Oracle® Cloud

Data Augmentation Scripts Reference Guide

Preview

G52695-01

May 2026

Oracle Cloud Data Augmentation Scripts Reference Guide, Preview

G52695-01

Primary Author: Shahana Mitra

Contributing Authors: Padma Rao

Contents

Preface

Learn how to get started with Data Augmentation Scripts (DAS).

Topics

Audience

Data Augmentation Scripts Reference Guide is for data engineers, administrators, technical support, product managers, and business analysts, who manage data refresh, augmentation, transformation, and pipeline management within the Oracle Fusion Data Intelligence platform.

- **Data Engineers** design, implement, and monitor data pipelines, and manage data extraction, transformation and loading.
- **Administrators** configure and maintain the Oracle Fusion Data Intelligence platform, schedule data pipelines, refresh data, and manage system integrations.
- **Customer Support Engineers** help customers and partners set up and use Oracle Fusion Data Intelligence .
- **Product Managers** and **Business Analysts** create product prototypes.

Related Documentation

These related Oracle resources provide more information.

- Oracle Cloud `http://cloud.oracle.com`
- Getting Started with Oracle Cloud
- Managing and Monitoring Oracle Cloud
- Get Started with Oracle Fusion Data Intelligence
- Using Oracle Fusion Data Intelligence
- Getting Started with Oracle Analytics Cloud
- Visualizing Data and Building Reports in Oracle Analytics Cloud
- Preparing Data in Oracle Analytics Cloud

Conventions

The following text conventions are used in this document.

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Overview of Data Augmentation Scripts

From the Oracle Fusion Data Intelligence Administrator Console, you can build organization-specific or industry-specific data pipelines programmatically with custom logic using Data Augmentation Scripts.

Topics:

Custom Data Pipelines

Data Augmentation Scripts DA Scripts is a declarative ETL language designed to help you build custom data pipelines from the Oracle Fusion Data Intelligence Administrator Console.

Data Augmentation Scripts DA Scripts enables you to ingest data from various sources, such as Oracle Fusion Cloud Applications or Salesforce. You can combine and transform that data, load it into your data warehouse as new tables, and use that data to extend existing entities with supplementary information.

Introduction to Data Augmentation Scripts

Let’s explore Data Augmentation Scripts (DA Scripts) and what you need to know to get started.

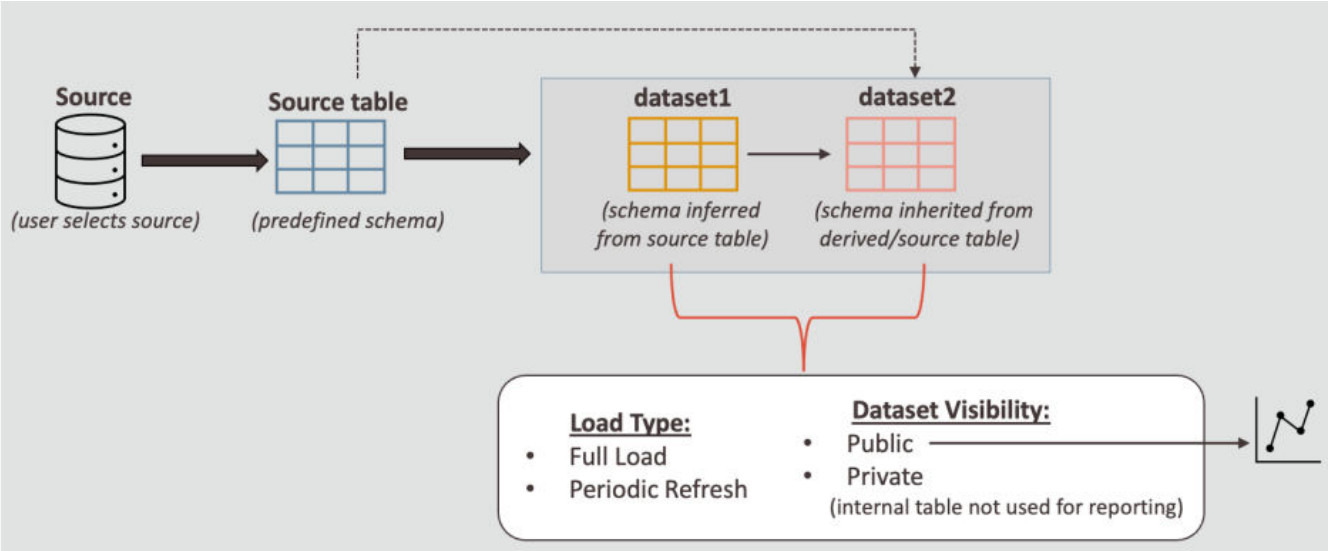
Topics:

Basic Elements of Data Augmentation Scripts

Data Augmentation Scripts(DA Scripts) simplifies the process of extracting, creating, transforming, and managing datasets.

Overview of Data Flow

This diagram illustrates the fundamental data flow, showing how data progresses from the source to the target dataset.



Key Components of Data Flow

The key components of the dataset are:

- **Source Data:** The point of origin for all data processing.
- **Dataset1:** The primary dataset created directly from the source. The schema is inferred from the source.
- **Dataset2:** The secondary dataset that can be derived from Dataset1 or created directly from the source (shown as a dotted line).
- **Load Type:** The refresh type for both tables using the full load or periodic refresh options.
- **Dataset Visibility:** The visibility of the dataset, which can be public (for reporting purposes) or private (for internal processing only).

> **Note**
>
> - Although you can use the terms **dataset** and **table** interchangeably, in the context of Data Augmentation Scripts syntax a dataset or a table is referred to as a *dataset*.
> - You have the ability to override the schema.

Create the Connection Prerequisite

To use the source data, connect to the Data Augmentation Scripts data files from the Oracle Fusion Data Intelligence Administrator Console.

See Create a Connection for Data Augmentation Scripts.

Data Augmentation Scripts Code Structure

Data Augmentation Scripts promotes a clean, modular, and declarative approach to building datasets.

The structure of a Data Augmentation Scripts program mirrors the data flow diagram shown in Basic Elements of Data Augmentation Scripts, organizing your logic into statements and code blocks. Each code block denotes a definition and a transformation block for the target dataset.

Schema

A schema defines the structure and constraints of a target dataset.

Building Blocks of Schema Creation

The foundation of a data scripting program begins with either the source table or the data warehouse tables in a module.

- **Source**: Existing data structures with predefined schemas.
- **Module**: Organized grouping of warehouse tables that assist in analyzing one or more related business processes.

Import Source or Warehouse Tables

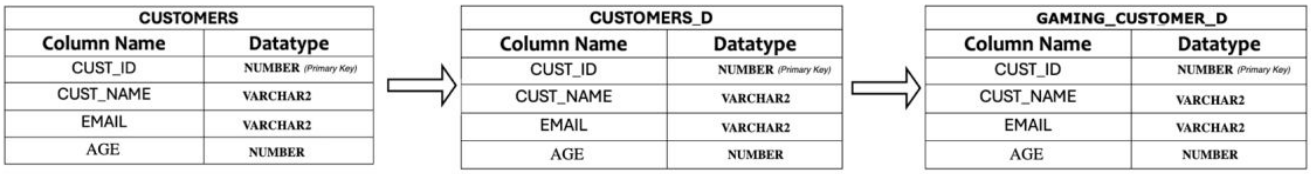
Run these commands to import the source table and module.

```
IMPORT SOURCE SALES   // Import a source table
IMPORT MODULE [FA_GL, FA_AP]                    // Import a single or list
of modules
```

Source tables are read-only definitions that serve as foundational building blocks for data transformations. Modules can be imported and their underlying tables can be used directly in the code.

Dataset Definitions

Datasets are the primary constructs in Data Augmentation Scripts. There are two ways to define datasets: Datasets can inherit their schema directly from a source table or a derived table.




```
IMPORT SOURCE CUSTOMERS
DEFINE DATASET CUSTOMERS_D
    ROWSOURCE CUSTOMERS;
    THIS = CUSTOMERS[CUST_ID];
    THIS = CUSTOMERS[CUST_NAME];
    THIS = CUSTOMERS[EMAIL];
    THIS = CUSTOMERS[AGE];
    PRIMARYKEY[CUST_ID];
 END
 DEFINE DATASET GAMING_CUSTOMER_D
    ROWSOURCE CUSTOMERS_D WHERE CUSTOMERS_D.AGE BETWEEN 13 AND 35;
    THIS = CUSTOMERS_D[CUST_ID];
```

```
    THIS = CUSTOMERS_D[CUST_NAME];
    THIS = CUSTOMERS_D[EMAIL];
    THIS = CUSTOMERS_D[AGE];
    PRIMARYKEY[CUST_ID];
 END
```

These are the key characteristics of a dataset directly inheriting the schema:

- Automatically inherits and preserves the source table's schema
- Eliminates the need for any explicit column definitions .

You can also rewrite the code as:

```
IMPORT SOURCE CUSTOMERS
DEFINE DATASET CUSTOMERS_D FROM CUSTOMERS[CUST_ID,CUST_NAME,EMAIL,AGE] END
DEFINE DATASET GAMING_CUSTOMER_D
    ROWSOURCE CUSTOMERS_D WHERE CUSTOMERS_D.AGE BETWEEN 13 AND 35;
    THIS = CUSTOMERS_D[CUST_ID,CUST_NAME,EMAIL,AGE];
    PRIMARYKEY[CUST_ID];
END
```

An error is displayed if a column that's not present in the corresponding table is referenced.

Example: The following code references the column `GENDER` that's not in the `CUSTOMERS` table.

```
THIS = CUSTOMERS_D[CUST_ID,CUST_NAME,EMAIL,GENDER];
```

An error that `GENDER` is not present in the table `CUSTOMERS` is displayed.

You can also define dataset schemas using the **Custom Schema Definition** feature.

```
DEFINE SCHEMA DW_PROJECT_D_SCHEMA
[
     PROJECT_ID          NUMBER(38,0) PRIMARYKEY,
     PROJECT_NUMBER      VARCHAR2(32),
     START_DATE          DATE       NOT NULL,
     COMPLETION_DATE     DATE
]
END
DEFINE DATASET DW_PROJECT_D
      SCHEMA DW_PROJECT_D_SCHEMA;
      ROWSOURCE PROJECTS;
      THIS[PROJECT_ID]  = PROJECTS[PROJECT_ID, PROJECT_NUMBER];
END
```

For more information about the data types that Data Augmentation Scripts supports, see Data Types.

About Data Population

Data population refers to the process of filling target tables with data derived from a source, where specific source columns are referenced and mapped to target columns, often applying

column manipulation to ensure the data is correctly structured and meets business requirements during the transformation or migration process.

Topics:

- Rowsource
- Column Referencing
- Column Mapping
- Column Manipulation

Rowsource

`ROWSOURCE` defines the initial data before any transformations are performed.

You can consider `ROWSOURCE` as an input from a single table, multiple table joins or unions, and filter conditions, that you can further refine to produce the final dataset.

ROWSOURCE with a single dataset

In its most basic form, `ROWSOURCE` points directly to a single table:


```
IMPORT SOURCE CUSTOMERS
 DEFINE DATASET CUSTOMERS_D
  ROWSOURCE CUSTOMERS;
  THIS = CUSTOMERS;
END
```

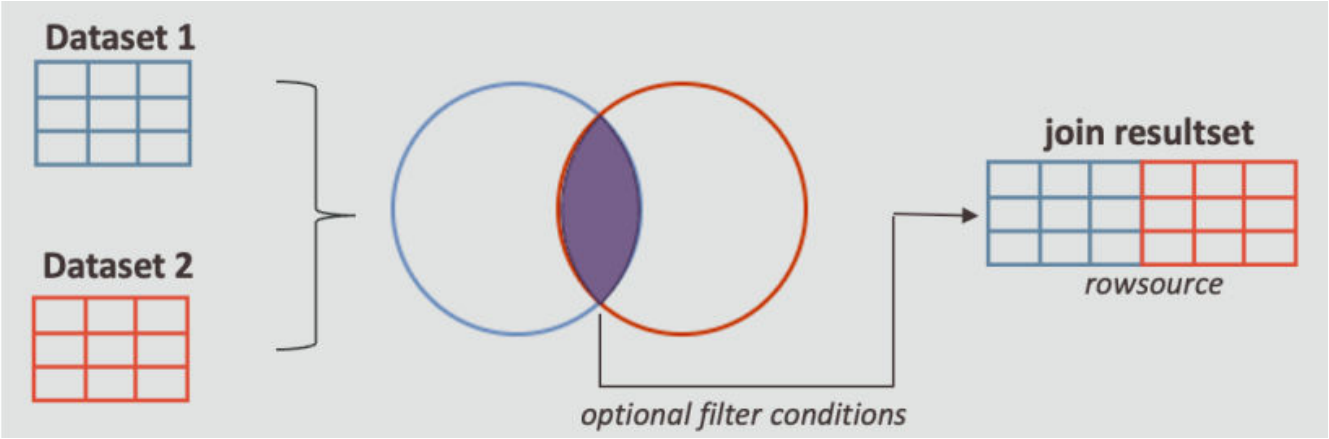
In this example, the `ROWSOURCE` stores all the records from the `CUSTOMERS` table.

