ***All values changed*** |

**Note: Here all the values are updated, Because I didn’t mention a condition.**

# **Database Views or SQL Views in SAP HANA**

* **In SQL, a view is a virtual table based on the dynamic results returned in response to an SQL statement. Every time a user queries an SQL view, the database uses the view's SQL statement to recreate the data specified in the SQL view. The data displayed in an SQL view can be extracted from one or more database tables.**

* **An SQL view contains rows and columns, just like a real database table; the fields in an SQL view are fields from one or more real tables in the database. You can add SQL functions, for example, WHERE or JOIN statements, to a view and present the resulting data as if it were coming from one, single table.**

**Example: Simple View**

**Code:**

**create view "KABIL\_PRACTICE"."SUPERSTORE\_VIEW"**

**as**

**select**

**"Order\_ID",**

**"Customer\_Name",**

**"Sales"**

**from**

**"KABIL\_PRACTICE"."SUPERSTORE\_SALES";**

**(OR)**

**create view "KABIL\_PRACTICE"."SUPERSTORE\_VIEW" ("O\_ID","C\_NAME","SALES\_PRICE")**

**as**

**select**

**"Order\_ID",**

**"Customer\_Name",**

**"Sales"**

**from**

**"KABIL\_PRACTICE"."SUPERSTORE\_SALES";**

**Note:**

**Here I just change the column name of the view as (O\_ID","C\_NAME","SALES\_PRICE"). Both are Same.**

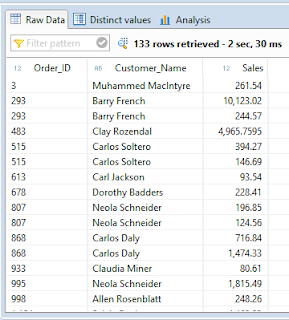
**To Call the View:**

**Right click the view from the view folder and click open Data Preview or Open Data Content.**

**To Drop the View:**

**Drop view "KABIL\_PRACTICE"."SUPERSTORE\_VIEW";**

**Result:**

[](https://3.bp.blogspot.com/-zTBs9Ike-f8/WZQPdbjI7sI/AAAAAAAABew/Q_tCeRAzoIATrvBB4my5AsX1qmozOnnGQCLcBGAs/s1600/Result.PNG)

**Example: View with Joins**

**create view "KABIL\_PRACTICE"."PAYMENT\_DETAILS" as**

**(**

**select**

**c."CUSTOMERNUMBER",**

**c."CUSTOMERNAME",**

**p."PAYMENTDATE",**

**p."AMOUNT"**

**from**

**"KABIL\_PRACTICE"."CUSTOMER\_SAP\_STUDENT" as c**

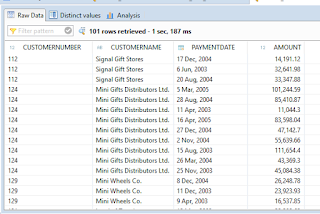
**inner join**

**"KABIL\_PRACTICE"."PAYMENTS\_SAP\_STUDENT" as p**

**on**

**c."CUSTOMERNUMBER" = p."CUSTOMERNUMBER");**

**RESULT:**

[](https://3.bp.blogspot.com/-Uz2EpLe66sE/WZQYljpHoyI/AAAAAAAABfA/by3sf5ldlroDKd-cJoHjfAqpDZlU8F6yQCLcBGAs/s1600/view%2Bjoins%2Bresult.PNG)

# **SAP HANA: Synonyms**

**Description:**

**The CREATE SYNONYM creates an alternate name for a table, view, procedure or sequence.**

* **You can use a synonym to re-point functions and stored procedures to differing tables, views or sequences without needing to re-write the function or stored procedure.**
* **The optional PUBLIC element allows for the creation of a public synonym. A public synonym can be accessed by any users, but the users that have proper privilege on its base object can access the base object.**

**Syntax:**

**CREATE [PUBLIC] SYNONYM <synonym\_name> FOR <object\_name>**

**EXAMPLE:**

**create** **synonym** "KABIL\_PRACTICE"."CUSTOMER" **for** "KABIL\_PRACTICE"."CUSTOMER\_SAP\_STUDENT";

**We can access the synonym from the Synonyms folder under the specified Schema.**

**To Create Public Synonym:**

**create** **Public** **synonym** "CUSTOMER\_KABIL" **for** "KABIL\_PRACTICE"."CUSTOMER\_SAP\_STUDENT";

**We can access the synonym from the Public Synonyms folder under the Catalog.**

**To Drop Synonym:**

**Drop** **synonym** "KABIL\_PRACTICE"."CUSTOMER";

**Drop** **public** **synonym** "CUSTOMER\_KABIL";

# **Table Types in SAP HANA**

# **Introduction to Table Type: v   A table type is Similar to a database table but do not have an instance Used to define paramet...**

**Introduction to Table Type:**

* **A table type is Similar to a database table but do not have an instance**
* **Used to define parameters for a procedure that represent tabular results.**

**In HANA, with the help of SQLScript, we can create a Table Type.**

**How to Create a Table Type:**

* **A table type is created using statement CREATE TYPE and can be deleted using statement DROP TYPE.**

**Syntax:**

**CREATE TYPE [schema.]name AS TABLE**

**(name1 type1 [, name2 type2,...])**

**DROP TYPE [schema.]name [CASCADE]**

**Example:  
Code:**

**To Create the Type:**

**create** **type** "KABIL\_PRACTICE"."TT\_PROFIT" **as** **Table**

(

"PRODUCTCODE" **nvarchar**(15),

"PRODUCTNAME" **nVarchar** (70),

"BUYPRICE" **decimal**(34,2),

"MSRP" **decimal**(34,2),

"PROFIT" **decimal**(34,2)

);

**To Drop the Type:**

**drop** **type** "KABIL\_PRACTICE"."TT\_PROFIT";

**To Create the Procedure:**

**create** **procedure** "KABIL\_PRACTICE"."TT\_PROCEDURE" ( **OUT** "OUTPUT\_TABLE" "KABIL\_PRACTICE"."TT\_PROFIT" )

**as** **begin**

"OUTPUT\_TABLE" = **Select** "PRODUCTCODE","PRODUCTNAME","BUYPRICE","MSRP",("MSRP"-"BUYPRICE") **as** "PROFIT"

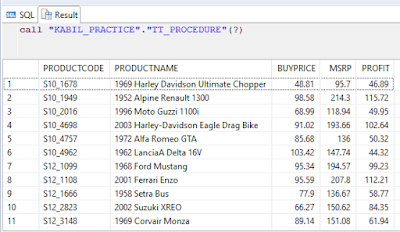
**from** "KABIL\_PRACTICE"."PRODUCTS\_SAP\_STUDENT";

**END**;

**To Call the Procedure:**

**call** "KABIL\_PRACTICE"."TT\_PROCEDURE"(?);

**Result:**

[](https://2.bp.blogspot.com/-x5JiuO27N_g/WZaHHiZyo8I/AAAAAAAAACg/Mrq3ltBBioELmbUWtjQMk2OkUkyCcrvAwCLcBGAs/s1600/Procedure%2BResult.PNG)

# **Create Procedure Using For Loop**

* **The For loop iterates a range of numeric values.**
* **For loop is a programming language conditional iterative statement which is used to check for certain conditions and then repeatedly execute a block of code as long as those conditions are met.**

**BREAK:**

* **Specifies that a loop should stop being processed.**

**CONTINUE:**

* **Specifies that a loop should stop processing the current iteration, and should immediately start processing the next.**

**FOR LOOP EXAMPLE 1:**

**To create a Procedure with For loop:**

**Create** **procedure** "KABIL\_PRACTICE"."FORLOOP\_1"

**as** **begin**

**declare** i **integer**;

**create** **column** **table** "KABIL\_PRACTICE"."FOR\_LOOP\_1" ("ID" **integer**);

**for** i **in** 1..10 **do**

**insert** **into** "KABIL\_PRACTICE"."FOR\_LOOP\_1" **values**(:i);

**end** **for**;

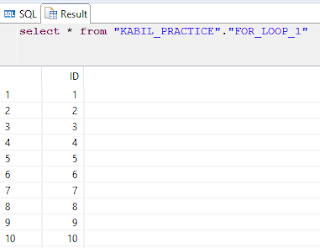
**end**;

**To call the Procedure:**

**call** "KABIL\_PRACTICE"."FORLOOP\_1";

**select** \* **from** "KABIL\_PRACTICE"."FOR\_LOOP\_1"

**Result:**

[](https://2.bp.blogspot.com/-nS8Qh2WgT7k/WZu4NykK70I/AAAAAAAAADs/RQy0jIQ9LMgzwY330KfJ2ll7cqx7BW9wACLcBGAs/s1600/result%2B1.PNG)

**FOR LOOP EXAMPLE 2:**

**To create the table for inserting a records:**

**CREATE** **COLUMN** **TABLE** "KABIL\_PRACTICE"."FOR\_LOOP\_2"(LOOP\_1 **INT**);

**To create a Procedure with For loop:**

**create** **procedure** "KABIL\_PRACTICE"."FOR\_LOOP\_TEST"

**AS**

**BEGIN**

**DECLARE** N\_START **INTEGER** :=3;

**DECLARE**

N\_STOP **INTEGER** :=6;

**DECLARE** I **INT**;

**FOR** I **IN** N\_START .. N\_STOP **DO**

N\_STOP :=100;

**INSERT** **INTO**  "KABIL\_PRACTICE"."FOR\_LOOP\_2" **VALUES** (I);

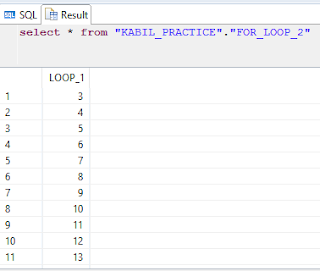
**END** **FOR**;

**END**;

**To call the Procedure:  
CALL** "KABIL\_PRACTICE"."FOR\_LOOP\_TEST";

**select** \* **from** "KABIL\_PRACTICE"."FOR\_LOOP\_2"

**Result:**

[](https://3.bp.blogspot.com/-Zx53jk3B5hs/WZu5AT59IuI/AAAAAAAAAD4/I1sSn1T8EsEnZxac2lCEp8WoyzQtb3SEwCLcBGAs/s1600/for%2Bloop%2B2.PNG)

**Note:**

**Here the value starts from “3” and ends at “100”.**

**FOR LOOP EXAMPLE 3:**

**To create the table for inserting a records:**

**CREATE** **COLUMN** **TABLE** "KABIL\_PRACTICE"."FOR\_LOOP\_3"(LOOP\_1 **INT**);

**To create a Procedure with For loop:**

**create** **procedure** "KABIL\_PRACTICE"."FOR\_LOOP\_TEST1"(**IN** N\_START **INT**,**IN** N\_STOP **INT**)

**AS**

**BEGIN**

**DECLARE** I **INT**;

**FOR** I **IN** :N\_START .. :N\_STOP **DO**

**INSERT** **INTO**  "KABIL\_PRACTICE"."FOR\_LOOP\_3" **VALUES** (I);

**END** **FOR**;

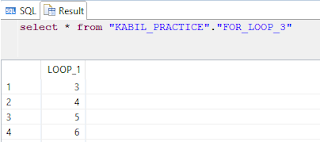
**END**;

**To call the Procedure:**

**CALL** "KABIL\_PRACTICE"."FOR\_LOOP\_TEST1" (3,6);

**select** \* **from** "KABIL\_PRACTICE"."FOR\_LOOP\_3"

**Result:**

[](https://2.bp.blogspot.com/-vZqZTIJEWIE/WZu5pqLk70I/AAAAAAAAAEA/XrlGeXCOsx8uwo__hVQO-6EbhaJNjK9GwCLcBGAs/s1600/result-%2Bfor%2Bloop%2B3.PNG)

**FOR LOOP EXAMPLE 4:**

**To create a Procedure with For loop:**

**create** **procedure** "KABIL\_PRACTICE"."FOR\_LOOP\_TEST2"(**IN** N\_START **INT**,**IN** N\_STOP **INT**)

**AS**

**BEGIN**

**DECLARE** I **INT**;

**FOR** I **IN** REVERSE :N\_START .. :N\_STOP **DO**

**INSERT** **INTO**  "KABIL\_PRACTICE"."FOR\_LOOP\_4" **VALUES** (I);

**END** **FOR**;

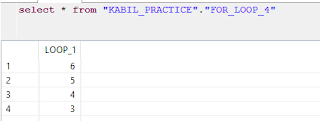
**END**;

**To call the Procedure:**

**CALL** "KABIL\_PRACTICE"."FOR\_LOOP\_TEST2" (3,6);

**select** \* **from** "KABIL\_PRACTICE"."FOR\_LOOP\_4"

**Result:**

[](https://4.bp.blogspot.com/-q5-wUZ4SWZg/WZu6IoloIiI/AAAAAAAAAEI/fliJckpV038suHX5ms_jZAr21j1Tf0hywCLcBGAs/s1600/result-%2Bfor%2Bloop%2B4.PNG)

**Note:**

**Reverse keyword -**it inserts the values in a reverse order.

**FOR LOOP EXAMPLE 5:**

**CREATE** **COLUMN** **TABLE** "KABIL\_PRACTICE"."FACTORIAL\_LOOP"

(

 RESULT **INT**

);

**To create a Procedure with For loop to find the factorial :**

**CREATE** **PROCEDURE** "KABIL\_PRACTICE"."FACTORIAL\_USING\_LOOP"(**IN** FACT **INT**)

**AS** **BEGIN**

**DECLARE** HOLD **INT** :=1;

**DECLARE** I **INT**;

**FOR** I **IN** REVERSE 1 .. FACT **DO**

HOLD :=HOLD\*I;

**END** **FOR**;

**INSERT** **INTO** "KABIL\_PRACTICE"."FACTORIAL\_LOOP" **VALUES**(HOLD);

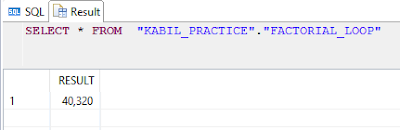
**END**;

**To call the Procedure:**

**CALL** "KABIL\_PRACTICE"."FACTORIAL\_USING\_LOOP" (8);

**SELECT** \* **FROM**  "KABIL\_PRACTICE"."FACTORIAL\_LOOP";

**Result:**

[](https://1.bp.blogspot.com/-U3Z8run02GY/WZu6iSpv4ZI/AAAAAAAAAEM/M1EHjCwOeD0xmbqQsHhi6QnTYCtzzeeRgCLcBGAs/s1600/result%2Bfactorial%2Bloop.PNG)

# **While Loop in SAP HANA**

* **A while loop will check the condition first and then executes the block of Sql Statements within it as long as the condition evaluates to true.**

**Syntax:**

**WHILE Condition**

**BEGIN**

**Statements**

**END**

* **Sets a condition for the repeated execution of an SQL statement or statement block. The statements are executed repeatedly as long as the specified condition is true. The execution of statements in the WHILE loop can be controlled from inside the loop with the BREAK and CONTINUE keywords.**

**While Loop Example:**

**To create Table:**

**create** **column** **table** "KABIL\_PRACTICE"."FACTORIAL" ("NUMBER" **integer**, "FACTORIAL" **double** );

**To create a Procedure:**

**create** **procedure** "KABIL\_PRACTICE"."WHILE\_LOOP\_FACTORIAL\_PROCEDURE"

(**in** I **integer**)

**as** **begin**

**declare**X **integer** := 2;

**declare** F **double** := 1;

**while** X <= :I **do**

F := :X \* :F;

X := :X + 1;

**end** **while**;

**insert** **into** "KABIL\_PRACTICE"."FACTORIAL" **values** (I,F);

**select** \* **from** "KABIL\_PRACTICE"."FACTORIAL";

**end**;

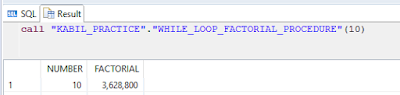
**To Call a Procedure:**

**call** "KABIL\_PRACTICE"."WHILE\_LOOP\_FACTORIAL\_PROCEDURE"(10);

**To Drop Procedure:**

**Drop Procedure** "KABIL\_PRACTICE"."WHILE\_LOOP\_FACTORIAL\_PROCEDURE";

**Result:**

[](https://4.bp.blogspot.com/-SdLOQxUYCtE/WZvjI-VEcOI/AAAAAAAAAEo/X057q4-8O0sUmg1Vt3Jjig-mRaXJQBlhQCLcBGAs/s1600/factorial%2Bresult.PNG)

# **Arrays in SAP HANA**

**ARRAYS:**

**An array is an indexed collection of elements of a single data type. In the following section we explore the varying ways to define and use arrays in SQLScript.**

**UNNEST FUNCTION**

**The UNNEST function converts one or many arrays into a table. The result table includes a row for each element of the specified array. The result of the UNNEST function needs to be assigned to a table variable.**

**DECLARE ARRAY-TYPED VARIABLE**

**An array-typed variable will be declared by using the keyword ARRAY.**

**Syntax:**

**DECLARE <variable\_name> <sql\_type> ARRAY;**

**You can declare an array <variable\_name> with the element type <sql\_type>.**

**The following SQL types are supported:**

**DATE**

**TIME**

**TIMESTAMP**

**SECONDDATE**

**TINYINT**

**SMALLINT**

**INTEGER**

**BIGINT**

**DECIMAL**

**SMALLDECIMAL**

**REAL**

**DOUBLE**

**VARCHAR**

**NVARCHAR**

**ALPHANUM**

**VARBINARY**

**CLOB**

**NCLOB**

**BLOB**

**You can declare the arr array of type INTEGER as follows:**

**DECLARE arr INTEGER ARRAY;**

**To Create a Column:**

**create** **column** **table** "KABIL\_PRACTICE"."arrayproc\_tbl" ("a" **nvarchar**(50),"b" **nvarchar**(50));

**To Create a Procedure:**

**CREATE** **PROCEDURE** "KABIL\_PRACTICE"."ARRAYPROC"

(**IN** a **NVARCHAR**(20), **IN** b **NVARCHAR**(20))

**AS**

**BEGIN**

**DECLARE** arrayNvarchar **NVARCHAR**(20) ARRAY;

arrayNvarchar := ARRAY(:a,:b);

**insert** **into** "KABIL\_PRACTICE"."arrayproc\_tbl" **values** (:arrayNvarchar[1],:arrayNvarchar[2]);

**END**;

**To call a Procedure:**

**call** "KABIL\_PRACTICE"."ARRAYPROC"(1,2);

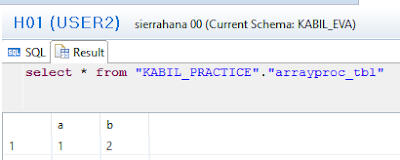
**To Drop a Procedure:**

**drop** **procedure** "KABIL\_PRACTICE"."ARRAYPROC";

**To see the output:**

**select** \* **from** "KABIL\_PRACTICE"."arrayproc\_tbl";

**Result:**

[](https://1.bp.blogspot.com/-48e4uNMHd0Y/WaPvFmcLe1I/AAAAAAAAAFE/AcgkC6E8XTYT8YPaGo2VHWdxqzY4XtW5gCLcBGAs/s1600/Result.PNG)

**Example 2:**

**To Create a table:**

**create** **column** **table** "KABIL\_PRACTICE"."Array\_Proc\_Dec"("A" **nvarchar**(10), "B" **nvarchar**(10));

**To Create a Procedure:**

**create** **procedure** "KABIL\_PRACTICE"."Array\_Proc\_Declaration"

**as**

**begin**

**DECLARE** array\_int **INTEGER** ARRAY := ARRAY(3, 4);

--array\_int := ARRAY(:A,:B);

**insert** **into** "KABIL\_PRACTICE"."Array\_Proc\_Dec" **values**(:array\_int[1],:array\_int[2]);

**END**;

**To Call a Procedure:**

**call** "KABIL\_PRACTICE"."Array\_Proc\_Declaration"();

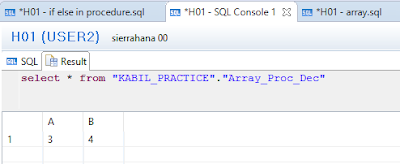
**(OR)**

**call** "KABIL\_PRACTICE"."Array\_Proc\_Declaration";

**To see the output:**

**select** \* **from** "KABIL\_PRACTICE"."Array\_Proc\_Dec"

**Result:**

[](https://2.bp.blogspot.com/-8XU4R24Irzw/WaQAq-zqklI/AAAAAAAAAFU/nxonaovlBZ87J2wD4K9PtIxLS3bakmvYQCLcBGAs/s1600/Res%2Bdeclaration.PNG)

# **Date and Time Functions in SAP HANA**

## ADD\_DAYS

**Syntax**

 ADD\_DAYS (d, n)

**Description**

Computes the date d plus n days.

**Example**

**SELECT** ADD\_DAYS (TO\_DATE ('2009-12-05', 'YYYY-MM-DD'), 30) "add days" **FROM** DUMMY;

**Output For ADD\_DAYS:**

            04, jan 2010

## ADD\_MONTHS

**Syntax**

 ADD\_MONTHS (d, n)

**Description**

Computes the date d plus n months.

**Example**

**SELECT** ADD\_MONTHS (TO\_DATE ('2009-12-05', 'YYYY-MM-DD'), 1) "add months" **FROM** DUMMY;

**Output For ADD MONTHS:**

 05 jan, 2010

## ADD\_SECONDS

**Syntax**

 ADD\_SECONDS (t, n)

**Description**

Computes the time t plus n seconds.

**Example**

**SELECT** ADD\_SECONDS (TO\_TIMESTAMP ('2012-01-01 23:30:45'), 60\*30) "add seconds" **FROM** DUMMY;

**OUTPUT FOR ADD SECONDS**

2 Jan, 2012 12:00:45.0 AM

## ADD\_YEARS

**Syntax**  
 ADD\_YEARS (d, n)

**Description**

Computes the date d plus n years.

**Example**

**SELECT** ADD\_YEARS (TO\_DATE ('2009-12-05', 'YYYY-MM-DD'), 1) "add years" **FROM** DUMMY;

**OUTPUT FOR ADD YEARS**

5 Dec, 2010

## CURRENT\_DATE

**Syntax**

 CURRENT\_DATE

**Description**

Returns the current local system date.

**Example**

**SELECT** **CURRENT\_DATE** "current date" **FROM** DUMMY;

**OUTPUT FOR** **CURRENT DATE**

 31 Aug, 2017

## CURRENT\_TIME

**Syntax**

CURRENT\_TIME

**Description**

Returns the current local system time.