You can rewrite this code example in the most compact form as shown:

```
IMPORT SOURCE CUSTOMERS
DEFINE DATASET CUSTOMERS_D FROM CUSTOMERS END
```

ROWSOURCE with multiple datasets

`ROWSOURCE` becomes more powerful when complex operations, such as joins, are performed.




```
IMPORT SOURCE [CUSTOMERS,COUNTRIES]
DEFINE DATASET CUSTOMERS_D
   ROWSOURCE CUSTOMERS;
   THIS = CUSTOMERS;
END
DEFINE DATASET GAMING_CUSTOMER_C
   ROWSOURCE CUSTOMERS_D INNER JOIN COUNTRIES ON (CUSTOMERS_D.COUNTRY_ID =
   COUNTRIES.COUNTRY_ID) WHERE CUSTOMERS_D.CUST_YEAR_OF_BIRTH > 1983;
   THIS = CUSTOMERS_D;
   THIS = COUNTRIES[COUNTRY_NAME,COUNTRY_REGION,COUNTRY_SUBREGION];
   PRIMARYKEY[CUST_ID];
END
```

In this example, the `ROWSOURCE` for creating the target dataset `GAMING_CUSTOMER_C` is created using the `COUNTRIES` source, `CUSTOMERS_D` dataset, and `CUST_YEAR_OF_BIRTH` filter.

Column Referencing

Column referencing is the process of identifying and accessing specific columns from a data source or table.

You can reference a column in three ways:

- `table_name.column_name`
  In this example, the traditional SQL convention of referencing a column is followed (`CUSTOMERS.CUST_ID`).
- `table_name[column_name]`
  Data Augmentation Scripts supports a list of columns so a single column can also be referred to as CUSTOMERS[CUST_ID].
- `THIS.column_name`
  The keyword **`THIS`** refers to the current target table.

  ```
  THIS[ID, FIRST_NAME,LAST_NAME] = CUSTOMERS[CUST_ID,FIRST_NAME,LAST_NAME];
  THIS[FULL_NAME] = CONCAT_WS('_',THIS.FIRST_NAME,THIS.LAST_NAME,THIS.ID) -
  DATATYPE VARCHAR2(18);

  //Instead of repeating the concat_ws logic
  THIS[NEW_ID] =
  ```

```
    UPPER(CONCAT_WS('_',THIS.FIRST_NAME,THIS.LAST_NAME,THIS.ID));

    //we can refer the FULL_NAME using THIS
    THIS[NEW_ID] = UPPER(THIS.FULL_NAME);
```

Column Mapping

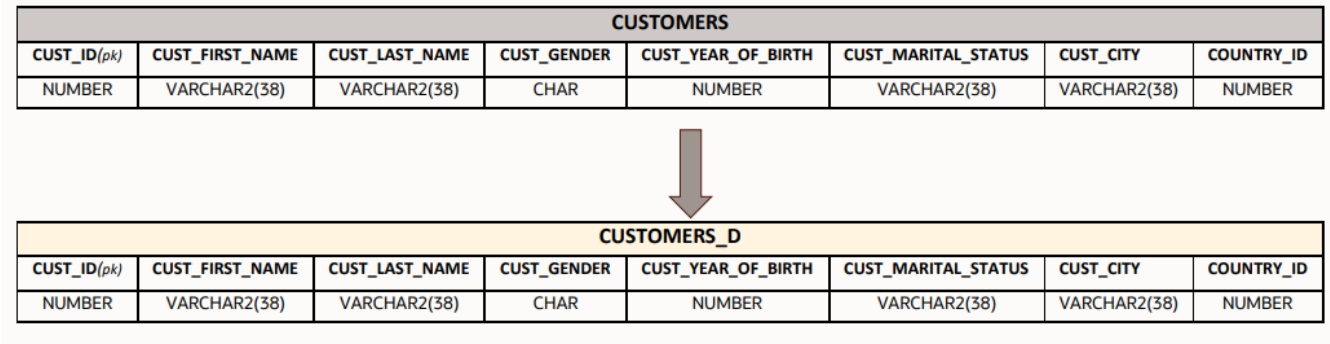
Column mapping is the process of deriving the schema of the target dataset from the source dataset, enabling you to select all columns, select some columns, or exclude columns from the source dataset, based on the needs of the business user.

You can choose to:

- Select all columns
- Select some columns
- Exclude columns

Select all columns

If you select all the columns from the source table or derived dataset, the column properties of the source dataset are inherited by the target dataset. The target dataset contains all the columns from the source or derived dataset with no modifications to the schema.




```
IMPORT SOURCE CUSTOMERS
DEFINE DATASET CUSTOMERS_D
   ROWSOURCE CUSTOMERS;
   THIS = CUSTOMERS;
END
```

In this example, these eight columns from the source table `CUSTOMERS` are in the target table `CUSTOMERS_D` with no modifications to their properties:

- `CUST_ID`
- `CUST_FIRST_NAME`
- `CUST_LAST_NAME`
- `CUST_GENDER`
- `CUST_YEAR_OF_BIRTH`
- `CUST_MARITAL_STATUS`

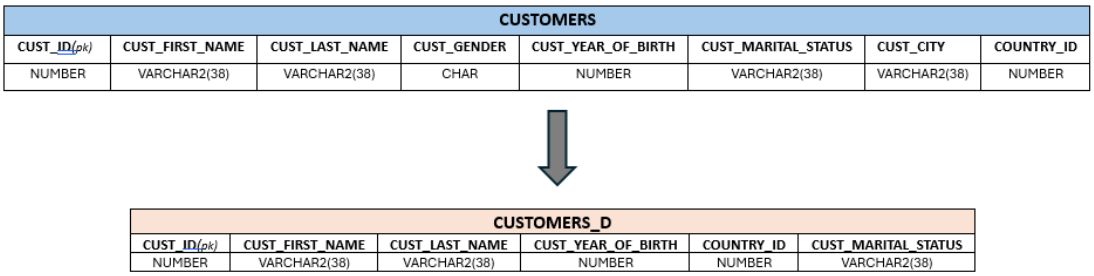
- CUST_CITY
- COUNTRY_ID

You can also simplify the code in the following way:

```
IMPORT SOURCE CUSTOMERS
DEFINE DATASET CUSTOMERS_D FROM CUSTOMERS END
```

Select some columns

You can choose to bring in a subset of columns from the schema of the source or derived dataset.




```
IMPORT SOURCE CUSTOMERS
 DEFINE DATASET CUSTOMERS_D
  ROWSOURCE CUSTOMERS;
  THIS = CUSTOMERS[CUST_ID,CUST_FIRST_NAME,CUST_LAST_NAME,
                   CUST_YEAR_OF_BIRTH,COUNTRY_ID,CUST_MARITAL_STATUS];
 END
```

In this example, only these selected columns from the source table CUSTOMERS form the schema in the target dataset CUSTOMERS_D:

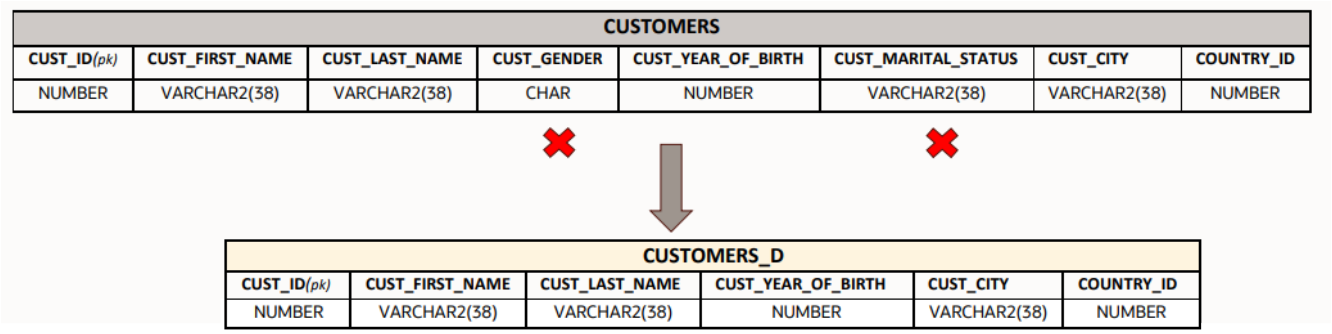
- CUST_ID
- CUST_FIRST_NAME
- CUST_LAST_NAME
- CUST_YEAR_OF_BIRTH
- COUNTRY_ID
- CUST_MARITAL_STATUS

You can also simplify the code in the following way:

```
IMPORT SOURCE CUSTOMERS
DEFINE DATASET CUSTOMERS_D FROM CUSTOMERS[CUST_ID,CUST_FIRST_NAME,
CUST_LAST_NAME,CUST_YEAR_OF_BIRTH,COUNTRY_ID, CUST_MARITAL_STATUS] END
```

Exclude columns

If you want to exclude certain columns from the source table or derived dataset, you can use the EXCLUDE keyword. The remaining columns remain in the target dataset.




```
IMPORT SOURCE CUSTOMERS
 DEFINE DATASET CUSTOMERS_D
   ROWSOURCE CUSTOMERS;
   THIS = CUSTOMERS EXCLUDE [CUST_GENDER,CUST_MARITAL_STATUS];
 END
```

You can also simplify the code in the following way:

```
IMPORT SOURCE CUSTOMERS
DEFINE DATASET CUSTOMERS_D FROM CUSTOMERS EXCLUDE
[CUST_GENDER,CUST_MARITAL_STATUS] END
```

Column Manipulation

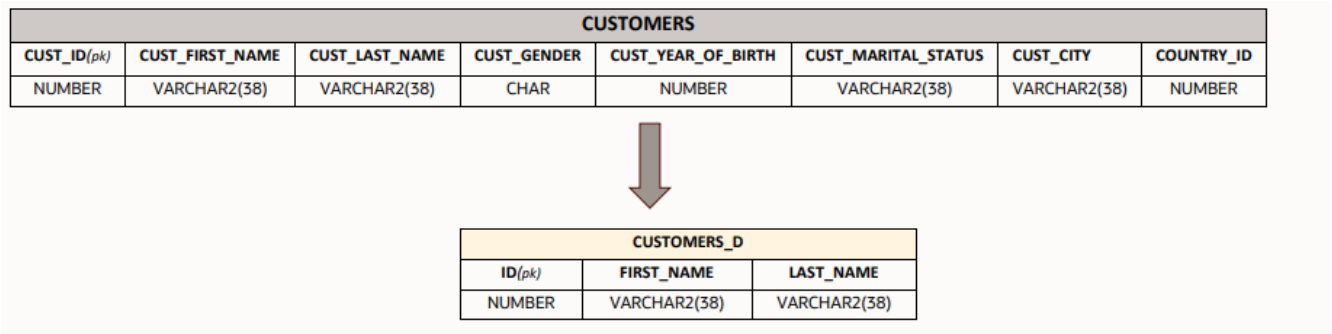
Column manipulation involves applying operations, such as renaming and transformations, to the values of one or more columns to prepare or modify data during processing.

You can manipulate columns in the following ways:

- Rename columns
- Transform Column Data

Rename Columns

You can rename columns during the dataset definition without altering the underlying source data.



Example:

```
IMPORT SOURCE CUSTOMERS

 DEFINE DATASET CUSTOMERS_D
   ROWSOURCE CUSTOMERS;

   //column renaming
   THIS[ID,FIRST_NAME,LAST_NAME] =
CUSTOMERS[CUST_ID,CUST_FIRST_NAME,CUST_LAST_NAME];
 END
```

In this example, you create the columns `ID`,`FIRST_NAME`,`LAST_NAME` in the target table `CUSTOMERS_D` with the same values and column properties as `CUST_ID`, `CUST_FIRST_NAME`,`CUST_LAST_NAME` from the source table `CUSTOMERS`.