**Example**

**Select** **current\_time** " CURRENT TIME" **From** dummy;

**OUTPUT FOR CURRENT\_TIME**  
11:35:05 PM

## DAYNAME (d):

 This function returns the weekday in English for date d , which you need to pass it.

**Example:**

**SELECT**  **DAYNAME** ('2017-09-04') "DayName" **FROM** DUMMY;

**Result:**

 MONDAY

## DAYOFMONTH (d):

This function returns an integer the day of the month for date d , which you need to pass it.

**Example:**

**SELECT**  DAYOFMONTH ('2017-09-04') "DayOfMonth" **FROM** DUMMY;

**Result:**

1. **DAYOFYEAR (d):**

**This function returns an integer representation of the day of the year for date d, which you need to pass it.**

**Example:**

**SELECT  DAYOFYEAR ('2017-09-04') "DayOfYear" FROM DUMMY;**

**Result:**

**247**

## **DAYS\_BETWEEN (d1, d2):**

**It returns the number of days between two days d1 and d2**

**Example:**

**SELECT** **DAYS\_BETWEEN** (TO\_DATE ('2017-09-04', 'YYYY-MM-DD'), TO\_DATE('2017-09-24', 'YYYY-MM-DD')) "DaysBetween" **FROM** Dummy;

**Result:**

**20**

## **EXTRACT ({YEAR | MONTH | DAY | HOUR | MINUTE | SECOND} FROM d):**

**This function returns the value of a specified DateTime field from date d, like you can get a year, month, day, hour, minute and second.**

**Example:**

**SELECT**  **EXTRACT** (**YEAR** **FROM** TO\_DATE ('2017-01-04', 'YYYY-MM-DD')) "Extracted Value" **FROM** DUMMY;

**Result:**

2017

# **SAP HANA: Cursor in Procedure**

**What is Cursor?**

1. Cursor is used to fetch single rows from the result set returned by a query.
2. Cursors can be defined either after the signature of the procedure and before the procedure’s body or at the beginning of a block with the DECLARE token.
3. The cursor is defined with a name, optionally a list of parameters, and an SQL SELECT statement.
4. The cursor provides the functionality to iterate through a query result row-by-row.
5. Updating cursors are not supported.

**Note:**

Avoid using cursors when it is possible to express the same logic with SQL. You should do this as cursors cannot be optimized the same way SQL can.

**To Create a Table:**

**CREATE** **COLUMN** **TABLE** "KABIL\_PRACTICE"."PRODUCT\_DETAILS\_CUR" (

          "PRODUCT\_ID" **INTEGER** **PRIMARY** **KEY**,

          "PRODUCT\_NAME" **VARCHAR**(100),

          "PRICE" **FLOAT**

);

To insert records into the table:

**INSERT** **INTO** "KABIL\_PRACTICE"."PRODUCT\_DETAILS\_CUR" **VALUES**(1,'SHIRTS', 500);

**INSERT** **INTO** "KABIL\_PRACTICE"."PRODUCT\_DETAILS\_CUR" **VALUES**(2,'JACKETS', 2000);

**INSERT** **INTO** "KABIL\_PRACTICE"."PRODUCT\_DETAILS\_CUR" **VALUES**(3,'TROUSERS', 1000);

**INSERT** **INTO** "KABIL\_PRACTICE"."PRODUCT\_DETAILS\_CUR" **VALUES**(4,'COATS', 5000);

**INSERT** **INTO** "KABIL\_PRACTICE"."PRODUCT\_DETAILS\_CUR" **VALUES**(5,'PURSE', 800);

**To Create a Table Type:**

**CREATE** **TYPE** "KABIL\_PRACTICE"."TT\_PRODUCT\_DETAILS\_CUR" **AS** **TABLE**(

          "PRODUCT\_ID" **INTEGER** **PRIMARY** **KEY** ,

          "PRODUCT\_NAME" **VARCHAR**(100),

          "PRICE" **FLOAT**

);

**To Drop a Table Type:**

**drop** **type** "KABIL\_PRACTICE"."TT\_PRODUCT\_DETAILS\_CUR";

**To Create a Procedure:**

**CREATE** **PROCEDURE** "KABIL\_PRACTICE"."CURSOR\_EXAMPLE" (

**IN** ip\_rate **DECIMAL**(15,2),

**OUT** ex\_products "KABIL\_PRACTICE"."TT\_PRODUCT\_DETAILS\_CUR")

**LANGUAGE** SQLSCRIPT

**SQL** SECURITY INVOKER

**AS**

**BEGIN**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

    Write your procedure logic

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**DECLARE** v\_new\_price **DECIMAL**(15,2);

**DECLARE** **CURSOR** c\_products **FOR**

**SELECT** PRODUCT\_ID, PRODUCT\_NAME, PRICE

**FROM** "KABIL\_PRACTICE"."PRODUCT\_DETAILS\_CUR";

**FOR** cur\_row **as** c\_products **DO**

    v\_new\_price := cur\_row.PRICE + (cur\_row.PRICE \* :ip\_rate);

**UPDATE** "KABIL\_PRACTICE"."PRODUCT\_DETAILS\_CUR"

**SET** PRICE = v\_new\_price **where** PRODUCT\_ID = cur\_row.PRODUCT\_ID;

**END** **FOR**;

 ex\_products = **select** PRODUCT\_ID, PRODUCT\_NAME, PRICE

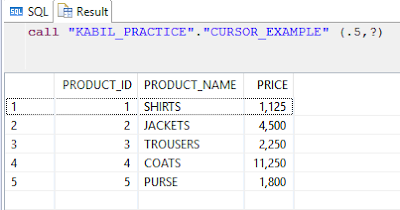
**FROM** "KABIL\_PRACTICE"."PRODUCT\_DETAILS\_CUR" ;

**END**;

**To Call a Procedure:**

**call** "KABIL\_PRACTICE"."CURSOR\_EXAMPLE" (.5,?);

**Result:**

[](https://4.bp.blogspot.com/-YQnOXlRbh6A/WcH44iwEJuI/AAAAAAAAAeM/2o1FHeM4pykOWsPToQQZqTiNTXEexgWTgCLcBGAs/s1600/Result.PNG)

# **SAP HANA: Dynamic Filter Using Procedure**

# Dynamic Filter in HANA

**In this article, we will show an example on - How to implement and call dynamic filtering in a procedure.**

**Introduction:**

* **SAP does not recommend the use of dynamic SQL (EXEC statement) when developing SQLScript procedures.**
* **For dynamic SQL, we now have a new statement in SQLScript called APPLY\_FILTER.**

**This statement accepts two parameters.**

* **The first parameter is the dataset in which you want to apply the filter.  This dataset can be a database table, database view, HANA attribute or calculation view, or even an intermediate table variable.**
* **The second parameter is the filter condition itself. This would be very similar syntax that you would use in the WHERE clause of a SELECT statement.**

**In this example, we are going to use table PRODUCT and pass the filter condition dynamically while calling the procedure. The output will be sent to output using a table type.**

**Create Table:**

**Copy and paste the below script in SQL editor and execute.**

**Note: If you already have created the PRODUCT table in previous example, then skip this step.**

**To Create a Table:**

**create** **column** **table** "KABIL\_PRACTICE"."PRODUCT\_DF"(

      "PRODUCT\_ID" **VARCHAR**(10),

      "PRODUCT\_NAME" **VARCHAR** (100),

    "CATEGORY" **VARCHAR**(100),

**primary** **key** ("PRODUCT\_ID")

);

**To Insert a records into the Table:**

**insert** **into** "KABIL\_PRACTICE"."PRODUCT\_DF" **values**('P1','Shirts', 'Clothes');

**insert** **into** "KABIL\_PRACTICE"."PRODUCT\_DF" **values**('P2','Jackets', 'Clothes');

**insert** **into** "KABIL\_PRACTICE"."PRODUCT\_DF" **values**('P3','Trousers', 'Clothes');

**insert** **into** "KABIL\_PRACTICE"."PRODUCT\_DF" **values**('P4','Coats', 'Clothes');

**insert** **into** "KABIL\_PRACTICE"."PRODUCT\_DF" **values**('P5','Purse', 'Accessories');

**To Create a Table Type:**

**CREATE** **TYPE** "KABIL\_PRACTICE"."TT\_PRODUCT\_DF" **AS** **TABLE**(

      "PRODUCT\_ID" **VARCHAR**(10),

      "PRODUCT\_NAME" **VARCHAR** (100),

      "CATEGORY" **VARCHAR** (100)

);

**To Create a Procedure:**

**CREATE** **PROCEDURE** "KABIL\_PRACTICE"."PROCEDURE\_DYNAMIC\_FILTER"(

**IN** im\_filter\_string **VARCHAR**(5000),

**OUT** output\_table "KABIL\_PRACTICE"."TT\_PRODUCT\_DF" )

**LANGUAGE** SQLSCRIPT

**SQL** SECURITY INVOKER

**AS**

/\*\*\*\*\*\*\*\*\*BEGIN PROCEDURE SCRIPT \*\*\*\*\*\*\*\*\*\*\*\*/

**BEGIN**

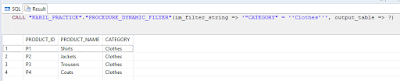
    output\_table = APPLY\_FILTER("KABIL\_PRACTICE"."PRODUCT\_DF", :im\_filter\_string) ;

**END**;

**To Call a Procedure:**

**CALL** "KABIL\_PRACTICE"."PROCEDURE\_DYNAMIC\_FILTER"(im\_filter\_string => '"CATEGORY" = ''Clothes''', output\_table => ?);

**Result:**

[](https://3.bp.blogspot.com/-ruqzp7KLDA4/WcH07kyR93I/AAAAAAAAAds/GZLUj4OS59wpZhqgvLgZ7rlO9Nk4IH6NQCLcBGAs/s1600/Df_Result.PNG)

# **SAP HANA: How to copy tables from one schema to another schema in SAP HANA?**

1. How to create a new table based on the existing table in another schema without data i.e., only the structure of the table?

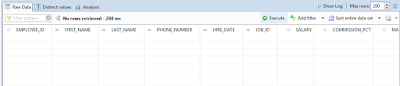
**SYNTAX:**

**create** **column** **table** "New\_Schema\_Name"."New\_Table\_Name" **like** "Old\_Schema\_Name"."Old\_Table\_Name";

**CODE:**

**create** **column** **table** "KABIL"."EMPLOYEE\_SAMPLE" **like** "KABIL\_PRACTICE"."EMPLOYEE\_SAMPLE";

**RESULT:**

[](https://4.bp.blogspot.com/-hnhBsz_-sY8/WbostWSVS1I/AAAAAAAAAbI/4PHDWJURmyoEulZq27XmITb3q5xyOITeACLcBGAs/s1600/structure%2Bonly.PNG)

**2. How to create a new table based on the existing table in another schema with data?**

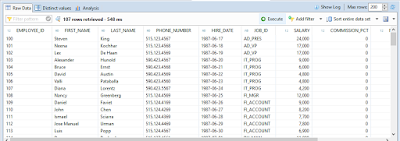
**SYNTAX:**

**create** **column** **table** "New\_Schema\_Name"."New\_Table\_Name" **as** ( **select** \* **from** "Old\_Schema\_Name"."Old\_Table\_Name");

**CODE:**

**create** **column** **table** "KABIL"."EMPLOYEE\_SAMPLE" **as** ( **select** \* **from** "KABIL\_PRACTICE"."EMPLOYEE\_SAMPLE");

**RESULT:**

[](https://3.bp.blogspot.com/-hMKTrwszLqA/Wbosy0FrBdI/AAAAAAAAAbM/zHUd2D61FLMNM8QgiXq8Ufi1WknboKZ3ACLcBGAs/s1600/with%2Bdata.PNG)

# **SQL:**

# Alter:

**Create Column Table:**

**create** **column** **table** "YSATHISH"."COLUMN\_TABLE"

( "EMPNO" **varchar**(4) **not** **null**,

"NAME" **varchar**(20),

"SALARY" **INTEGER** CS\_INT,

"DOJ" **DATE** cs\_DAYDATE );

### **How to add the primary key to existing table?**

**Syntax:**

ALTER TABLE schema.table ADD PRIMARY KEY (column1,column2)

**Ex:**

**alter** **table** "YSATHISH"."COLUMN\_TABLE" **ADD** **PRIMARY** **KEY** ("EMPNO");

### **How to change the table column store to Row store?**

**Syntax:**

**Alter table "KABIL\_PRACTICE"."DEMO" Row;**

### **To alter Row Store to Column Store table:**

**EX:**

**Alter table "KABIL\_PRACTICE"."DEMO" column;**

### **Alter table to add an additional column.**

**Ex:**

**Alter table "KABIL\_PRACTICE"."DEMO" Add ("CITY" Nvarchar(35));**

### **Alter table to change the data type of an existing column.**

**Ex:**

**Alter table "KABIL\_PRACTICE"."DEMO" Alter ("CITY" varchar(35));**

### **Note:**

* **For column table, only increasing the size of a column data type is allowed to prevent data loss.**
* **For example, changing from NVARCHAR (20) to NVARCHAR (10) or from INTEGER to TINYINT raises an error.**
* **For row table, only increasing the size of VARCHAR and NVARCHAR type column is allowed. Other data type changes are not allowed.**

**Ex:**

### **To add a Salary column:**

**Alter table "KABIL\_PRACTICE"."DEMO" Add ("Salary" Decimal(10,2));**

### **To alter a Salary Column data type:**

**Alter** **table** "KABIL\_PRACTICE"."DEMO" **Alter** ("Salary" **Decimal**(15,2));

            It raises an error like feature not supported: cannot change the field length: Salary

### **To Alter Existing Column with NOT NULL**

**Alter** **table** "KABIL\_PRACTICE"."DEMO" **Alter** ("ID" **integer** **NOT** **NULL**);

**Adding a NOT NULL constraint to an existing column is allowed if either of the following cases are true:**

* **The table is empty.**
* **The default value is specified when the table contains data.**
* **The table does not contain a NULL-value in that column.**

### **To Add Primary Key for Existing table column**

**Alter** **table** "KABIL\_PRACTICE"."DEMO" **Add** **primary** **key** ("ID");

### **To Add Primary Key for Multiple Columns in an Existing table**

**Alter** **table** "KABIL\_PRACTICE"."DEMO" **Add** **primary** **key** ("ID","NAME");

### **To Drop Primary Key for Existing table column**

**Alter** **table** "KABIL\_PRACTICE"."DEMO" **drop** **primary** **key**;

### **Alter table to Drop column.**

**Alter** **table** "KABIL\_PRACTICE"."DEMO" **Drop** ("CITY");

### **COMMENT ON:**

* **This statement is used to add descriptive comments to tables, views or their individual columns in SAP HANA system.**
* **Comments are a useful way to record a meaningful description of schema tables/views, and their columns, for future reference.**
* **We can also remove comments using same statement.**

### **Examples:**

### **To create COMMENT ON for DEMO table in “KABIL\_PRACTICE” Schema**

**COMMENT** **ON** **TABLE** "KABIL\_PRACTICE"."DEMO" **IS** 'DEMO Table for Alter Statements';

### **Note:**

* **we can see this comment by Right click your table from the corresponding schema and click open definition. A new tab will open from that select Further Properties, you can see the comment of your table.**

### **To Remove comment on Table**

**COMMENT** **ON** **TABLE** "KABIL\_PRACTICE"."DEMO" **IS** **null**;

### **To create COMMENT ON ID Column from DEMO table in “KABIL\_PRACTICE” Schema**

**COMMENT** **ON** **COLUMN** "KABIL\_PRACTICE"."DEMO"."ID" **is** 'EMPLOYEE\_ID';

**Note:**

**we can see this comment by Right click your table from the corresponding schema and click open definition. A new tab will open with that comment will be shown in the corresponding column.**

#### **To Remove comment on Column**

**COMMENT** **ON** **COLUMN** "KABIL\_PRACTICE"."DEMO"."ID" **is** **null**;

### **To Rename Column**

**Rename** **column** "KABIL\_PRACTICE"."DEMO"."ID"  **To** "E\_ID";

### **To Rename Table**

**Rename** **Table** "KABIL\_PRACTICE"."DEMO" **To** "DEMO\_TABLE";

# **SAP HANA: Insert into with Select Statement**

### **Example 1:**

To insert a record for a Particular Columns in a table:

### **Code:**

**INSERT** **INTO** "KABIL\_PRACTICE"."PROC\_TEST" ("ID","NAME") (**SELECT** 1,'KABIL' **FROM** DUMMY);

### **Example 2:**

To insert a record in a table:

### **Code:**

**INSERT** **INTO** "KABIL\_PRACTICE"."PROC\_TEST" (**SELECT** 1,'KABIL' **FROM** DUMMY);

**INSERT** **INTO** "KABIL\_PRACTICE"."PROC\_TEST" **SELECT** 1,'KABIL' **FROM** DUMMY;

### **Example 3:**

**To insert a record in a table with some Calculations:**

### **Code:**

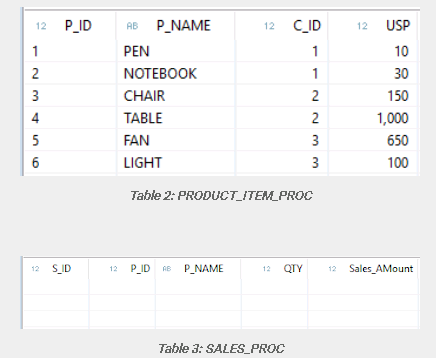
**INSERT INTO "KABIL\_PRACTICE"."PROC\_TEST" VALUES (1, TO\_CHAR(ADD\_DAYS(CURRENT\_DATE,1)));**

# **SAP HANA: Functions in Procedures**

### **Scenario:**

* **Whenever I enter a record into the sales table, it will affect the other tables based on that entry. Such as if I make an entry in “SALES\_PROC” for 20 PEN sold it will affect the “PRODUCT\_MASTER\_PROC” table Stock.**
* **And also “SALES\_AMOUNT” column in “SALES\_PROC” table will be calculated automatically based “USP” (Unit Selling Price) column in a “PRODUCT\_ITEM\_PROC” table.**
* **Let’s see for details, here I used 3 tables named as "PRODUCT\_MASTER\_PROC", "PRODUCT\_ITEM\_PROC" and "SALES\_PROC" as shown below:**

****

****

### **CODE:**

### **To create a PRODUCT\_MASTER\_PROC table:**

**create column table "KABIL\_PRACTICE"."PRODUCT\_MASTER\_PROC"**

**(**

**"P\_ID" integer Primary Key,**

**"P\_NAME" nvarchar(35),**

**"C\_ID" integer,**

**"Category" nvarchar(35),**

**"Stock" integer**

**);**

### **To insert a record in a PRODUCT\_MASTER\_PROC table:**

**insert into "KABIL\_PRACTICE"."PRODUCT\_MASTER\_PROC" values (1,'PEN',1,'STAIONARY',150);**

**insert into "KABIL\_PRACTICE"."PRODUCT\_MASTER\_PROC" values (2,'NOTEBOOK',1,'STAIONARY',100);**

**insert into "KABIL\_PRACTICE"."PRODUCT\_MASTER\_PROC" values (3,'CHAIR',2,'FURNITURE',150);**

**insert into "KABIL\_PRACTICE"."PRODUCT\_MASTER\_PROC" values (4,'TABLE',2,'FURNITURE',150);**

**insert into "KABIL\_PRACTICE"."PRODUCT\_MASTER\_PROC" values (5,'FAN',3,'ELECTRICALS',150);**

**insert into "KABIL\_PRACTICE"."PRODUCT\_MASTER\_PROC" values (6,'LIGHT',3,'ELECTRICALS',150);**

#### **To create a PRODUCT\_ITEM\_PROC table:**

**create column table "KABIL\_PRACTICE"."PRODUCT\_ITEM\_PROC"**

**(**

**"P\_ID" integer primary key,**

**"P\_NAME" nvarchar(35),**

**"C\_ID" integer,**

**"USP" Decimal (34,2),**

**foreign key("P\_ID") references "KABIL\_PRACTICE"."PRODUCT\_MASTER\_PROC"("P\_ID")**

**);**

#### **To insert a record in a PRODUCT\_ITEM\_PROC table:**

**insert into "KABIL\_PRACTICE"."PRODUCT\_ITEM\_PROC" values (1,'PEN',1,10);**

**insert into "KABIL\_PRACTICE"."PRODUCT\_ITEM\_PROC" values (2,'NOTEBOOK',1,30);**

**insert into "KABIL\_PRACTICE"."PRODUCT\_ITEM\_PROC" values (3,'CHAIR',2,150);**

**insert into "KABIL\_PRACTICE"."PRODUCT\_ITEM\_PROC" values (4,'TABLE',2,1000);**

**insert into "KABIL\_PRACTICE"."PRODUCT\_ITEM\_PROC" values (5,'FAN',3,650);**

**insert into "KABIL\_PRACTICE"."PRODUCT\_ITEM\_PROC" values (6,'LIGHT',3,100);**

### **To create a SALES\_PROC table:**

**create column table "KABIL\_PRACTICE"."SALES\_PROC"**

**(**

**"S\_ID" integer primary key generated by default as IDENTITY,**

**"P\_ID" integer,**

**"P\_NAME" nvarchar(35),**

**"QTY" integer,**

**"Sales\_AMount" Decimal(34,2),**

**foreign key ("P\_ID") References "KABIL\_PRACTICE"."PRODUCT\_ITEM\_PROC" ("P\_ID")**

**);**

## **TABLE TYPE:**

### **A table type is**

* **Similar to a database table but we can’t insert/update/delete. It contains only structure.**
* **Used to define parameters for a procedure that represent tabular results.**

### **To create a Table Type:**

**create type "KABIL\_PRACTICE"."TT\_SALES\_PRODUCT\_INVENTORY" as table**

**(**

**"S\_ID" integer primary key,**

**"P\_ID" integer,**

**"P\_NAME" nvarchar(35),**

**"QTY" integer,**

**"Sales\_AMount" Decimal(34,2)**

**)**

**Procedure:**

### **To Create a Procedure:**

**create procedure "KABIL\_PRACTICE"."PROC\_SALES\_INVENTORY"**

**(**

**IN "P\_ID" integer,**

**IN "P\_NAME" nvarchar(35),**

**IN "QTY" integer,**

**OUT Result "KABIL\_PRACTICE"."TT\_SALES\_PRODUCT\_INVENTORY"**

**)**

**language SQLSCRIPT**

**as begin**

**declare PROD\_ID integer;**

**declare INV integer;**

**declare EP decimal (34,2);**

**declare SALES\_AMOUNT Decimal(34,2);**

**select count(\*) into PROD\_ID from "KABIL\_PRACTICE"."PRODUCT\_ITEM\_PROC" where "P\_ID" = :P\_ID;**

**select "Stock" into INV from "KABIL\_PRACTICE"."PRODUCT\_MASTER\_PROC" where "P\_ID" =:P\_ID;**

**select "USP" into EP from "KABIL\_PRACTICE"."PRODUCT\_ITEM\_PROC" where "P\_ID" =:P\_ID;**

**IF INV >= :QTY THEN**

**SALES\_AMOUNT := (:QTY \* :EP);**

**insert into "KABIL\_PRACTICE"."SALES\_PROC" ("P\_ID","P\_NAME","QTY","Sales\_AMount") values (:P\_ID,:P\_NAME,:QTY,:SALES\_AMOUNT);**