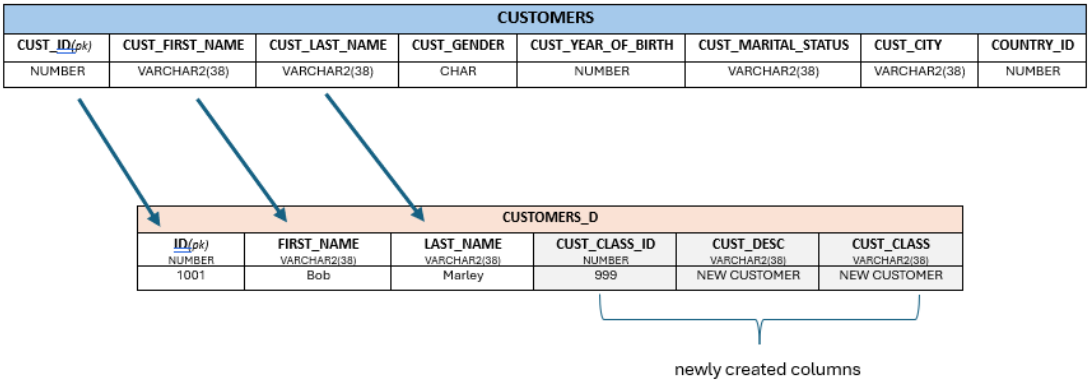
Transform Column Data

You can transform column data in the following ways:

- Assign Static Column Values
- Apply Functions
- Create User-Defined Functions
- Specify the Data Type

Assign Static Column Values

You can directly assign fixed literal values to one or more columns. The values must be from these following types: `NUMBER`, `VARCHAR2`, `DATE`, `Timestamp`.



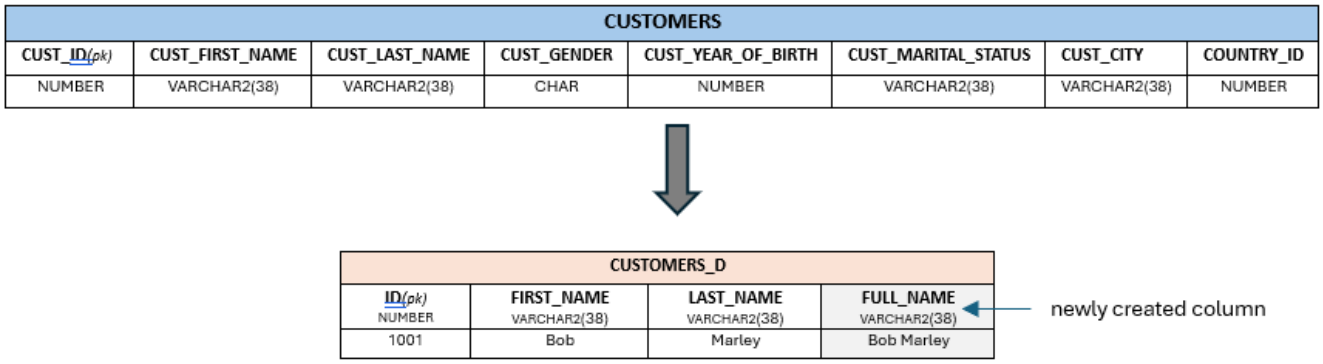
Example:

```
DEFINE DATASET CUSTOMERS_D
 ROWSOURCE CUSTOMERS WHERE CUSTOMERS.CUST_JOINING_DATE = '2025/02/24';
 THIS[ID,FIRST_NAME,LAST_NAME]=
CUSTOMERS[CUST_ID,CUST_FIRST_NAME,CUST_LAST_NAME];
 THIS[CUST_CLASS_ID] = 999; //NUMBER
 THIS[CUST_DESC,CUST_CLASS] = 'NEW CUSTOMER'; //VARCHAR2(38)
END
```

In this example, you've assigned the column `CUST_CLASS_ID` to hold the static numeric value `999` and columns `CUST_DESC`,`CUST_CLASS` to hold the `varchar2` value `NEW CUSTOMER`.

Apply Functions

Data Augmentation Scripts supports a wide range of generic functions for data manipulation. When you apply one or more functions on the right-hand side (RHS) of a column mapping, the resulting data populates the target column. Data Augmentation Scripts allows this, provided that the data returned by the functions matches the expected format, such as a scalar, list, or specific datatype.



Example:

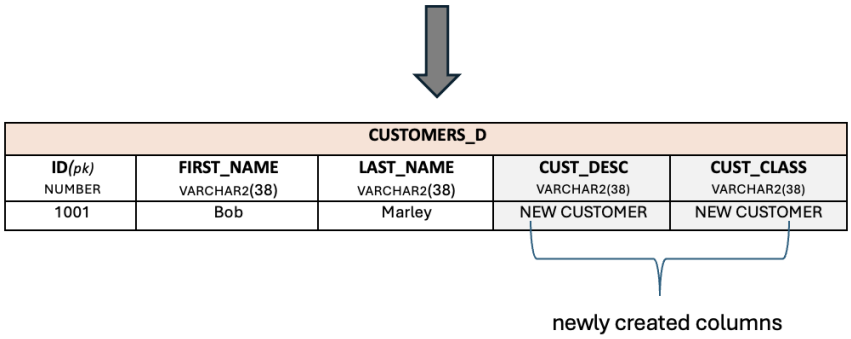
```
DEFINE DATASET CUSTOMERS_D
   ROWSOURCE CUSTOMERS;
   THIS[ID,FIRST_NAME,LAST_NAME] =
CUSTOMERS[CUST_ID,CUST_FIRST_NAME,CUST_LAST_NAME];
   THIS[FULL_NAME] = CONCAT_WS(' ',
CUSTOMERS.CUST_FIRST_NAME,CUSTOMERS.CUST_LAST_NAME);
 END
```

In this example, you apply the `CONCAT_WS` function on the columns `CUST_FIRST_NAME` and `CUST_LAST_NAME` to create the column `FULL_NAME`.

Create User-Defined Functions

You can create reusable user-defined functions (UDFs) that encapsulate specific logic, which you can then apply to a single column or across multiple columns in your dataset.

CUSTOMERS							
CUST_ID(pk) NUMBER	CUST_FIRST_NAME VARCHAR2(38)	CUST_LAST_NAME VARCHAR2(38)	CUST_GENDER CHAR	CUST_YEAR_OF_BIRTH NUMBER	CUST_MARITAL_STATUS VARCHAR2(38)	CUST_CITY VARCHAR2(38)	COUNTRY_ID NUMBER
1001	Bob	Marley	M	1890	NA	SF	2321



CUSTOMERS_D				
ID(pk) NUMBER	FIRST_NAME VARCHAR2(38)	LAST_NAME VARCHAR2(38)	CUST_DESC VARCHAR2(38)	CUST_CLASS VARCHAR2(38)
1001	Bob	Marley	NEW CUSTOMER	NEW CUSTOMER

There are two ways you can use a user-defined function:

- Define the function within the main Data Augmentation Scripts program.
  Example:

```
DEFINE FUNCTION CUSTOMERTYPE(col)
    CASE WHEN col > '2025/02/24' THEN 'NEW CUSTOMER' ELSE 'OLD CUSTOMER'
END;
 END
 DEFINE DATASET CUSTOMERS_D
   ROWSOURCE CUSTOMERS;
   THIS[ID,FIRST_NAME,LAST_NAME] =
CUSTOMERS[CUST_ID,CUST_FIRST_NAME,CUST_LAST_NAME];
   THIS[CUST_DESC,CUST_CLASS] = CUSTOMERTYPE(x) FOR x IN
CUSTOMERS[CUST_JOINING_DATE,CUST_EFF_FROM]
                                -DATATYPE VARCHAR2;
 END
```

- Define the function in a `.func` file within the project directory, which you can then use within the main Data Augmentation Scripts program with the `INCLUDE` keyword.
  Example:

my_function.func

```
DEFINE FUNCTION CUSTOMERTYPE(col)
    CASE WHEN col > '24/02/2025'
         THEN 'NEW CUSTOMER'
    ELSE 'OLD CUSTOMER' END;
END
```

main DAS program

```
INCLUDE FUNCTION 'my_function.func'

DEFINE DATASET CUSTOMERS_D
  ROWSOURCE CUSTOMERS;
  THIS[ID,FIRST_NAME,LAST_NAME] = CUSTOMERS[CUST_ID,CUST_FIRST_NAME,
                                                    CUST_LAST_NAME];
   THIS[CUST_DESC,CUST_CLASS] = CUSTOMERTYPE(x) FOR x IN
   CUSTOMERS[CUST_JOINING_DATE,CUST_EFF_FROM] -DATATYPE VARCHAR2(38);
END
```

Specify the Data Type

Data Augmentation Scripts can infer the data type. You must include a datatype specification (`-DATATYPE`) for the new column when the datatype can't be inferred or needs to be intentionally overridden.

- Create a new column and assign a datatype value.

```
DEFINE DATASET MY_SALES
   ROWSOURCE SALES;
   THIS[PROD_VERSION] = 101 -DATATYPE NUMBER;
END
```

  This example creates a column `PROD_VERSION` of the datatype `NUMBER`.

- Create a new column when the datatype can't be inferred.

```
DEFINE DATASET MY_SALES
  ROWSOURCE SALES;
  THIS[INFO] =CONCAT(SALES.PROD_NAME, ' - ',
              CAST(SALES.SALES_AMT AS VARCHAR2(28))) -DATATYPE
VARCHAR2(50);
END
```

  In this example, you perform the `CONCAT` function on columns with two different data types: `PROD_NAME` (`VARCHAR2(38)`) and `SALES_AMT` (`NUMBER`). You explicitly define the resulting column `INFO` as datatype `VARCHAR2(50)`.

- Create a new column and override the datatype.

```
DEFINE DATASET MY_SALES
   ROWSOURCE SALES;
   THIS[TOTAL_SALES] = SUM(SALES.SALE_AMOUNT) -DATATYPE NUMBER(10,2);
END
```

  In this example, the `SALE_AMOUNT` column is of datatype `NUMBER(10)`, but you specify that the `TOTAL_SALES` value is stored with a specific precision as `NUMBER(10,2)`.

Table Types

You define table types primarily to manage data changes appropriately, based on the business needs, and to keep the source data and target data warehouse in sync for accurate insights.

Data Augmentation Scripts supports these two main types of tables for refreshing data:

- **Updated tables**: Adds new records and updates modified data in the target table.
  Key characteristics of updated tables:
  - Default table type, if you don't specify one.
  - Retains deleted records in the Autonomous Data Warehouse.
  - Useful for managing large datasets that require current data and want to retain deleted data.
- **Versioned tables**: Truncates the target table during each incremental run and reinserts all the data from the source dataset.
  The key characteristic of versioned tables is that it's useful for smaller datasets that require current data but don't need to retain deleted data.

For detailed information on data refresh, see Incremental.

Export Specifications

Oracle Fusion Data Intelligence uses the Autonomous Data Warehouse as its default data warehouse. Using export specification, Data Augmentation Scripts provides a way to control the visibility of the datasets in Autonomous Data Warehouse.

Export specification is essential in ensuring that only the necessary datasets are exposed to the Autonomous Data Warehouse. Limiting access to certain data or functionality enables you to protect critical parts of the system from unintended use or modification, reducing the risk of errors or security vulnerabilities. This separation helps maintain system integrity, enhances modularity, and makes it easier to manage, test, and update the code over time without disrupting the broader system.

Data Augmentation Scripts provides the following export specifications:

- **PRIVATE**: Typically used for creating stage or intermediate tables. These tables aren't exported to the Autonomous Data Warehouse.
  A PRIVATE VERSIONED dataset stores data temporarily for preprocessing or transformation before updating a permanent dataset.

```
DEFINE PRIVATE VERSIONED DATASET TEMP_DAILY_SALES
    ROWSOURCE SALES;
    // Temporary calculations
    THIS[PRODUCT_ID]       = SALES.PROD_ID;
    THIS[SALES_AMOUNT]     = SALES.SALES_AMOUNT;
    THIS[DISCOUNTED_SALES] = THIS[SALES_AMOUNT] - (THIS[SALES_AMOUNT]*
0.10);

END

DEFINE VERSIONED DATASET DW_MONTHLY_SALES
     ROWSOURCE TEMP_DAILY_SALES;

     // Aggregate temporary data into a permanent dataset
      THIS[PRODUCT_ID]       = TEMP_DAILY_SALES.PRODUCT_ID;
      THIS[TOTAL_SALES]     = SUM(TEMP_DAILY_SALES.DISCOUNTED_SALES);
     GROUPBY[PRODUCT_ID];
     PRIMARYKEY[PRODUCT_ID];

END
```

  The `TEMP_DAILY_SALES` dataset performs temporary calculations on `SALES`, where `PRODUCT_ID` isn't unique. This dataset has no primary key and is still allowed. No Primary Key declaration is required. It's then used in `DW_MONTHLY_SALES` to aggregate the data into a permanent dataset with a `PRIMARYKEY` on `PRODUCT_ID`.
- **PROTECTED**: Typically used for internal housekeeping or backing up a VIEW dataset. These datasets are exported to the Oracle Autonomous Data Warehouse, but they aren't visible to end users.
- **PUBLIC**: Default export specification for a dataset. These datasets are exported in the Autonomous Data Warehouse and are visible to business users for building insights.