**update "KABIL\_PRACTICE"."PRODUCT\_MASTER\_PROC" set "Stock" = "Stock" - :QTY where  "P\_ID" = :P\_ID;**

**END IF;**

**Result = select \* from "KABIL\_PRACTICE"."SALES\_PROC";**

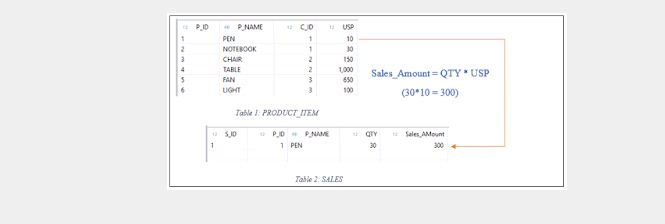
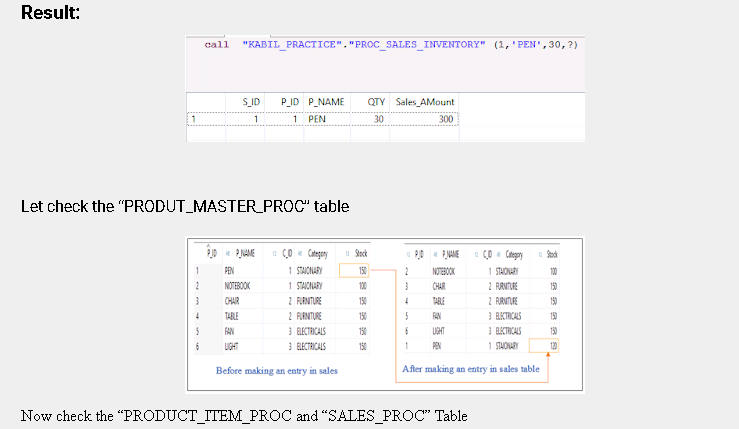
**end;**

### **To Call a Procedure:**

**call "KABIL\_PRACTICE"."PROC\_SALES\_INVENTORY" (1,'PEN',30,?);**

### **Note:**

**Here, I make an entry by mentioning “Product\_Id”, “Product\_Name” and “Quantity” alone.**

****

**Functions:**

### **How to Use Function in Procedure:**

* **From the above, we can see the “SALES\_AMOUNT” is calculated by performing multiplication of quantity with a unit selling price. The same logic can be implemented by using the function.**

### **Example:**

### **To Create a Function:**

**create function "KABIL\_PRACTICE"."SALES\_AMOUNT"**

**( Q integer, USP integer)**

**returns result decimal (34,2)**

**language SQLSCRIPT**

**SQL SECURITY INVOKER as**

**Begin**

**result := :Q \* :USP;**

**END;**

### **To use function in Procedure:**

**create procedure "KABIL\_PRACTICE"."PROC\_SALES\_INVENTORY\_EHE"**

**(**

**IN "P\_ID" integer,**

**IN "P\_NAME" nvarchar(35),**

**IN "QTY" integer,**

**OUT Result "KABIL\_PRACTICE"."TT\_SALES\_PRODUCT\_INVENTORY"**

**)**

**language SQLSCRIPT**

**as begin**

**declare PROD\_ID integer;**

**declare INV integer;**

**declare EP decimal (34,2);**

**declare SALES\_AMOUNT Decimal(34,2);**

**DECLARE empty\_name CONDITION FOR SQL\_ERROR\_CODE 10001;**

**DECLARE EXIT HANDLER FOR empty\_name**

**begin**

**SELECT ::SQL\_ERROR\_CODE, ::SQL\_ERROR\_MESSAGE FROM DUMMY;**

**end;**

**select count(\*) into PROD\_ID from "KABIL\_PRACTICE"."PRODUCT\_ITEM\_PROC" where "P\_ID" = :P\_ID;**

**select "Stock" into INV from "KABIL\_PRACTICE"."PRODUCT\_MASTER\_PROC" where "P\_ID" =:P\_ID;**

**select "USP" into EP from "KABIL\_PRACTICE"."PRODUCT\_ITEM\_PROC" where "P\_ID" =:P\_ID;**

**IF INV >= :QTY THEN**

**--- calling Functions in procedure(Here “SALES\_AMOUNT 🡪 Function)**

**SALES\_AMOUNT := "KABIL\_PRACTICE"."SALES\_AMOUNT"(:QTY,:EP);**

**(or)**

**/\*--select "KABIL\_PRACTICE"."SALES\_AMOUNT"(:QTY,:EP) into SALES\_AMOUNT from dummy;---\*/**

**insert into "KABIL\_PRACTICE"."SALES\_PROC" ("P\_ID","P\_NAME","QTY","Sales\_AMount") values (:P\_ID,:P\_NAME,:QTY,:SALES\_AMOUNT);**

**update "KABIL\_PRACTICE"."PRODUCT\_MASTER\_PROC" set "Stock" = "Stock" - :QTY where  "P\_ID" = :P\_ID;**

**END IF;**

**Result = select \* from "KABIL\_PRACTICE"."SALES\_PROC";**

**end;**

**Note:**

The highlighted area shows the various ways to use function in HANA Procedures.

# **SAP HANA: Table User Defined Functions**

**Table User Define Function always returns a table and its parameter may also be off table type.**

**Table UDFs are read-only user-defined functions which accept multiple input parameters and return exactly one results table.  SQLScript is the only language which is supported by table UDF's. Since these functions are read-only, only read-only statements can be used within the function.  So, you may not use statements like INSERT, UPDATE or DELETE. Also, any procedure calls within the function must also be read-only.  Currently, you can only create these functions in the catalog via the SQL Editor.**

### **Syntax:**

**CREATE FUNCTION FUNCTION\_NAME**

**(Parameter1 DATA\_TYPE,Parameter2 DATA\_TYPE)**

**RETURNS TABLE**

**( RetruenValue1 DATA\_TYPE, ReturnValue2 DATA\_TYPE )**

**BEGIN**

**RETURN SELECT 1\* Parameter1 AS RetruenValue1 ,1\*Parameter2 AS ReturnValue2 FROM mytab;**

**END;**

### **Table User-defined Functions Support the Following:**

* **They can have any number of input parameters**
* **They return exactly one table**
* **Table operations are allowed within the body**
* **They are used in the FROM clause of SELECT statements**
* **They must be free of side-effects, i.e. DDL statements or the DML statements insert, update and delete cannot be used in function bodies**

|  |
| --- |
| **The SQL statement to define table user-defined functions also is create function. The difference to the definition of a scalar user-defined function is keyword table for the return parameter and how the value of the sole return parameter is assigned using keyword return:** |
|  |

### **Basic Syntax to Define a Table User-Defined Function**

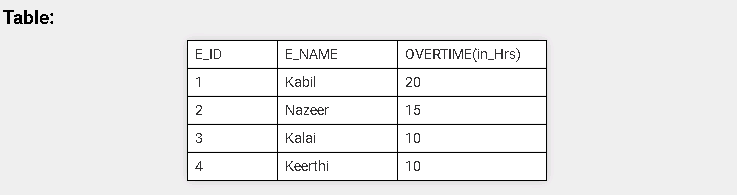
**CREATE FUNCTION <function name> (<list of input parameters with type>) RETURNS TABLE [table type| (<list of table column definitions>)] AS BEGIN**

**<function body>**

**RETURN <expression to set return table>**

**END;**

**A feature that can be helpful in the context of defining table user-defined functions or stored procedures is dynamic filtering using built-in table function “apply\_filter”. This function allows you to apply a dynamic WHERE clause to a database table or table variable and assign the result to a table variable.**

****

### **To Create a  Function:**

**CREATE FUNCTION "KABIL\_PRACTICE"."Convert\_OfficialsHours"**

**(im\_filter VARCHAR(1000), im\_to VARCHAR(1) )**

**RETURNS TABLE ( "E\_ID" NVARCHAR(3),**

**"E\_NAME" NVARCHAR(20),**

**Overtime DEC (5,2))**

**AS BEGIN**

**lt\_official = APPLY\_FILTER("KABIL\_PRACTICE"."OVERTIME"**

**,:im\_filter);**

**RETURN SELECT "E\_ID", "E\_NAME",**

**"KABIL\_PRACTICE"."Convert\_Hours\_IF\_ELSE"("OVERTIME(in\_Hrs)",:im\_to) AS Overtime**

**FROM :lt\_official;**

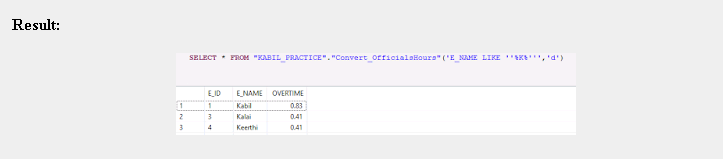
**END;**

### **Note:**

* **Apply\_filter: You can use “APPLY\_FILTER” with persistent tables and table variables.**
* **The “APPLY\_FILTER” function applies a dynamic filter on a table or table variable. Logically it can be considered a partial dynamic SQL statement. The advantage of the function is that you can assign it to a table variable and will not block SQL – inlining. Despite this, all other disadvantages of a full dynamic SQL yield also for the APPLY\_FILTER.**
* **"KABIL\_PRACTICE"."Convert\_Hours\_IF\_ELSE": It is a Scalar User Defined function which is already created.**

### **To call a Function:**

**SELECT** \* **FROM** "KABIL\_PRACTICE"."Convert\_OfficialsHours"('E\_NAME LIKE ''%K%''','d');

****

# **SAP HANA : Scalar User Defined Functions**

## User Defined Function in SAP HANA:

User Defined Functions in SAP HANA are the read only function that means we cannot perform any DDL and DML (insert update and delete) operation inside the body of the function. There are two types of user-defined functions: Scalar User Defined Functions and Table User Defined Functions.

There are two types of user define the function in SAP HANA:

1. Table User Define Function (Table UDF)

2. Scalar User Define Function (Scalar UDF)

These categories are defined on the basis of input/output parameter, supported functionality in the body of the function and how they are going to consume in the SQL statement.

### Scalar Used-Defined Functions

The simplest database object for which you can make use of SQL Script are Scalar User- Defined Functions (Scalar UDFs)

Scalar UDFs allow you to the define functions which take a number of input parameters and return scalar values. Only expressions are allowed in the body of the UDF, so no table operations, CE functions or array operations.

**Scalar User-Defined Functions Support the Following:**

         They can have any number of scalar input parameters (primitive SQL types). Input parameters of table type are not supported.

         They can return multiple scalar values.

         They can contain expressions within their body. Table and array operations are not supported.

         They can be used in the field list or the WHERE clause of SELECT statements — like built-in functions.