Supported Files

Data Augmentation Scripts supports a modular file system, enabling you to create multiple file types, such as `.param`, `.func`, multiple Data Augmentation Scripts program files (`.hrf`), and so on.

You can include these files in the Data Augmentation Scripts program by using the `INCLUDE` keyword.

Example:

```
//utility is a .func file with naming convention utility.func
 INCLUDE FUNCTION “utility.func”

 // ConfigParam is a .param file with naming convention ConfigParam.param
 INCLUDE PARAMETER “ConfigParam.param”
```

These files serve distinct purposes, as explained in the following topics:

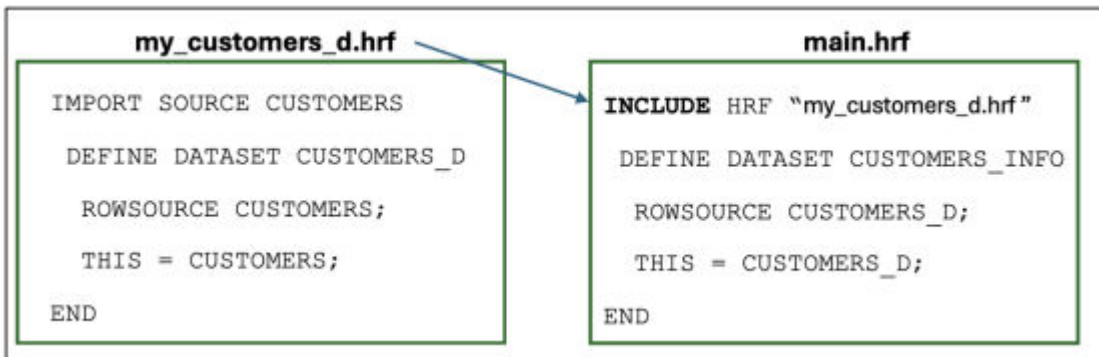
- Data Augmentation Scripts Program Files
- Source Definition Files
- Parameter Definition Files
- Module File
- Function Files
- Conf File
- Query Files

Data Augmentation Scripts Program Files

A Data Augmentation Scripts program file is a file that contains the extract, transform and load logic for the Data Augmentation Scripts program.

Data Augmentation Scripts files have an `.hrf` extension.

You can create multiple `.hrf` files, as shown in this example:



Source Definition Files

A source definition file is a file that contains one or more source definitions.

Source definition files have an `.src` extension.

You can create multiple `.src` files.

To create a source definition file, you can specify the source type and primary key, as shown in this example:

```
IMPORT UPDATEABLE SOURCE CHANNELS WITH PRIMARYKEY[CHANNEL_ID]
IMPORT VERSIONED SOURCE CUSTOMERS WITH PRIMARYKEY[CUSTOMER_ID]
FILTEREDBY(CUST_VALID = 'D') AS CUST
```

`UPDATEABLE` is the default when the `Last Updated Date` (LUD) is in the source table.

Parameter Definition Files

A parameter definition file is a file that contains one or more parameter definitions.

Parameter definition files have a `.param` extension.

You can create multiple `.param` files.

Parameter input values act as configurable constants that influence the logic of the Data Augmentation Scripts program. They provide flexibility by enabling you to adjust key values without changing the main logic of the program, as shown in this example:

```
DEFINE PARAMETER PARAM_SEGMENT_A_CHAR, VARCHAR2(20), "Segment A" END
DEFINE PARAMETER PARAM_SEGMENT_B_CHAR, VARCHAR2(20), "Segment B" END

DEFINE PRIVATE DATASET SALES_SEGMENT_F
 ROWSOURCE SO;
 THIS = SO.CUST_ID;
 THIS[TOTAL_AMT_SOLD] = SUM( SO.AMOUNT_SOLD) ;
 GROUPBY[CUST_ID];
END

DEFINE DATASET CUSTOMERS_SALES_SEGMENT_F
ROWSOURCE CUST INNER JOIN SALES_SEGMENT_F ON (CUST.CUST_ID = SSEG.CUST_ID);
THIS = CUST;
THIS[SALES_SEGMENT] = CASE WHEN SSEG.TOTAL_AMT_SOLD > 10000
                        THEN PARAMETER[PARAM_SEGMENT_A_CHAR]
                        ELSE PARAMETER[PARAM_SEGMENT_B_CHAR]
                      END;
PRIMARYKEY[CUST_ID];
END
```

Module File

A module file is a single read-only file that contains the Data Augmentation Scripts application definition.

The module file has a `.mod` extension.

This file is auto-generated when you create a Data Augmentation Scripts application.

Example:

```
MODULE TIME
SOURCETYPE FUSION
NAMESPACE TIME_
PREFIX DW_FA_X_
```

All the target datasets generated by the Data Augmentation Scripts program contain the prefix `DW_X`, where source is `x`. Example: If source is Fusion, then the source is Fusion applications, and the prefix is `DW_FA_X_`.

Function Files

A function file is a file that contains a list of user-defined functions.

Function files have a `.func` extension.

You can create multiple `.func` files.

Example:

```
DEFINE FUNCTION dateToInt(col) INT(DATE_FORMAT(col,'yyyyMMdd')) END
```

For more information, see Create User-Defined Functions.

Conf File

A `.conf` file is a single `ADW.conf` file that contains constructs such as indexes, column groups, and partitions for PROTECTED and PUBLIC datasets.

You can use `.conf` files to improve performance of reporting queries in the database.

The file has a `.conf` extension.

A `.conf` file contains:

- **Column groups**: A set of columns in a single dataset that's treated as a single unit for query performance.
  Example:

  ```
  COLUMNGROUPS
  [
     CREATE COLUMNGROUP COL_SALES1 ON SALES[CUST_ID, SALE_ID];
     CREATE COLUMNGROUP COL_CUSTOMER ON CUSTOMER[CUST_ID, CUST_NAME];
  ]
  ```

- **Indexes**: A quick lookup of data in a column or columns of a table using B-Tree indexing.
  Example:

  ```
  INDEXES
  [
      CREATE NON-UNIQUE INDEX city_index ON CUST_D[CUST_CITY, COUNTRY_ID];
      CREATE UNIQUE INDEX unique_city_indx ON CUST_D[CUST_ID,
  ```

```
CUST_FIRST_NAME];
]
```

- **Partitions**: A partition allows tables, indexes, and index-organized tables to be subdivided into smaller pieces for managing and accessing them at a finer level of granularity. A table can have only one partition.
  Example:

```
PARTITIONS
[
    // LIST
    CREATE LIST PARTITION ON PRODUCT_SALES[PROD_CATEGORY];
]
```

Query Files

A query file is a file that contains view definitions using SQL language, as an alternative to the standard dataset definition approach: `DEFINE VIEW DATASET`.

Query files have a `.qry` extension.

You can create multiple `.qry` files.

Example: To perform a `channels.qry`, enter:

```
SELECT

CHANNELS_D.CHANNEL_CLASS,CHANNELS_D.CHANNEL_CLASS_ID,CHANNELS_D.CHANNEL_DESC,

CHANNELS_D.CHANNEL_ID,TRUNC(SYSDATE) AS CURDATE

FROM DW_LOCODE_X_APP_CHANNELS_D CHANNELS_D
```

Additional Features

Let's look at these additional features that Data Augmentation Scripts supports.

Topics:

DefaultRow

You can define a default or fallback row in a dataset by using the `DEFAULTROW` feature when a foreign key is missing or invalid.

Defining a default or fallback row serves as a catch-all during joins, aggregations, and data quality checks.

```
IMPORT SOURCE CHANNELS
DEFINE DATASET CHANNELS_D
```

```
 ROWSOURCE CHANNELS;
 THIS = CHANNELS[CHANNEL_ID, CHANNEL_DESC, CHANNEL_CLASS,CHANNEL_CLASS_ID];
 DEFAULTROW
 [
    THIS[CHANNEL_ID,CHANNEL_CLASS_ID] = 999;
    THIS[CHANNEL_DESC,CHANNEL_CLASS] = 'NO CHANNEL';
 ]
END
```

In this example, a synthetic row with the constant values of `999` for the columns `CHANNEL_ID,CHANNEL_CLASS_ID`, and `"NO CHANNEL"` for the columns `CHANNEL_DESC,CHANNEL_CLASS` is inserted using `DEFAULTROW`.

Time Dimensions

Data Augmentation Scripts provides built-in time dimensions for you to extract information or directly incorporate these dimensions.

You can use these built-in time dimensions by importing the system module which contains dimensions, such as Day, Week, Month, Quarter, and Year.

Example:

```
IMPORT MODULE SYSTEM

DEFINE DATASET MY_DAY FROM DW_APPS_DAY_D END
```

The Data Dimension Language (DDL) of the ready-to-use system time dimensions are shown in the following sections:

- DW_APPS_DAY_D
- DW_APPS_WEEK_D
- DW_APPS_MONTH_D
- DW_APPS_QUARTER_D
- DW_APPS_YEAR_D

DW_APPS_DAY_D

The following table shows the data available for the DW_APPS_DAY_D time dimension:

Name	Label	DataType	Primary Key
CALENDAR_DATE	Calendar date.	DATE	
CAL_DAY_ID	Date in YYYYMMDD format.	NUMBER	Y
CAL_HALF_NUMBER	Calendar half-year of this day. Possible values are 1 and 2.	NUMBER	
CAL_MONTH_CODE	Calendar month period. For example, 1979/12.	VARCHAR2	
CAL_MONTH_END_DATE	End date of the month.	DATE	
CAL_MONTH_END_DATE_ID	Last day of month in YYYYMMDD format.	NUMBER	
CAL_MONTH_ID	Month identifier in YYYYQMM format. For example, 1979412.	NUMBER	
CAL_MONTH_LOCALE_NAME	Calendar month name of this day. Possible values are January, February	VARCHAR2	
CAL_MONTH_NUMBER	Calendar month of this day. Possible values are 1 through 12.	NUMBER	
CAL_MONTH_START_DATE	Start date of the month.	DATE	
CAL_MONTH_START_DATE_ID	First day of month in YYYYMMDD format.	NUMBER	
CAL_QUARTER_CODE	Calendar quarter period. For example, 1979 Q4.	VARCHAR2	
CAL_QUARTER_END_DATE	End date of the quarter.	DATE	
CAL_QUARTER_END_DATE_ID	Last day of quarter in YYYYMMDD format.	NUMBER	
CAL_QUARTER_ID	Quarter identifier in YYYYQ format. For example, 19794.	NUMBER	
CAL_QUARTER_NUMBER	Calendar quarter of this day. Possible values are 1, 2, 3, and 4.	NUMBER	
CAL_QUARTER_START_DATE	Start date of the quarter.	DATE	
CAL_QUARTER_START_DATE_ID	First day of quarter in YYYYMMDD format.	NUMBER	
CAL_TRIMESTER_NUMBER	Calendar trimester of this day. Possible values are 1, 2 and 3.	NUMBER	
CAL_WEEK_CODE	Calendar week period. For example, 1979 Week53.	VARCHAR2	
CAL_WEEK_END_DATE	End date of the week.	DATE	
CAL_WEEK_END_DATE_ID	Last day of week in YYYYMMDD format.	NUMBER	
CAL_WEEK_ID	Week identifier in YYYYW format. For example, 197949.	NUMBER	
CAL_WEEK_NUMBER	Calendar week identifier. Possible values are 1 through 53.	NUMBER	
CAL_WEEK_START_DATE	Start date of the week.	DATE	
CAL_WEEK_START_DATE_ID	First day of week in YYYYMMDD format.	NUMBER	
CAL_YEAR_CODE	Calendar Year period. For example, 1979	VARCHAR2	
CAL_YEAR_END_DATE	End date of the year.	DATE	
CAL_YEAR_END_DATE_ID	Last day of year in YYYYMMDD format.	NUMBER	
CAL_YEAR_ID	Year identifier in YYYY format. For example, 1979.	NUMBER	
CAL_YEAR_START_DATE	Start date of the year.	DATE	
CAL_YEAR_START_DATE_ID	First day of year in YYYYMMDD format.	NUMBER	
DAY_AGO_DATE	Previous day's date.	DATE	
DAY_AGO_ID	Previous date in YYYYMMDD format.	NUMBER	
DAY_CODE	Name of the day. Possible values are SUN, MON, and so on.	VARCHAR2	
DAY_LOCALE_NAME	Full Name of the day. Possible values are Sunday, Monday..etc	VARCHAR2	
DAY_OF_MONTH	Day of the month. Possible values are 1 through 31.	NUMBER	
DAY_OF_WEEK	Day of the week. Possible values are 1 through 7.	NUMBER	
DAY_OF_YEAR	Day of the year. Possible values are 1 through 366.	NUMBER	
FIRST_DAY_CAL_MONTH_FLAG	Indicates that this day is the first day of the calendar month.	VARCHAR2	
FIRST_DAY_CAL_QTR_FLAG	Indicates that this day is the first day of the calendar quarter.	VARCHAR2	
FIRST_DAY_CAL_WEEK_FLAG	Indicates that this day is the first day of the calendar week.	VARCHAR2	
FIRST_DAY_CAL_YEAR_FLAG	Indicates that this day is the first day of the calendar year.	VARCHAR2	
JULIAN_DAY_NUMBER	Date in Julian format.	NUMBER	
LAST_DAY_CAL_MONTH_FLAG	Indicates that this day is the last day of the calendar month.	VARCHAR2	
LAST_DAY_CAL_QTR_FLAG	Indicates that this day is the last day of the calendar quarter.	VARCHAR2	
LAST_DAY_CAL_WEEK_FLAG	Indicates that this day is the last day of the calendar week.	VARCHAR2	
LAST_DAY_CAL_YEAR_FLAG	Indicates that this day is the last day of the calendar year.	VARCHAR2	

DW_APPS_WEEK_D

The following table shows the data available for the DW_APPS_WEEK_D time dimension:

Name	Label	DataType	Primary Key
CAL_WEEK_CODE	Calendar week period Name. For example, 1979 Week53.	VARCHAR2	
CAL_WEEK_END_DATE	End date of the week.	DATE	
CAL_WEEK_END_DATE_ID	Last day of week in YYYYMMDD format.	NUMBER	
CAL_WEEK_ID	Week identifier in YYYYW format. For example, 197949.	NUMBER	Y
CAL_WEEK_NUMBER	Calendar week identifier. Possible values are 1 through 53.	NUMBER	
CAL_WEEK_START_DATE	Start date of the week.	DATE	
CAL_WEEK_START_DATE_ID	First day of week in YYYYMMDD format.	NUMBER	
CAL_YEAR_CODE	Calendar year period Name. For example, 1979	VARCHAR2	
CAL_YEAR_END_DATE	End date of the year.	DATE	
CAL_YEAR_END_DATE_ID	Last day of year in YYYYMMDD format.	NUMBER	
CAL_YEAR_ID	Year identifier in YYYY format. For example, 1979.	NUMBER	
CAL_YEAR_START_DATE	Start date of the year.	DATE	
CAL_YEAR_START_DATE_ID	First day of year in YYYYMMDD format.	NUMBER	
FIRST_WEEK_CAL_YEAR_FLAG	Indicates that this week is the first week of calendar year.	VARCHAR2	
LAST_WEEK_CAL_YEAR_FLAG	Indicates that this week is the last week of calendar year.	VARCHAR2	

DW_APPS_MONTH_D

The following table shows the data available for the DW_APPS_MONTH_D time dimension:

Name	Label	DataType	Primary Key
CAL_HALF_NUMBER	Calendar half-year of this day. Possible values are 1 and 2.	NUMBER	
CAL_MONTH_CODE	Calendar month period. For example, 1979/12.	VARCHAR2	
CAL_MONTH_END_DATE	End date of the month.	DATE	
CAL_MONTH_END_DATE_ID	Last day of month in YYYYMMDD format.	NUMBER	
CAL_MONTH_ID	Month identifier in YYYYQMM format. For example, 1979412.	NUMBER	Y
CAL_MONTH_LOCALE_NAME	Calendar month name of this day. Possible values are January, February	VARCHAR2	
CAL_MONTH_NUMBER	Calendar month of this day. Possible values are 1 through 12.	NUMBER	
CAL_MONTH_START_DATE	Start date of the month.	DATE	
CAL_MONTH_START_DATE_ID	First day of month in YYYYMMDD format.	NUMBER	
CAL_QUARTER_CODE	Calendar quarter period. For example, Q4 in 1979 Q4.	VARCHAR2	
CAL_QUARTER_END_DATE	End date of the quarter.	DATE	
CAL_QUARTER_END_DATE_ID	Last day of quarter in YYYYMMDD format.	NUMBER	
CAL_QUARTER_ID	Quarter identifier in YYYYQ format. For example, 19794.	NUMBER	
CAL_QUARTER_NUMBER	Calendar quarter of this day. Possible values are 1, 2, 3, and 4.	NUMBER	
CAL_QUARTER_START_DATE	Start date of the quarter.	DATE	
CAL_QUARTER_START_DATE_ID	First day of quarter in YYYYMMDD format.	NUMBER	
CAL_TRIMESTER_NUMBER	Calendar trimester of this day. Possible values are 1, 2 and 3.	NUMBER	
CAL_YEAR_CODE	Calendar year period. For example, 1979	VARCHAR2	
CAL_YEAR_END_DATE	End date of the year.	DATE	
CAL_YEAR_END_DATE_ID	Last day of year in YYYYMMDD format.	NUMBER	
CAL_YEAR_ID	Year identifier in YYYY format. For example, 1979.	NUMBER	
CAL_YEAR_START_DATE	Start date of the year.	DATE	
CAL_YEAR_START_DATE_ID	First day of year in YYYYMMDD format.	NUMBER	
FIRST_MONTH_CAL_QTR_FLAG	Indicates that this month is the first month of calendar quarter.	VARCHAR2	
FIRST_MONTH_CAL_YEAR_FLAG	Indicates that this month is the first month of calendar year.	VARCHAR2	
LAST_MONTH_CAL_QTR_FLAG	Indicates that this month is the last month of calendar quarter.	VARCHAR2	
LAST_MONTH_CAL_YEAR_FLAG	Indicates that this month is the last month of calendar year.	VARCHAR2	

DW_APPS_QUARTER_D

The following table shows the data available for the DW_APPS_QUARTER_D time dimension:

Name	Label	DataType	Primary Key
CAL_HALF_NUMBER	Calendar half-year of this quarter. Possible values are 1 and 2.	NUMBER	
CAL_QUARTER_CODE	Calendar quarter period. For example, 1979 Q4.	VARCHAR2	
CAL_QUARTER_END_DATE	End date of the quarter.	DATE	
CAL_QUARTER_END_DATE_ID	Last day of quarter in YYYYMMDD format.	NUMBER	
CAL_QUARTER_ID	Quarter identifier in YYYYQ format. For example, 19794.	NUMBER	Y
CAL_QUARTER_NUMBER	Calendar quarter. Possible values are 1, 2, 3, and 4.	NUMBER	
CAL_QUARTER_START_DATE	Start date of the quarter.	DATE	
CAL_QUARTER_START_DATE_ID	First day of quarter in YYYYMMDD format.	NUMBER	
CAL_YEAR_CODE	Calendar year period. For example, 1979	VARCHAR2	
CAL_YEAR_END_DATE	End date of the year.	DATE	
CAL_YEAR_END_DATE_ID	Last day of year in YYYYMMDD format.	NUMBER	
CAL_YEAR_ID	Year identifier in YYYY format. For example, 1979.	NUMBER	
CAL_YEAR_START_DATE	Start date of the year.	DATE	
CAL_YEAR_START_DATE_ID	First day of year in YYYYMMDD format.	NUMBER	
FIRST_QUARTER_CAL_YEAR_FLAG	Indicates that this quarter is the first quarter of calendar year.	VARCHAR2	
LAST_QUARTER_CAL_YEAR_FLAG	Indicates that this quarter is the last quarter of calendar year.	VARCHAR2	

DW_APPS_YEAR_D

The following table shows the data available for the DW_APPS_YEAR_D time dimension:

Name	Label	DataType	Primary Key
CAL_YEAR_CODE	Calendar year period. For example, 1979	VARCHAR2	
CAL_YEAR_END_DATE	End date of the year.	DATE	
CAL_YEAR_END_DATE_ID	Last day of year in YYYYMMDD format.	NUMBER	
CAL_YEAR_ID	Year identifier in YYYY format. For example, 1979.	NUMBER	Y
CAL_YEAR_START_DATE	Start date of the year.	DATE	
CAL_YEAR_START_DATE_ID	First day of year in YYYYMMDD format.	NUMBER	

Inline Dataset

You can use an inline dataset to embed a static table of hard-coded values directly in your Data Augmentation Scripts program without having to import a source or warehouse table.

The INLINE table type requires instructions for the table structure and a set of records as input. The table is then generated and loaded with the provided data.

Example:

```
DEFINE INLINE DATASET MYINLINEDATA
    ROWSOURCE INTABLE([PROMO_CATEGORY_ID:NUMBER,PROMO_CATEGORY:VARCHAR2(128)]
                VALUES
                     ([2 ,'NO PROMOTION'],
                      [3 ,'TV'],
                      [4 ,'ad news']
                      )
                     );
    PRIMARYKEY [PROMO_CATEGORY_ID];
 END

 DEFINE DATASET PROMOTION_CATEGORY_D
    ROWSOURCE MYINLINEDATA;
    THIS = MYINLINEDATA;
 END
```

The output is shown in the PROMOTION_CATEGORY_D table:

PROMO_CATEGORY_ID (NUMBER) PK	PROMO_CATEGORY (VARCHAR2)
2	NO_PROMOTION
3	TV
4	ad news

Advanced Data Augmentation Scripts Features

Data Augmentation Scripts offers advanced features for data transformations.

Topics:

Incremental

The Data Warehouse must continuously be in sync as the source data changes over time.

The UPDATEABLE and VERSIONED table types provide options for refreshing data.

When you run the first ETL job for a data application with UPDATEABLE or VERSIONED table types, the data in the source system or staging area and the target data warehouse are the same.

For subsequent data refreshes, there are two ways you can keep data in sync between the source system and target data warehouse:

- **Full Refresh**: Copies all the data from the source system to the data warehouse.
- **Incremental Refresh**: Processes only the data that was newly added or modified since the last load from the source system to the data warehouse. Incremental refreshes are preferred because they enable faster and more efficient updates with minimal impact on system resources.

The following diagram illustrates how data changes are handled for VERSIONED and UPDATEABLE table types:

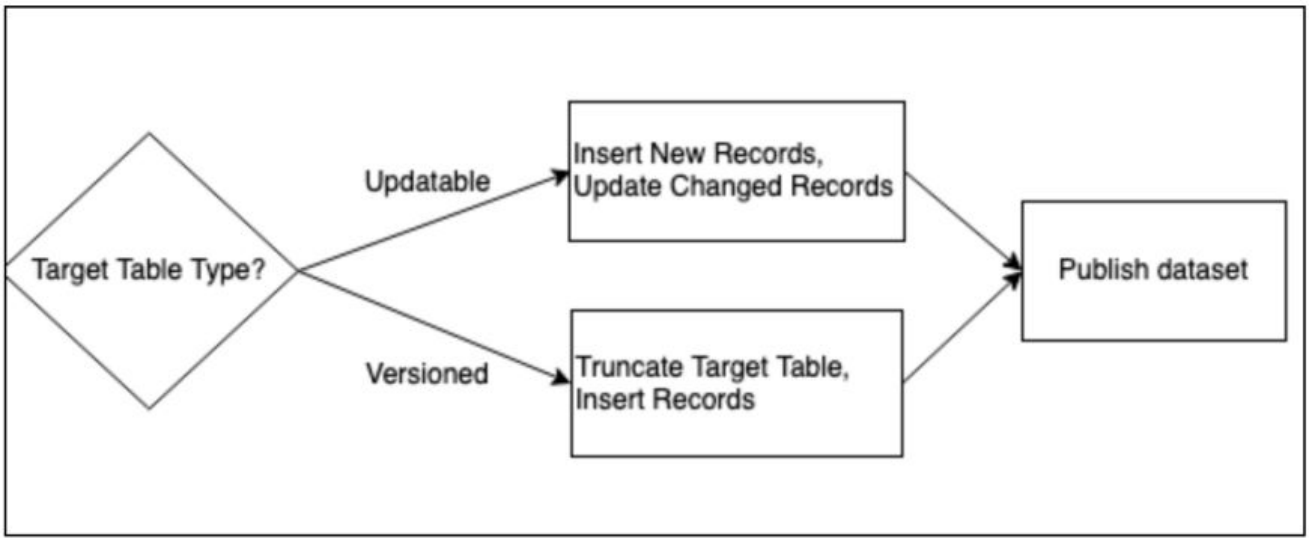


Table Type with Source Dataset

- **VERSIONED**: The system extracts all the data from the source table.
  The data in the mirror copy is truncated and loaded again from the source system. Deleted records aren't retained in the data warehouse.

  Example:

  ```
  IMPORT VERSIONED SOURCE SALES
  ```

  All data from `SALES` is extracted.
- **UPDATEABLE**: The system extracts only the changed records from the source, and updates only the changed data in the mirror copy of the extracted data.
  Deleted records are retained in the data warehouse.