         They are callable via direct assignment in other user-defined functions or stored procedures

         (x := my\_scalar\_func () ).

         They must be free of side-effects and do not support any type of SQL statement in their body.

The SQL statement to define user-defined functions is create function. The basic syntax to define a scalar user-defined function looks as follows:

#### Basic Syntax to Define a Scalar User-Defined Function

CREATE FUNCTION <function name> (<list of input parameters with

type>)

RETURNS <scalar result parameter name and type>

AS BEGIN

<function body>

END;

  You can create scalar user-defined functions for use like built-in functions.

  Prefix parameter names with “**:**”to access their values.

#### Code:

##### To Create a table:

**create** **column** **table** "KABIL\_PRACTICE"."OVERTIME"

 (

 "E\_ID" **integer**,

 "E\_NAME" **Nvarchar**(35),

 "OVERTIME(in\_Hrs)" **Integer**

);

##### To Drop a table:

**Drop** **table** "KABIL\_PRACTICE"."OVERTIME";

##### To Insert the values into a table:

**Insert** **into** "KABIL\_PRACTICE"."OVERTIME" **values** (1,'Kabil',20);

**Insert** **into** "KABIL\_PRACTICE"."OVERTIME" **values** (2,'Nazeer',15);

**Insert** **into** "KABIL\_PRACTICE"."OVERTIME" **values** (3,'Kalai',10);

##### To Create function:

**Create** **Function** "KABIL\_PRACTICE"."CONVERT\_HRS"

(i\_Hours **integer**)

**returns**  result **Decimal**(34,2)

**as** **begin**

result := :i\_Hours \* 60;

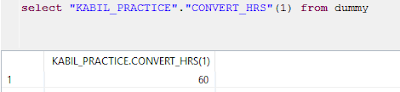
**End**;

**NOTE:**

            We should not mention parameter name inside the double quotes. If we did it throws an error such as inappropriate variable name: do not allow "" or '\_SYS\_' prefix for the name of variable or parameter

##### To Call the function using Dummy:

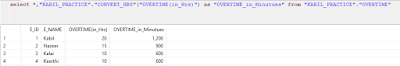
**select** "KABIL\_PRACTICE"."CONVERT\_HRS"(1) **from** dummy;

**Result:**  
[](https://2.bp.blogspot.com/-fb4iIa1QwW8/Wg1Av60fV7I/AAAAAAAABaU/V5FpUy3lO5oTYVxNf_vLZYmesfeudk5uwCLcBGAs/s1600/Convert%2Bhours%2Bfn%2Bresult%2Bfrom%2Bdummy.PNG)

##### To Use the function with the Table:

**select** \*,"KABIL\_PRACTICE"."CONVERT\_HRS"("OVERTIME(in\_Hrs)") **as** "OVERTIME\_in\_Minutues" **from** "KABIL\_PRACTICE"."OVERTIME";

**Result:**

[](https://4.bp.blogspot.com/-DYf5NG-EjLk/Wg1BFOt6g-I/AAAAAAAABac/XzTqzlRHjcsRqwDu9hgHvKBCGH_j9CxQACLcBGAs/s1600/function%2Bin%2Btable%2Bresult.PNG)

##### To Drop a function:

**Drop** **function** "KABIL\_PRACTICE"."CONVERT\_HRS";

**NOTE:**

The only way to change the body of an existing user-defined function using SQL statements is to delete the function and re-create it.

You can also use imperative logic in scalar user-defined functions, to the extent this does not conflict with the statements above. Imperative language constructs allow the developer to control data and control flow, for example loops, scalar variables and if-then-else statements.

### Imperative Logic in SQLScript

Imperative logic allows you to control the flow of the logic

         Scalar variable manipulation

         Branching logic, for example using if-then-else

         Loops — while and for

         DDL statements and insert, update and delete statements

Imperative logic is executed exactly as scripted and procedural. It prevents parallel processing.

**Note:**

DDL and DML are not supported in scalar user-defined functions anyway. They can be used in table user-defined functions and database procedures.

#### IF\_THEN\_ELSE Statements in Function Bodies:

#### CODE:

##### To Create Function IF\_ELSE:

**CREATE** **FUNCTION** "KABIL\_PRACTICE"."Convert\_Hours\_IF\_ELSE"(im\_hours **INTEGER**,im\_to **VARCHAR**(1))

**RETURNS** ex\_result **DEC** (5,2)

**AS** **BEGIN**

**IF** :im\_to = 'm' **THEN**

ex\_result := :im\_hours \* 60;

**ELSEIF** :im\_to = 'd' **THEN**

ex\_result := :im\_hours / 24;

**ELSE**

ex\_result := :im\_hours;

**END** **IF**;

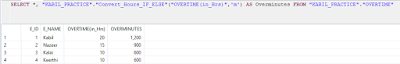
**END**;

##### To Use the Function with the Table:

##### CODE 1:

**SELECT** \*, "KABIL\_PRACTICE"."Convert\_Hours\_IF\_ELSE"("OVERTIME(in\_Hrs)",'m') **AS** Overminutes **FROM** "KABIL\_PRACTICE"."OVERTIME";

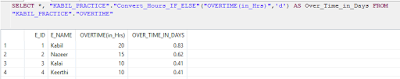
**Result**:

[](https://1.bp.blogspot.com/-AeZ8pND-whY/Wg1BUOWsfXI/AAAAAAAABak/qOQ8tgGaL6wTYRsXgWQaCvK2GqraQmoSQCLcBGAs/s1600/Fn%2Bwith%2Bif_else_M.PNG)

##### CODE 2:

**SELECT** \*, "KABIL\_PRACTICE"."Convert\_Hours\_IF\_ELSE"("OVERTIME(in\_Hrs)",'d') **AS** Overminutes **FROM** "KABIL\_PRACTICE"."OVERTIME";

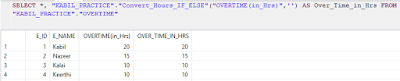
**Result**:

[](https://1.bp.blogspot.com/-QLOOpfSL8rY/Wg1BfZPr24I/AAAAAAAABao/w_WdQjLCnvcoKKaAbnPon9Esd2CKQ6LSgCLcBGAs/s1600/Fn%2Bwith%2Bif_else_D.PNG)

##### CODE 3:

**SELECT** \*, "KABIL\_PRACTICE"."Convert\_Hours\_IF\_ELSE"("OVERTIME(in\_Hrs)",'') **AS** Overminutes **FROM** "KABIL\_PRACTICE"."OVERTIME";

**Result**:

[](https://1.bp.blogspot.com/-alDGSZcb6W8/Wg1BqK2PqiI/AAAAAAAABas/d8CrMQQLfQ86_NPb7hYBXDCd-3wM7MbawCLcBGAs/s1600/Fn%2Bwith%2Bif_else_empty.PNG)

# **SAP HANA: How to DELETE duplicate records from the Table but keep original record?**

### How to delete duplicate records from the Table but keep the original record?

### Scenario:

|  |  |  |
| --- | --- | --- |
| **DEPTID** | **DEPTNAME** | **PRICE** |
| 1 | Blog | 2000 |
| 2 | Article | 5000 |
| 3 | Resource | 7000 |
| 4 | Book | 500 |
| 4 | Book | 500 |
| 1 | Blog | 1000 |

From the above table, DEPTID 4 have some duplicate records. i.e., DEPTID, DEPTNAME, PRICE columns having the same value.

## Solution:

### Code:

**delete** **from** "ADZSUPRI\_PRACTICE"."cpy\_tab\_Department"

**where** "$rowid$" **in**

(

**SELECT**   LEAD("$rowid$") over (**partition** **by** DEPTID,DEPTNAME,"price" **order** **by** DEPTID,DEPTNAME)

**from** "ADZSUPRI\_PRACTICE"."cpy\_tab\_Department" ;

)

### Result:

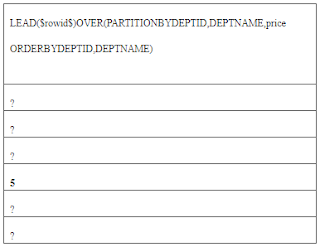
|  |  |  |
| --- | --- | --- |
| **DEPTID** | **DEPTNAME** | **PRICE** |
| 1 | Blog | 2000 |
| 2 | Article | 5000 |
| 3 | Resource | 7000 |
| 4 | Book | 500 |
| 1 | Blog | 1000 |

**Note:**

When we execute the inner query i.e., Sub query we get the result like as shown below:

### Code:

**SELECT**   LEAD("$rowid$") over (**partition** **by** DEPTID,DEPTNAME,"price" **order** **by** DEPTID,DEPTNAME) **from**"ADZSUPRI\_PRACTICE"."cpy\_tab\_Department" ;

[](https://4.bp.blogspot.com/-nb9bI6eAmbk/Wg0ywAzh_qI/AAAAAAAABZY/ClPQvNRmMxkyg1re1Fsm7e0W7P1tuHayQCLcBGAs/s1600/123.PNG)

**ROWID**

         For each row in the database, the rowid pseudo column returns the address of the row.

         Usually, a rowid value uniquely identifies a row in the database.

         Rowid values have several important uses:

  They are the fastest way to access a single row.

  They can show you how the rows in a table are stored.

  They are unique identifiers for rows in a table.

# **SAP HANA: SELECT Statements**

HANA SQL Restricting and Sorting data: [9 exercises with solution]

**1. Write a query to display the name (first\_name, last\_name) and salary for all employees whose salary is not in the range $10,000 through $15,000.**

**CODE:**

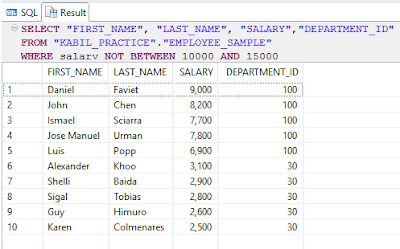
**SELECT** "FIRST\_NAME", "LAST\_NAME", "SALARY","DEPARTMENT\_ID"

**FROM** "KABIL\_PRACTICE"."EMPLOYEE\_SAMPLE"

**WHERE** salary **NOT** **BETWEEN** 10000 **AND** 15000

**AND** "DEPARTMENT\_ID" **IN** (30, 100);

**RESULT:**

[](https://4.bp.blogspot.com/-f3uEy0sViQM/WbohskNTDXI/AAAAAAAAAZ4/DyYNbGaR7PUpJYCk9envGT67FqZRXuhkwCLcBGAs/s1600/salary%2Bnot%2Bbtwn%2B10000%2Band%2B15000.PNG)

2**. Write a query to display the name (first\_name, last\_name) and department ID of all employees in departments 30 or 100 in ascending order.**

**CODE:**

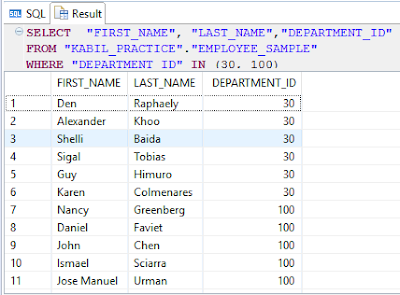
**SELECT**  "FIRST\_NAME", "LAST\_NAME","DEPARTMENT\_ID"