  Example:

  ```
  IMPORT UPDATEABLE SOURCE SALES WITH LUD[LAST_UPDATE_DATE]
  ```

  Only the changed records from `SALES` are extracted.

> **Note**
>
> Deleted records need additional handling because they are no longer in the source and can't be included in the extracted data.

Table Type with Target Dataset

- **VERSIONED**: The system extracts all the data from the source table.
  Deleted records aren't retained in the data warehouse.

Example:

```
IMPORT VERSIONED SOURCE SALES

DEFINE VERSIONED DATASET DW_SALES_FACT FROM SALES END
```

The SALES dataset is fully refreshed even if some records don't have any updates.

- **UPDATEABLE**
  The incremental refresh directive (IRD) identifies which dataset drives changes for the insert or update selection. When you assign a source as the change-driving dataset:
  - ETL processes consider all change records in the driving source and their matching records in other sources.
  - If a non-driving source has changes that do not have matching changes in the driving source, the process ignores these changes during the incremental refresh.

  You must specify the IRD for any dataset that uses two or more sources.

  When defining a target dataset in the Data Augmentation Scripts application, you must decide its incremental data refresh behavior.

  If the table type is UPDATEABLE, you have to nominate the input tables that are driving the changes using the incremental refresh directive (IRD) of REFRESH ON CHANGES in the target dataset.

  ```
  IMPORT UPDATEABLE SOURCE SALES
  DEFINE UPDATEABLE DATASET DW_SALES_FACT FROM SALES END
  ```

  New and updated records in `SALES` are updated in the target dataset `DW_SALES_FACT`.

  This directive within the DEFINE DATASET block handles the complexity of change detection and updating of the target datasets. You don't have to write boilerplate code to detect changes and deal with complex logic for updating target datasets.

  The directive also provides predictability in the incremental refresh behavior:
  - When only one input table is used to create an UPDATEABLE dataset, the <incremental-refresh-directive> and the change driving input table are inferred.
    In the following diagram, the two lines of code illustrate how to create the dataset and define its incremental refresh behavior.

```
IMPORT SOURCE SALES
DEFINE DATASET DW_SALES_F FROM SALES END
```

```
IMPORT UPDATEABLE SOURCE SALES
DEFINE UPDATEABLE DATASET DW_SALES_F
    ROWSOURCE SALES;
    THIS = SALES;
    REFRESH ON CHANGES IN [SALES];
END
```

In this example, only the changed records from Sales are brought in.

– When multiple input tables are used to create an UPDATEABLE dataset, you must explicitly specify which input tables are the change driving tables in the <incremental-refresh-directive>.
Changed records from the driving tables identify the delta and then considers only the corresponding matching records from the non-driving tables. Changes in the non-driving tables by themselves are ignored.

Source table type: UPDATEABLE
Target table type: UPDATEABLE

IRD: Only on one of the input tables

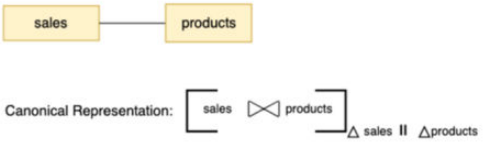
```
IMPORT SOURCE SALES WITH LUD[LAST_UPDATE_DATE]
IMPORT SOURCE PRODUCTS WITH LUD[LAST_UPDATE_DATE]

DEFINE DATASET DW_SALES_F
    ROWSOURCE SALES INNER JOIN PRODUCTS on SALES.PROD_ID = PRODUCTS.PROD_ID;

    THIS = SALES;
    THIS = PRODUCTS EXCLUDE [PROD_ID];

    PRIMARYKEY[SALE_ID];
    REFRESH ON CHANGES IN [SALES];
END
```

In this example, Sales is the change-driving table. Only the changed (△) records from Sales are joined with Products.

Source table type: UPDATEABLE
Target table type: UPDATEABLE

IRD: On all of the input tables

```
IMPORT SOURCE SALES    WITH LUD[LAST_UPDATE_DATE]
IMPORT SOURCE PRODUCTS WITH LUD[LAST_UPDATE_DATE]

DEFINE DATASET DW_SALES_F
    ROWSOURCE SALES INNER JOIN PRODUCTS on SALES.PROD_ID = PRODUCTS.PROD_ID;

    THIS = SALES;
    THIS = PRODUCTS EXCLUDE [PROD_ID];

    PRIMARYKEY[SALE_ID];
    REFRESH ON CHANGES IN [SALES, PRODUCTS];
END
```



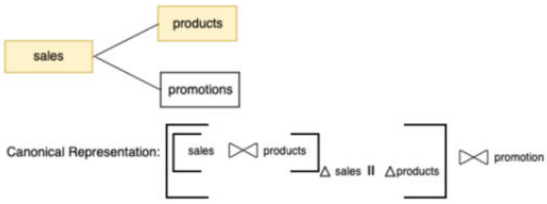
In this example, Sales and Products are both change-driving tables. The changed (△) records from both tables are brought in.

Source table type: UPDATEABLE

Target table type: UPDATEABLE

IRD: On few of the input tables

```
IMPORT SOURCE SALES    WITH LUD[LAST_UPDATE_DATE]
IMPORT SOURCE PRODUCTS WITH LUD[LAST_UPDATE_DATE]
IMPORT SOURCE PROMOTIONS WITH LUD[LAST_UPDATE_DATE]

DEFINE DATASET DW_SALES_F
    ROWSOURCE SALES INNER JOIN PRODUCTS ON SALES.PROD_ID = PRODUCTS.PROD_ID
                    INNER JOIN PROMOTIONS ON SALES.PROMO_ID = PROMOTIONS.PROMO_ID;

    THIS = SALES;
    THIS = PRODUCTS EXCLUDE [PROD_ID];
    THIS = PROMOTIONS EXCLUDE [PROMO_ID];

    PRIMARYKEY[SALE_ID];
    REFRESH ON CHANGES IN [SALES, PRODUCTS];
END
```



In this example, changed (△) records from both change-driving tables Sales and Products are joined with the non-change driving table Promotions.

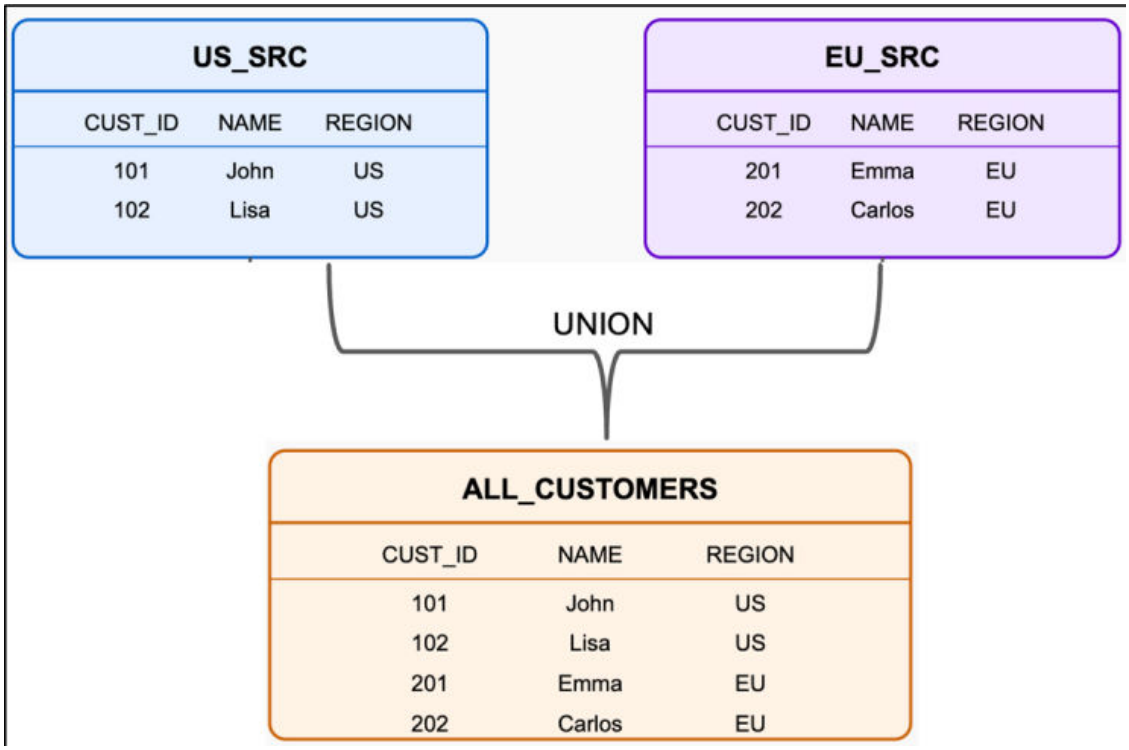
The following table summarizes the behavior of source and target table types for UPDATEABLE and VERSIONED refreshes:

SOURCE TYPE	TARGET TYPE	DESCRIPTION	TYPICAL USAGE
UPDATEABLE	UPDATEABLE	This is the most common and default combination. • New records are inserted and changed records are updated in the target dataset. • The records that got deleted in the source system are retained in the data warehouse.	Transactional facts and dimensions, that should retain deleted records
VERSIONED	VERSIONED IRD not allowed	• Target table is truncated. • All data from the source table is inserted in the target dataset. • The records that got deleted in the source system are not retained in the data warehouse.	Transactional or aggregate tables that should not contain deleted records

Set Operation

A SET dataset forms when you combine two or more input datasets through a set operation. You can also use it as a source for another dataset.

In the following image, a set operation combines customer records from the `US_SRC` and `EU_SRC` tables to create a single `ALL_CUSTOMERS` table that contains customers from both the US and EU sources:



Example:

```
IMPORT SOURCE CUSTOMERS FILTEREDBY (REGION = 'US') AS US_SRC
IMPORT SOURCE CUSTOMERS FILTEREDBY (REGION = 'EU') AS EU_SRC

DEFINE DATASET ALL_CUSTOMERS
 ROWSOURCE UNION[US_CUSTOMERS, EU_CUSTOMERS];
 THIS = US_CUSTOMERS;
 PRIMARYKEY[CUST_ID];
END
```

AGGREGATION ONLY Dataset

`AGGREGATIONONLY` datasets are tables that contain summarized data in a single row.

The following rules are for creating `AGGREGATIONONLY` datasets:

- The dataset must always be `VERSIONED` and have only one row.
- All column assignments must use aggregate functions.
- No Primary Key declaration is required.
- The dataset is treated as a regular dataset and not as aggregation-only dataset if `GROUPBY` is specified.

Follow these rules for using `AGGREGATIONONLY` datasets as input in `ROWSOURCE`:

- Only `CROSS-JOIN` is allowed with `AGGREGATIONONLY` tables in `ROWSOURCE`.
- If a dataset is created using only an AGGREGATIONONLY dataset in ROWSOURCE, then the derived table must also be marked as `AGGREGATIONONLY` dataset.
- `SET` operations aren't supported directly on `AGGREGATIONONLY` datasets.

- `REFRESH ON CHANGES IN` aren't allowed on `AGGREGATIONONLY` tables.

Example:

```
IMPORT SOURCE SALES

// Single column assignment
DEFINE AGGREGATIONONLY DATASET DW_SALES_AGG
 ROWSOURCE SALES;
 THIS[AVG_SALES_AMT] = AVG(SALES[AMOUNT_SOLD]);
END

// Multiple column assignments
DEFINE AGGREGATIONONLY DATASET DW_SALES_AGG1
 ROWSOURCE SALES;

 THIS[AVG_SALES_AMT] = AVG(SALES[AMOUNT_SOLD]);

 THIS[SUM_SALES_AMT] = SUM(SALES[AMOUNT_SOLD]);
 THIS[MIN_SALES_AMT] = MIN(SALES[AMOUNT_SOLD]);
 THIS[MAX_SALES_AMT] = MAX(SALES[AMOUNT_SOLD]);
END

// Derived from another AGGREGATIONONLY dataset
DEFINE AGGREGATIONONLY DATASET DW_SALES_AGG2
 ROWSOURCE DW_SALES_AGG2;
 THIS = DW_SALES_AGG2 [AVG_SALES_AMT];;
END
```

The following output is derived:

Output:

Input Source table:

SALES		
SALE_ID	**PRODUCT**	**AMOUNT_SOLD**
1001	Laptop	1200.0
1002	Monitor	350.0
1003	Mouse	25.0
1004	Keyboard	75.0

Target tables:

DW_SALES_AGG
AVG_SALES_AMT
412.5

DW_SALES_AGG1			
AVG_SALES_AMT	**SUM_SALES_AMT**	**MIN_SALES_AMT**	**MAX_SALES_AMT**
412.5	1650.0	25.0	1200.0

`DW_SALES_AGG2` is the same as `DW_SALES_AGG1`.

Transposition Table Definition

You use the transposition table definition to restructure datasets by rotating rows into columns (PIVOT) or columns into rows (UNPIVOT).

Topics:

Pivot

Pivoting transforms data by converting rows into columns, enabling you to compare values side-by-side, spot trends efficiently, and gain clear insights across categories such as months and channels.

Example: You can create a versioned dataset `SALES_F` using the following command:

```
IMPORT SOURCE SALES
DEFINE VERSIONED DATASET SALES_F
 ROWSOURCE SALES;
 THIS = SALES[PROD_ID, CHANNEL_ID];
 THIS[TIME_ID] = DATE_FORMAT(SALES.TIME_ID,'MMM');
 THIS[AMOUNT_SOLD] = SUM(SALES.AMOUNT_SOLD);
 GROUPBY [PROD_ID, CHANNEL_ID,TIME_ID];
 PRIMARYKEY[PROD_ID, CHANNEL_ID,TIME_ID];
END
```

The output of `SALES_F` is as follows:

PROD_ID	CHANNEL_ID	TIME_ID	AMOUNT_SOLD
1	Online	Jan	500
1	InStore	Jan	150
1	InStore	Mar	700
2	Online	Jan	200
2	InStore	Feb	120
2	InStore	Mar	300

Single-Column Partitioned Pivot

You can use partitioning in pivot operations to group data by specific attributes, keeping each category, such as product or channel, distinct within the transformed dataset.

If you don't specify partition, all the columns in ROWSOURCE are included for partitioning, except those that you use in transpositions.

Example:

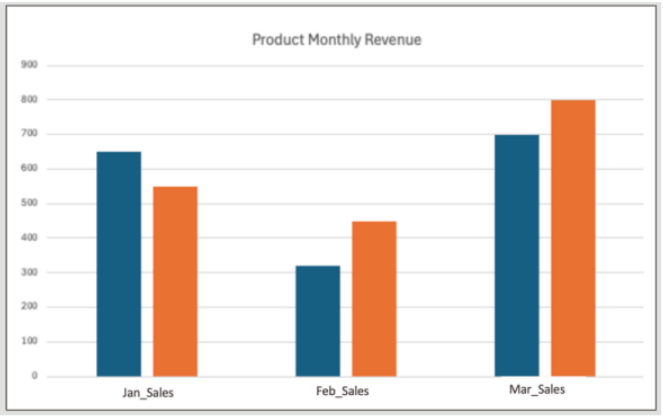
```
DEFINE VERSIONED DATASET PRODUCT_MONTHLY_REVENUE
 ROWSOURCE SALES_F;

 PIVOT
 (
```

```
 /* ------------------- SPECIFY PARTITION ------------------------- */

 WITHIN SALES_F[PROD_ID];

 /* --------------------SPECIFY TRANSPOSITIONS ---------------------- */
 // Either provide target column names on LHS for each of the month values
or if unspecified, it will auto-generate and map columns Jan,Feb,Mar

 THIS[Jan_Sales,Feb_Sales,Mar_Sales]= SUM(SALES_F.AMOUNT_SOLD) FOR
 SALES_F.TIME_ID IN('Jan','Feb','Mar');
 );

 // Optional. If not specified, PK will be assigned.
 PRIMARYKEY [PROD_ID];
END
```

In this example, the segment `WITHIN SALES_F[PROD_ID,CHANNEL_ID]` partitions the data by `Product ID` and `Channel ID`. The column mapping `THIS[Jan_Sales, Feb_Sales, Mar_Sales]` creates target columns for the months January, February, and March. The output creates separate columns for sales in January, February, and March.

The output from pivoting the `PRODUCT_MONTHLY_REVENUE` dataset is shown:

PROD_ID	Jan_Sales	Feb_Sales	Mar_Sales
2	550	450	800
1	650	320	700



Multi-Column Partitioned Pivot

You can use multi-column partitioning in pivot operations to group data by multiple attributes, providing users with a more detailed view of the dataset. This method enables analyst users to compare across dimensions, such as product and channel, helping them uncover deeper insights and patterns.

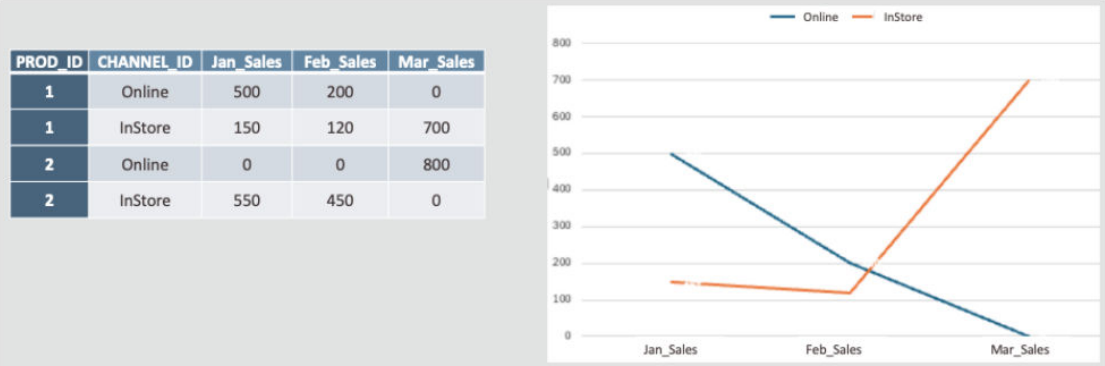
By partitioning on both `Product` and `Channel`, you can enable analysts to track unique revenue contributions for each combination.

Example:

```
DEFINE VERSIONED DATASET PRODUCT_BY_CHANNEL_MONTHLY_REVENUE
ROWSOURCE SALES_F;
 PIVOT
 (
/* ------------------------ SPECIFY PARTITION ------------------------- */
 WITHIN SALES_F[PROD_ID, CHANNEL_ID];
```

```
/* ----------------------SPECIFY TRANSPOSITIONS ---------------------- */
 THIS[Jan_Sales, Feb_Sales, Mar_Sales] = SUM(SALES_F.AMOUNT_SOLD) FOR
 SALES_F.TIME_ID IN ('Jan', 'Feb', 'Mar');
 );
 PRIMARYKEY [PROD_ID,CHANNEL_ID];
END
```

In this example, the segment `WITHIN SALES_F[PROD_ID,CHANNEL_ID]` partitions the data as composite key in multiple columns, Product ID and Channel ID. The column mapping `THIS[Jan_Sales,Feb_Sales,Mar_Sales]` creates target columns for each of the listed transposed values. The output generates separate columns for sales in January, February, and March by `PROD_ID` and `CHANNEL_ID`.



PROD_ID	CHANNEL_ID	Jan_Sales	Feb_Sales	Mar_Sales
1	Online	500	200	0
1	InStore	150	120	700
2	Online	0	0	800
2	InStore	550	450	0

Multi-Dimensional Partitioned Pivot

You can use multi-dimensional partitioning in pivot operations to generate columns based on multiple attributes while aggregating a specific value. This pivot operation reorganizes the data to show the aggregated sales amounts for each category combination, such as `Time` and `Channel`.

Example:

```
DEFINE VERSIONED DATASET PRODUCT_BY_CHANNEL_MONTHLY_REVENUE_2
 ROWSOURCE SALES_F;
 PIVOT
 (
 /* ------------------ SPECIFY PARTITION ------------------------- */
 WITHIN SALES_F[PROD_ID];

 /* --------------------SPECIFY TRANSPOSITIONS ------------------- */
 //Sales per month
 THIS[Jan_Sales, Feb_Sales, Mar_Sales] = SUM(SALES_F.AMOUNT_SOLD) FOR
 SALES_F.TIME_ID IN ('Jan', 'Feb', 'Mar');
 // Sales per month + channel (multi dimensions)
 THIS = [SUM(SALES_F.AMOUNT_SOLD) -COLPREFIX 'Amt'] FOR (SALES_F.CHANNEL_ID,
 SALES_F.TIME_ID )IN ((('Online','Jan'),
 ('Online','Feb'),('Online','Mar'),('InStore','Jan'),('InStore','Feb'),
 ('InStore','Mar'));
 );
```

```
// Other transformations allowed after PIVOT section and only using the
 columns generated in PIVOT section
THIS[Jan_Inst_Online_Diff] = THIS.Amt_InStore_Jan - THIS.Amt_Online_Jan;
THIS[Feb_Inst_Online_Diff] = THIS.Amt_InStore_Feb - THIS.Amt_Online_Feb;
THIS[Feb_Inst_Online_Diff] = THIS.Amt_InStore_Mar - THIS.Amt_Online_Mar;

 PRIMARYKEY [PROD_ID];
END
```

In this example, the output for the dataset `PRODUCT_BY_CHANNEL_MONTHLY_REVENUE_2` is as follows:

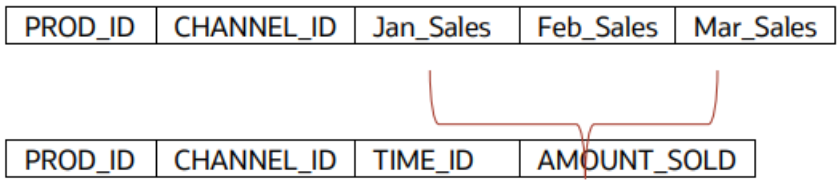
PROD_ID	Jan_Sales	...	Amt_InStore_Jan	Amt_InStore_Feb	Amt_InStore_Mar	Amt_Online_Jan	Amt_Online_Feb	Amt_Online_Mar	...
2	...	...	550	450	0	0	0	800	...
1	...	...	150	120	700	500	200	0	...

Unpivot

You can use the `UNPIVOT` operator to transform columns back into rows, enabling business users to analyze the data in detail.

`UNPIVOT` helps your users to explore each attribute, provides greater flexibly, and reveals patterns and changes over time.

Example: You can `UNPIVOT` `Jan_Sales`, `Feb_Sales`, and `Mar_Sales` into the `AMOUNT_SOLD` column.




```
DEFINE VERSIONED DATASET SALES_F2[
 ROWSOURCE MY_SALES;
 UNPIVOT INCLUDE NULLS
 (
 WITHIN MY_SALES[PROD_ID, CHANNEL_ID];
```

```
 /* Target Columns TIME_ID, AMOUNT_SOLD are specified on LHS.
 The corresponding display values for PRODUCT column are specified in
 the LHS. Pairs on LHS map to columns on RHS, in sequence /*
 THIS[(TIME_ID : 'Jan', AMOUNT_SOLD),(TIME_ID : 'Feb', AMOUNT_SOLD),
 (TIME_ID : 'Mar', AMOUNT_SOLD)]=MY_SALES[Jan_Sales,Feb_Sales,Mar_Sales];
 );

 PRIMARYKEY [PROD_ID, CHANNEL_ID, TIME_D];
END
```

The output of the versioned `SALES_F2` dataset table is as follows:

PROD_ID	CHANNEL_ID	TIME_ID	AMOUNT_SOLD
3	Online	Jan	500
3	InStore	Jan	150
4	Online	Jan	0
4	InStore	Jan	550
3	Online	Feb	2000

Delete Handling

In a data warehouse, the decision to retain or delete records from the data warehouse depends on the specific use cases and business requirements.

You can delete source data as it continually updates with new records. Deletions can be:

- **Hard Delete**: Permanently deletes records from data warehouse datasets, through propagation or explicit application.
- **Soft Delete**: Marks records as deleted without physically removing them from the dataset.

You can handle deletions by using the `TRACKDELETES`, `DELETESOURCE`, and `THEN DELETE` directives, which are designed to manage both hard and soft deletes during data import and processing.

The following topics describe how you can apply deletions:

- TRACKDELETES (Default Behavior)
- DELETESOURCE
- THEN DELETE

For more information about delete handling, see Deletions.

TRACKDELETES (Default Behavior)

You can use the `TRACKDELETES` directive to ensure that deletions are automatically tracked and handled during data extraction or import. It doesn't require separate deletion logs or manual intervention.

Hard Delete

When records are deleted in the source system, the `TRACKDELETES` directive ensures that these deleted records are automatically excluded during data import or processing.

Example:

```
IMPORT SOURCE PRODUCTS WITH TRACKDELETES AS PROD_BASE
IMPORT SOURCE AS SALES

DEFINE DATASET PROD_DIM
 ROWSOURCE PROD_BASE ;
 THIS = PROD_BASE;
END

DEFINE DATASET PROD_REPLENISH
 ROWSOURCE PROD_DIM INNER JOIN SALES ON (PROD_DIM.PROD_ID = SALES.PROD_ID);
 THIS = PROD_DIM[PROD_NAME, PROD_ID];
 THIS[REPLENISH_FLG] = CASE WHEN SUM(SALES.QUANTITY_SOLD) > 5 THEN 1 ELSE 0
END;
 GROUPBY[PROD_NAME, PROD_ID];
 PRIMARYKEY[PROD_ID];
 REFRESH ON CHANGES IN [PROD_DIM, SALES];
END
```

In this example, `TRACKDELETES` ensures that discontinued products are reflected in the `PROD_DIM` dataset without requiring deleted records to be handled separately.