**FROM** "KABIL\_PRACTICE"."EMPLOYEE\_SAMPLE"

**WHERE** "DEPARTMENT\_ID" **IN** (30, 100)

**ORDER** **BY**  "DEPARTMENT\_ID"  **ASC**;

**RESULT:**

[](https://3.bp.blogspot.com/-vdaPCf2X-as/Wboh35E8L6I/AAAAAAAAAaA/HzGCcKCAqxUhjF4RINt7xkztNqt6MVDkQCLcBGAs/s1600/Dept_is%2Bin%2B30%2Band%2B100.PNG)

**3. Write a query to display the name (first\_name, last\_name) and salary for all employees whose salary is not in the range $10,000 through $15,000 and are in department 30 or 100.**

**CODE:**

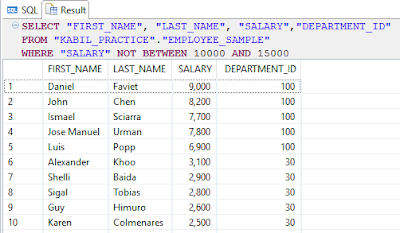
**SELECT** "FIRST\_NAME", "LAST\_NAME", "SALARY","DEPARTMENT\_ID"

**FROM** "KABIL\_PRACTICE"."EMPLOYEE\_SAMPLE"

**WHERE** "SALARY" **NOT** **BETWEEN** 10000 **AND** 15000

**AND** "DEPARTMENT\_ID" **IN** (30, 100);

**RESULT:**

[](https://3.bp.blogspot.com/-rT3OEFrRSVc/Wboh4g9TN1I/AAAAAAAAAaU/kIyHSFQsQ48MA9lrVS-IqFOght1CJ34TACEwYBhgL/s1600/salary%2Bnot%2Bbtwn%2B10000%2Band%2B15000%2Band%2Bdept%2Bid%2Bin%2B30%2Band%2B100.PNG)

4. Write a query to display the name (first\_name, last\_name) and hire date for all employees who were hired in 1987.

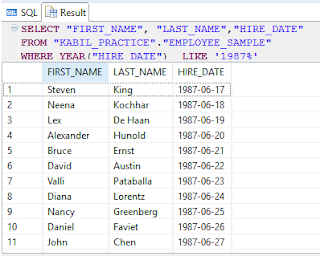
**CODE:**

**SELECT** "FIRST\_NAME", "LAST\_NAME","HIRE\_DATE"

**FROM** "KABIL\_PRACTICE"."EMPLOYEE\_SAMPLE"

**WHERE** **YEAR**("HIRE\_DATE")  **LIKE** '1987%';

**RESULT:**

[](https://3.bp.blogspot.com/-9U1aXNlm8GY/Wboh451d34I/AAAAAAAAAaY/W7UEDU-KSuAS3BdlF5ciujvHgmZQdeZsACEwYBhgL/s1600/year%2B1987.PNG)

5. Write a query to display the first\_name of all employees who have both "b" and "c" in their first name.

**CODE:**

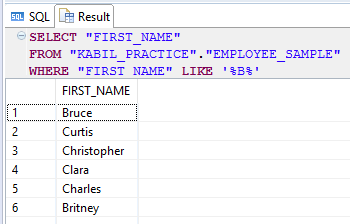
**SELECT** "FIRST\_NAME"

**FROM** "KABIL\_PRACTICE"."EMPLOYEE\_SAMPLE"

**WHERE** "FIRST\_NAME" **LIKE** '%B%'

**OR**  "FIRST\_NAME"**LIKE** '%C%';

**RESULT:**

[](https://3.bp.blogspot.com/-Yk1Uqahy4qc/Wboh30glueI/AAAAAAAAAaE/5FWUTQgmu0MoN0c_KUgrgna22cK_VhhugCEwYBhgL/s1600/f-name%2Bb%2Bor%2Bc.PNG)

6. Write a query to display the last name, job, and salary for all employees whose job is that of a Programmer or a Shipping Clerk, and whose salary is not equal to $4,500, $10,000, or $15,000.

**CODE:**

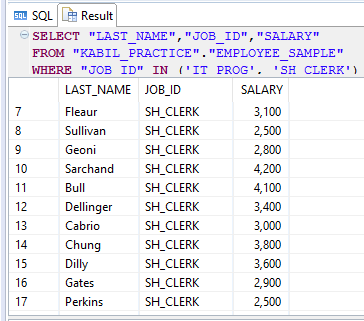
**SELECT** "LAST\_NAME","JOB\_ID","SALARY"

**FROM** "KABIL\_PRACTICE"."EMPLOYEE\_SAMPLE"

**WHERE** "JOB\_ID" **IN** ('IT\_PROG', 'SH\_CLERK')

**AND** "SALARY"**NOT** **IN** (4500,10000, 15000);

**RESULT:**

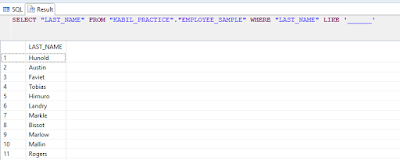
[](https://3.bp.blogspot.com/-gIZhSdnHxo8/Wboh4WJWJsI/AAAAAAAAAag/tK0FosBdzPci6SGou10X-IbG1STDOLLNQCEwYBhgL/s1600/job%2Bid.PNG)

7. Write a query to display the last name of employees whose names have exactly 6 characters.

**CODE:**

**SELECT** "LAST\_NAME" **FROM** "KABIL\_PRACTICE"."EMPLOYEE\_SAMPLE" **WHERE** "LAST\_NAME" **LIKE** '\_\_\_\_\_\_';

**RESULT:**

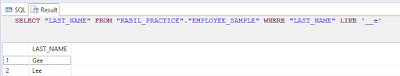
[](https://4.bp.blogspot.com/-42laRzaJz1U/Wboh4btWU-I/AAAAAAAAAag/AXpHmQTvqa0WEFoKi0JqyzbwGzwZdGnvQCEwYBhgL/s1600/like%2B6.PNG)

8. Write a query to display the last name of employees having 'e' as the third character.

**CODE:**

**SELECT** "LAST\_NAME" **FROM** "KABIL\_PRACTICE"."EMPLOYEE\_SAMPLE" **WHERE** "LAST\_NAME" **LIKE** '\_\_e';

**RESULT:**

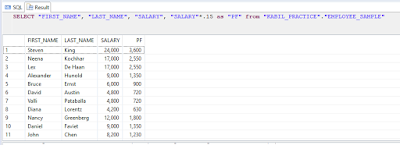
[](https://3.bp.blogspot.com/-0i4wRHWjYhI/Wboh4hZEluI/AAAAAAAAAag/Zcdd2oqgL6AZAsMi56aheNzYFwfyO7YZQCEwYBhgL/s1600/like%2Be.PNG)

9. Write a query to display the name (first\_name, last\_name), salary and PF (15% of salary) of all employees.

**CODE:**

**SELECT** "FIRST\_NAME", "LAST\_NAME", "SALARY", "SALARY"\*.15 **as** "PF" **from** "KABIL\_PRACTICE"."EMPLOYEE\_SAMPLE";

**RESULT:**

[](https://4.bp.blogspot.com/-dR8AOmL1U5Y/Wboh30guVbI/AAAAAAAAAag/9WhhaCX8qgAGLFo17WkQnu1sI6h0gShmgCEwYBhgL/s1600/PF.PNG)

# **SAP HANA: ORDER BY, GROUP BY and HAVING Clause**

### SQL ORDER BY Statement

The ORDER BY clause is used in aSELECT statement to sort results either in ascending or descending order. SAP HANA sorts query results in ascending order by default.

Order by should appear at the end of your select statement after your WHERE, GROUP BY and HAVING clauses if any or all of them exist.

### Syntax for using SQL ORDER BY clause to sort data is:

SELECT column-list

FROM table\_name [WHERE condition]

[ORDER BY column1 [, column2, .. columnN] [DESC or ASC]];

**Example:** Employee Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **NAME** | **DEPT** | **AGE** | **SALARY** | **LOCATION** |
| 100 | Kabil | Electrical | 24 | 25000 | Coimbatore |
| 101 | Nazeer | Electronics | 28 | 35000 | Coimbatore |
| 102 | Aravind | Aeronautics | 28 | 35000 | Chennai |
| 103 | Kalai | Electronics | 22 | 20000 | Coimbatore |
| 104 | Keerthi | InfoTech | 25 | 30000 | Bangalore |

### For Example:

If you want to sort the employee table by the salary of the employee, the SQL query would be.

#### CODE 1:

**Select** "NAME","SALARY" **FROM** "KABIL\_PRACTICE"."DEMO\_EMP" **ORDER** **BY** "SALARY";

**Result:**

|  |  |
| --- | --- |
| **NAME** | **SALARY** |
| Kalai | 20000 |
| Kabil | 25000 |
| Keerthi | 30000 |
| Nazeer | 35000 |
| Aravind | 35000 |

#### CODE 2:

**Select** "NAME","SALARY" **FROM** "KABIL\_PRACTICE"."DEMO\_EMP" **ORDER** **BY** 1;

Then the above code “1” indicates the 1st Column which we used in our select statement. Here I used “NAME” as a 1st column. So, the results set order by Name.

**Result:**

|  |  |
| --- | --- |
| **NAME** | **SALARY** |
| Aravind | 35000 |
| Kabil | 25000 |
| Kalai | 20000 |
| Keerthi | 30000 |
| Nazeer | 35000 |

#### CODE 3:

**Select** "NAME","SALARY" **FROM** "KABIL\_PRACTICE"."DEMO\_EMP" **ORDER** **BY** "NAME","SALARY";

The query first sorts the result according to name and then displays it. You can also use more than one column in the ORDER BY clause. If you want to sort the employee table by the name and salary.

**Result:**

|  |  |
| --- | --- |
| **NAME** | **SALARY** |
| Aravind | 35000 |
| Kabil | 25000 |
| Kalai | 20000 |
| Keerthi | 30000 |
| Nazeer | 35000 |

### To Sort Data in Descending Order

By default, the ORDER BY Clause sorts data in ascending order. If you want to sort the data in descending order, you must explicitly specify it as shown below.

#### CODE 4:

**Select** "NAME","SALARY" **FROM** "KABIL\_PRACTICE"."DEMO\_EMP" **ORDER** **BY** "NAME","SALARY" **desc**;

The above query sorts only the column 'salary' in descending order and the column 'name' by ascending order.

**Result:**

|  |  |
| --- | --- |
| **NAME** | **SALARY** |
| Aravind | 35000 |
| Kabil | 25000 |
| Kalai | 20000 |
| Keerthi | 30000 |
| Nazeer | 35000 |

If you want to select both name and salary in descending order, the query would be as given below.

#### CODE 5:

**Select** "NAME","SALARY" **FROM** "KABIL\_PRACTICE"."DEMO\_EMP" **ORDER** **BY** "NAME" **desc**,"SALARY" **desc**;

**Result:**

|  |  |
| --- | --- |
| **NAME** | **SALARY** |
| Nazeer | 35000 |
| Keerthi | 30000 |
| Kalai | 20000 |
| Kabil | 25000 |
| Aravind | 35000 |

### How to use expressions in the ORDER BY Clause?

Expressions in the ORDER BY clause of a SELECT statement**.**

#### For example:

If you want to display employee name, current salary, and a 20% increase in the salary for only those employees for whom the percentage increase in salary is greater than 30000 and in descending order of the increased price, the SELECT statement can be written as shown below

#### CODE 6:

**Select** "NAME","SALARY","SALARY" \* 1.2 **as** NEW\_SALARY **FROM** "KABIL\_PRACTICE"."DEMO\_EMP" **Where** "SALARY" \* 1.2 > 30000 **ORDER** **BY** NEW\_SALARY **desc**;

**Result:**

|  |  |  |
| --- | --- | --- |
| **NAME** | **SALARY** | **NEW\_SALARY** |
| Nazeer | 35000 | 42000 |
| Aravind | 35000 | 42000 |
| Keerthi | 30000 | 36000 |

### SQL GROUP BY Clause

The SQL GROUP BY Clause is used along with the group functions to retrieve data grouped according to one or more columns.

The GROUP BY clause is a SQL command that is used to group rows that have the same values.

The queries that contain the GROUP BY clause are called grouped queries and only return a single row for every grouped item.

### For Example:

If you want to know the total amount of salary spent on each department, the query would be:

#### CODE 1:

**Select** "DEPT", **SUM**("SALARY") **from** "KABIL\_PRACTICE"."DEMO\_EMP" **Group** **by** "DEPT";

**Result:**

|  |  |
| --- | --- |
| **DEPT** | **SUM(SALARY)** |
| Electrical | 25000 |
| Electronics | 55000 |
| Aeronautics | 35000 |
| InfoTech | 30000 |

**NOTE:**

The group by clause should contain all the columns in the select list expect those used along with the group functions.

#### CODE 2:

**Select** "LOCATION","DEPT", **SUM**("SALARY") **from** "KABIL\_PRACTICE"."DEMO\_EMP" **Group** **by** "LOCATION","DEPT";

**Result:**

|  |  |  |
| --- | --- | --- |
| **LOCATION** | **DEPT** | **SUM(SALARY)** |
| Coimbatore | Electrical | 25000 |
| Coimbatore | Electronics | 55000 |
| Chennai | Aeronautics | 35000 |
| Bangalore | InfoTech | 30000 |

### SQL GROUP BY with HAVING Clause

Having clause is used to filter data based on the group functions. This is similar to WHERE condition but is used with group functions. Group functions cannot be used in WHERE Clause but can be used in HAVING clause.

### For Example:

If you want to select the department that has total salary paid for its employees more than 30000, the SQL query would be like;

#### CODE 1:

**Select** "DEPT",**SUM**("SALARY") **from** "KABIL\_PRACTICE"."DEMO\_EMP" **Group** **by** "DEPT" **Having** **SUM**("SALARY") > 30000 ;

**Result:**

|  |  |
| --- | --- |
| **DEPT** | **SUM(SALARY)** |
| Electronics | 55000 |
| Aeronautics | 35000 |

**Note:**

We can use Order by at the end of the above code:

#### CODE 2:

**Select** "DEPT", **SUM**("SALARY") **from** "KABIL\_PRACTICE"."DEMO\_EMP" **Group** **by** "DEPT" **Having** **SUM**("SALARY") > 30000 **order** **by** "DEPT";

**Result:**

|  |  |
| --- | --- |
| **DEPT** | **SUM(SALARY)** |
| Aeronautics | 35000 |
| Electronics | 55000 |

**Note:**

When WHERE, GROUP BY and HAVING clauses are used together in a SELECT statement, the WHERE clause is processed first, then the rows that are returned after the WHERE clause is executed are grouped based on the GROUP BY clause.

Finally, any conditions on the group functions in the HAVING clause are applied to the grouped rows before the final output is displayed.

# **SAP HANA: Difference between TRUNCATE, DELETE and DROP commands**

### SQL Delete Statement

The DELETE Statement is used to delete rows from a table.

### Syntax of a SQL DELETE Statement

DELETE FROM table\_name [WHERE condition];

**NOTE:**

The WHERE clause in the SQL DELETE command is optional and it identifies the rows in the column that gets deleted. If you do not include the WHERE clause all the rows in the table are deleted, so be careful while writing a DELETE query without WHERE clause.

After performing a DELETE operation, you need to COMMIT or ROLLBACK the transaction to make the change permanent or to undo it. Note that this operation will cause all DELETE triggers on the table to fire.

### Example:

#### To Create a Table:

**create** **column** **table** "KABIL\_PRACTICE"."DEMO1"

(

"ID" **integer**,

"Name" **nvarchar**(35)

);

#### To Insert Some Records into the Table:

**insert** **into** "KABIL\_PRACTICE"."DEMO1" **values** (1,'A');

**insert** **into** "KABIL\_PRACTICE"."DEMO1" **values** (2,'B');

**insert** **into** "KABIL\_PRACTICE"."DEMO1" **values** (3,'C');

**insert** **into** "KABIL\_PRACTICE"."DEMO1" **values** (4,'D');

#### To Delete a Record from the Table:

**Delete** **From** "KABIL\_PRACTICE"."DEMO1" **Where** "ID" = 1;

#### To Delete All the Records from the Table:

**Delete** **From** "KABIL\_PRACTICE"."DEMO1";

### SQL Truncate Statement

         TRUNCATE removes all rows from a table. The operation cannot be rolled back and no triggers will be fired. As such, TRUNCATE is faster and doesn't use as much undo space as a DELETE.

         Does not require a WHERE clause, so you cannot filter rows while Truncating

         Cannot Truncate a table that is referenced by any Foreign Key constraint.

### Syntax to TRUNCATE a table:

TRUNCATE TABLE table\_name;

### Example:

#### To Truncate a Table:

**Truncate** **Table** "KABIL\_PRACTICE"."DEMO1";

### Difference between DELETE and TRUNCATE Statements:

|  |  |
| --- | --- |
| **DELETE** | **TRUNCATE** |
| This command deletes only the rows from the table based on the condition given in the where clause or deletes all the rows from the table if no condition is specified | This command is used to delete all the rows from the table. |
| WHERE clause is Optional | There is no WHERE clause |
| But it does not free the space containing the table. | It frees the space containing the table. |

## SQL DROP Statement:

         The SQL DROP command is used to remove an object from the database.

         If you drop a table, all the rows in the table is deleted and the table structure is removed from the database.

         Once a table is dropped we cannot get it back, so be careful while using DROP command.

         When a table is dropped all the references to the table will not be valid.

         All the tables' rows, indexes, and privileges will also be removed.

         Cannot drop a table that is referenced by any Foreign Key constraint.

         No DML triggers will be fired.

         The operation cannot be rolled back.

### Syntax to drop a SQL table structure:

DROP TABLE table\_name;

### Example:

#### To Drop a Table:

**Drop** **Table** "KABIL\_PRACTICE"."DEMO1";

### Difference between DROP and TRUNCATE Statement:

|  |  |
| --- | --- |
| **DROP** | **TRUNCATE** |
| If a table is Dropped, Data and the table structure will not exist. | But, if a table is truncated, the table structure remains the same and Data will not exist |
| If a table is dropped, all the relationships with other tables will no longer be valid, the integrity constraints will be dropped, grant or access privileges on the table will also be dropped, if you want to use the table again it has to be recreated with the integrity constraints, access privileges and the relationships with other tables should be established again. | All the relationships with other tables will also be valid. |

# **SAP HANA: LIMIT with OFFSET keywords**

**LIMIT:**

The limit keyword is used to limit the number of rows returned in a  query result.

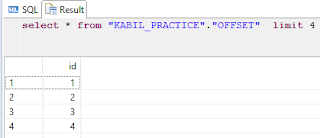
**SYNTAX:**

LIMIT <unsigned\_integer>

**EXAMPLE:**

**select** \* **from** "KABIL\_PRACTICE"."OFFSET"  **limit** 4;

**RESULT:**

[](https://1.bp.blogspot.com/-GFu6zRUJzI0/WbeRFWiMOTI/AAAAAAAAAWE/quPC4I8Wj1YWDc96ZAr0UvosYpA5QksZgCLcBGAs/s1600/limit%2Bonly.PNG)

**LIMIT with OFFSET:**

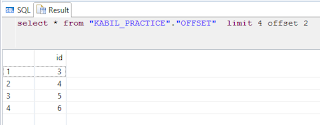
Returns the first <unsigned\_integer> grouped records after skipping OFFSET<unsigned\_integer> for each grouping set.

**SYNTAX:**

LIMIT <unsigned\_integer> [OFFSET <unsigned\_integer>]

**EXAMPLE:**

**select** \* **from** "KABIL\_PRACTICE"."OFFSET"  **limit** 4 offset 2;

**RESULT:**  
[](https://2.bp.blogspot.com/-vSZqavhUEFk/WbeRKpQn6tI/AAAAAAAAAWI/pk29iLwagMo2PGeRe4cexNg7p_3Lzp-ggCLcBGAs/s1600/limit%2Bwith%2Boffset%2Bresult.PNG)

In this post i'm going explain the use **with** result **view**use in SAP HANA Procedure.

WITH RESULT VIEW <view\_name>

 <view\_name> ::= <identifier>

Specifies the result view to be used as the output of a read-only procedure.

**Example - Creating an SQL Procedure with Result View:**

**create** **procedure** "KABIL\_PRACTICE"."ARRAY\_UNNEST\_SIMPLE\_With\_Result\_View"(**OUT** result "KABIL\_PRACTICE"."TTYPE")

**LANGUAGE** SQLSCRIPT

  READS **SQL** DATA **with** result **view** "KABIL\_PRACTICE"."ARRAY\_UNNEST\_RES"

**as**

**begin**

**declare** ID **integer** array := array(1,2);

**declare** NAME **nvarchar**(65) array := Array ('name1','name2','name3');

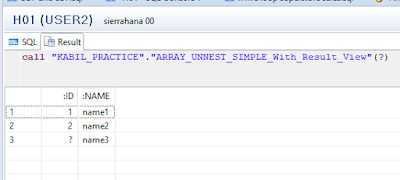
result = UNNEST(:ID,:NAME);

**end**;

**To Call a Procedure:**

**call** "KABIL\_PRACTICE"."ARRAY\_UNNEST\_SIMPLE\_With\_Result\_View"(?);

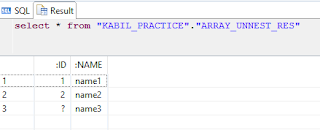
**RESULT:**

[](https://4.bp.blogspot.com/-YyF3F6dk7dM/WbEF8ftH2SI/AAAAAAAAARs/uiXUK3zO_74kVf_r2quv64V04VkueEPiwCLcBGAs/s1600/call%2Bby%2Bnormal.PNG)

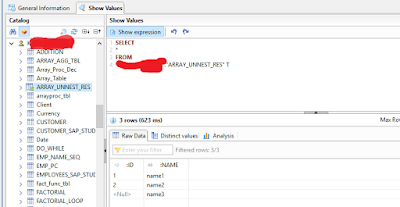
            we can also see the same result by using the select statement with the view name which is we created in a procedure.

**select** \* **from** "KABIL\_PRACTICE"."ARRAY\_UNNEST\_RES";

**RESULT:**

[](https://3.bp.blogspot.com/-DmBEg3rmfOQ/WbEGc9PxF0I/AAAAAAAAAR8/akVedgBmAnYLIQawtrEuYsj5VgFBq7GfACLcBGAs/s1600/call%2Bthe%2Bprocedure%2Bwith%2Bthe%2Bselect%2Bstatement.PNG)

**Result in Information Design Tool:**

[](https://4.bp.blogspot.com/-bA1AlAUO6y4/WbEEkoC_JGI/AAAAAAAAARY/Q6Idg7DcZPgiiPBpNC4Odkfp8nd_QvTbwCLcBGAs/s1600/procedure%2Bwith%2Bresult%2Bview%2Bin%2BIDT.PNG)