Soft Delete

The deletion operation is applied during the extraction of data. The `DELETETYPE[SOFT]` flag is used to mark the deleted records, and the extraction process identifies which records need to be flagged. If you don't specify a flag name, the default flag `ISDELETED` is used.

```
IMPORT SOURCE CUSTOMERS DELETETYPE[SOFT] WITH PRIMARYKEY[CUST_ID] TRACKDELETES

DEFINE UPDATEABLE DATASET CUSTOMERS_SD_D FROM CUSTOMERS END
```

In this example, soft delete is initiated during the import of the `CUSTOMERS` dataset. The records identified by `TRACKDELETES` are flagged as deleted using the default `ISDELETED` column.

You can specify a custom flag name in the `DELETETYPE` directive:

```
IMPORT SOURCE SALES DELETETYPE[SOFT[ISTRANDELETED]] TRACKDELETES[IN[THDELETE]]

DEFINE UPDATEABLE DATASET CUSTOMERS_SD_D FROM CUSTOMERS END
```

DELETESOURCE

You can use the `DELETESOURCE` directive to remove records from datasets based on a matching condition, ensuring that records marked for deletion in the source system are excluded from the data processing job.

The `DELETESOURCE` directive deletes records during dataset transformations, based on a source dataset that tracks deletions.

Hard Delete

The `DELETESOURCE` directive specifies the deletion of records from a dataset based on a matching condition. You apply the directive within the dataset definition to handle the removal

of records that have been flagged for deletion in a source system. When DELETESOURCE is used, it directly impacts the dataset by ensuring that records marked for deletion are excluded from the dataset. This method requires specifying a subtrahend (deletion set) dataset, which contains the records to be removed. The matching condition is used to correlate records from the deletion set with those in the target dataset.

```
IMPORT SOURCE SALES_LOG FILTEREDBY (ACTION = 'D') AS SALES_DEL;
IMPORT SOURCE SALES;
DEFINE DATASET SALES_F
 ROWSOURCE SALES;
 THIS = SALES;
 DELETESOURCE SALES_DEL [SALES_ID] MATCHING [SALES_ID];
END;
```

In this example, the DELETESOURCE directive removes records from the SALES dataset based on matching SALES_ID values from the SALES_DEL dataset, which tracks deletions from the source system.

Soft Delete

The DELETETYPE[SOFT] flag marks records as deleted and the DELETESOURCE directive identifies the source dataset containing the records to be deleted.

```
IMPORT SOURCE SALESDEL
IMPORT SOURCE SALES
DEFINE UPDATEABLE DATASET SALES_F
 ROWSOURCE SALES;
 THIS = SALES;
 DELETETYPE[SOFT[SALESDELETED]];
 DELETESOURCE SALESDEL[SALES_ID] MATCHING [SALES_ID];
END
```

In this example, the DELETESOURCE directive deletes records from SALES that match the SALES_ID values in the SALESDEL dataset. The DELETETYPE[SOFT[SALESDELETED]] flag marks the deleted records, but they aren't physically removed from the dataset.

THEN DELETE

You can specify the THEN DELETE directive at the source reference level in the IMPORT SOURCE statement, rather than within dataset definitions.

By specifying the delete operation at the source level, you apply the delete operations earlier in the data pipeline, directly at the source level, ensuring that deletions are processed as soon as the source data is imported.

Hard Delete

You apply the THEN DELETE directive to the source reference in the import statement. It ensures that records marked for deletion in the source dataset are excluded during the import process.

```
IMPORT SOURCE SALES_LOG FILTEREDBY (ACTION = 'D') AS SALES_DEL;
IMPORT SOURCE SALES THEN DELETE SALES_DEL [SALES_ID] MATCHING [SALES_ID];
```

In this example, deletions are applied immediately when the SALES dataset is imported, based on the SALES_DEL dataset, which contains the records marked for deletion.

Soft Delete

You apply the `THEN DELETE` deletion logic after the data has been imported. The soft delete process executes, based on a list of records that you provide in the `THEN DELETE` directive. This method allows for the deletion to be handled separately from the import process, while still ensuring that deleted records are flagged in the dataset.

```
IMPORT SOURCE SALESDEL
IMPORT SOURCE SALES DELETETYPE[SOFT] THEN DELETE [SALESDEL[SALES_ID] MATCHING
[SALES_ID]]
DEFINE DATASET SALES_SD_F FROM SALES END
```

In this example, the `SALESDEL` dataset, which contains deletion records, is used in conjunction with the `THEN DELETE` directive. Records from `SALES` that match the IDs in `SALESDEL` are flagged as deleted, based on the `DELETETYPE[SOFT]` directive

2

Create Custom Data Pipelines

From the Oracle Fusion Data Intelligence Administrator Console, you can build organization-specific or industry-specific data pipelines programmatically with Data Augmentation Scripts (DAS) custom logic.

> **Note**
>
> Data Augmentation Scripts (DAS) is a preview feature.
>
> - For more information, see Preview Features.
> - Administrators can enable Preview Features. See Make Preview Features Available.

Topics:

About Creating Custom Data Pipelines

You can build custom data pipelines with logic that brings source data to meet your business requirements using the functionality in the Data Augmentation Scripts (DAS) application.

You can bring data from different sources such as Oracle Fusion Cloud Applications or Salesforce, join the different data together, bring the data as another table in the warehouse, and extend an entity using the additional data.

While creating the custom application, in the Data Augmentation Scripts (DAS) dialog, the application name you provide serves as an identifier that allows you to easily find and edit it later on. The application ID that you provide functions as a namespace, differentiating the tables into separate groupings.

Within each application, you see the Source folder that contains `main.hrf` and `main.mod` files. The `main.mod` file is read-only and provides information regarding the module's name, source type, and prefix. The `main.hrf` file contains the main logic for the data pipeline. You can add additional logic in the Code, Function, and Parameter types of files by right-clicking `main.hrf` and selecting New.

In your code, there is no need to explicitly reference the prefix or application ID because the code execution process automatically applies them. After you have added the code for your

custom logic, you must build to compile the custom data pipeline, verify that the syntax functions correctly, and to ensure that the source and related metadata is mapped properly. After a successful build, your code is ready to be deployed. The build step produces the mapping logic, target table structure, and loading directives based on the source metadata You must successfully build before deploying the application.

- Deploy: The deploy step initiates the actual execution of Extract, Process, and Load phases into the data warehouse. This is the initial full load of this application.
- Verify (optional): After deployment, you can verify the creation and loading of the tables. If you want to further examine the data, you can execute additional `SELECT` statements as needed.
- Update: To edit an existing Data Augmentation Scripts (DAS) application source, open any of the files, such as `main.hrf`, and make the necessary changes.

There is no limit to the number of times that you can edit an existing Data Augmentation Scripts (DAS) data file. After making changes each time, compile or build the files to validate the syntax. After the syntax is validated, deploy the application to refresh the source data in the warehouse.

You can load subsequent data manually using the **Refresh** option or load periodically based on the configuration settings.

Prerequisites for Creating a Custom Data Pipeline

Prior to creating a custom data pipeline for Data Augmentation Scripts (DAS), ensure that you create a connection to the data source.

1. Create a connection to the source that you want to use in your Data Augmentation Scripts (DAS) application. See Create a Connection for Data Augmentation Scripts.
2. After creating the connection, to complete the registration of the data, on the Manage Connections page, select the **Actions** menu for the Data Augmentation Scripts connection, and then select **Refresh Metadata**.

> **Note**
>
> You can’t create augmentations using a specific source unless you perform a metadata extract.

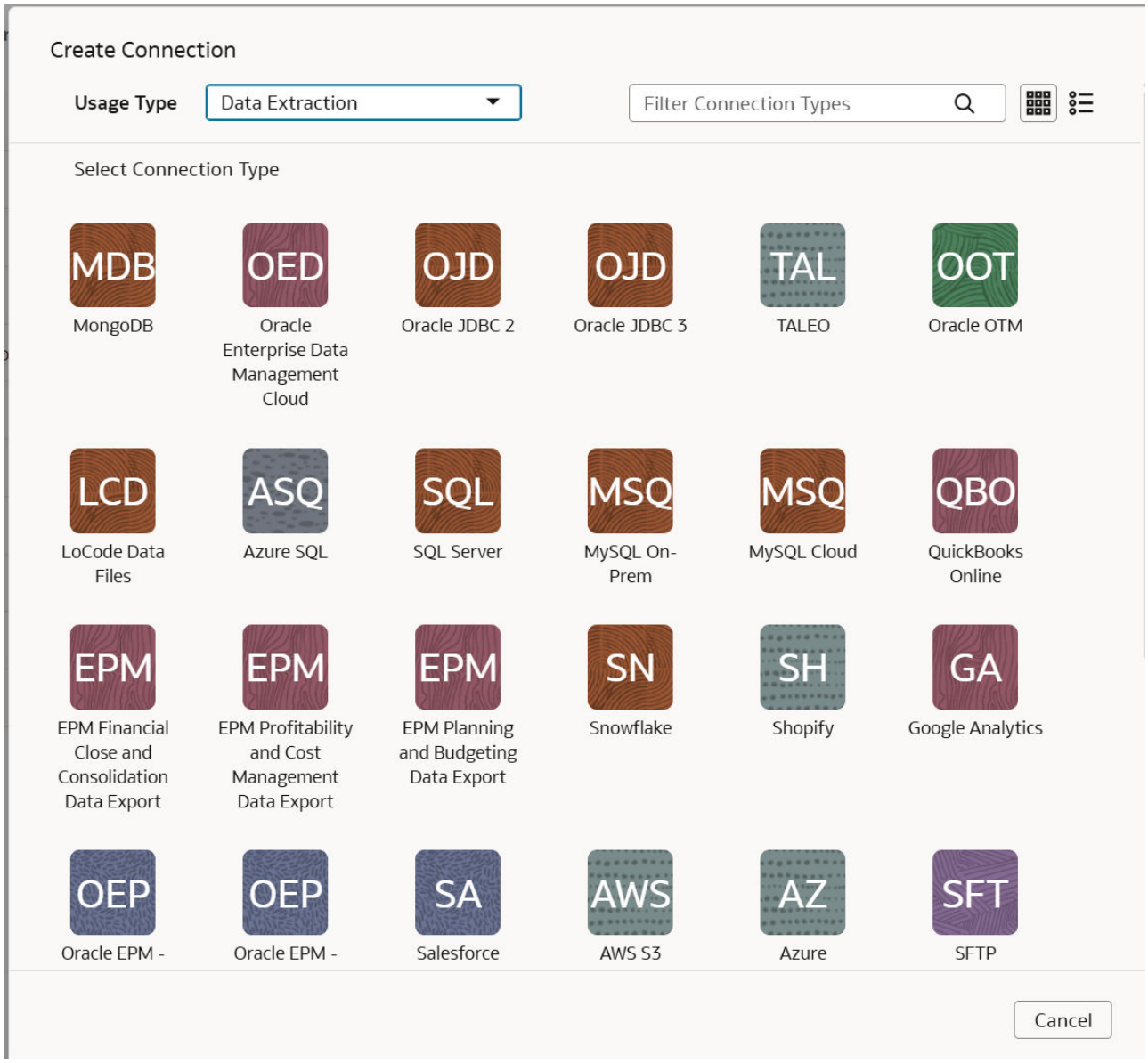
3. On the Data Configuration page, in **Data Source**, select the source for which you created a connection and set up the Data Augmentation Scripts pipeline parameters. See Set Up Pipeline Parameters for Data Augmentation Scripts.
4. Create an augmentation because prior to creating a custom Data Augmentation Scripts (DAS) application, you must have at least one existing augmentation. See Create Augmentation for Data Augmentation Scripts.
5. Create your custom Data Augmentation Scripts (DAS) application. See Create a Data Augmentation Scripts Application

Create a Connection for Data Augmentation Scripts

As a functional administrator, create a connection to create a data source to the data warehouse.

1. In Oracle Fusion Data Intelligence **Console**, click **Data Configuration** under **Application Administration**.
2. On the Data Configuration page, click **Manage Connections** under Configurations.
3. On the Manage Connections page, click **Create** and then click **Connection**.
4. In Create Connection:
   a. For Usage Type, select **Data Extraction**.
   b. For Select Connection Type, select **LoCode Data Files**.

      These are your sample Data Augmentation data files to help you get started. You can select a different connection to extract data for the data warehouse.



5. In Create Connection:
   a. (Optional) Enter a **Notification Email** to receive notifications.
   b. Enter a **Sample Data Set**. Example: `Sales-Sample`.

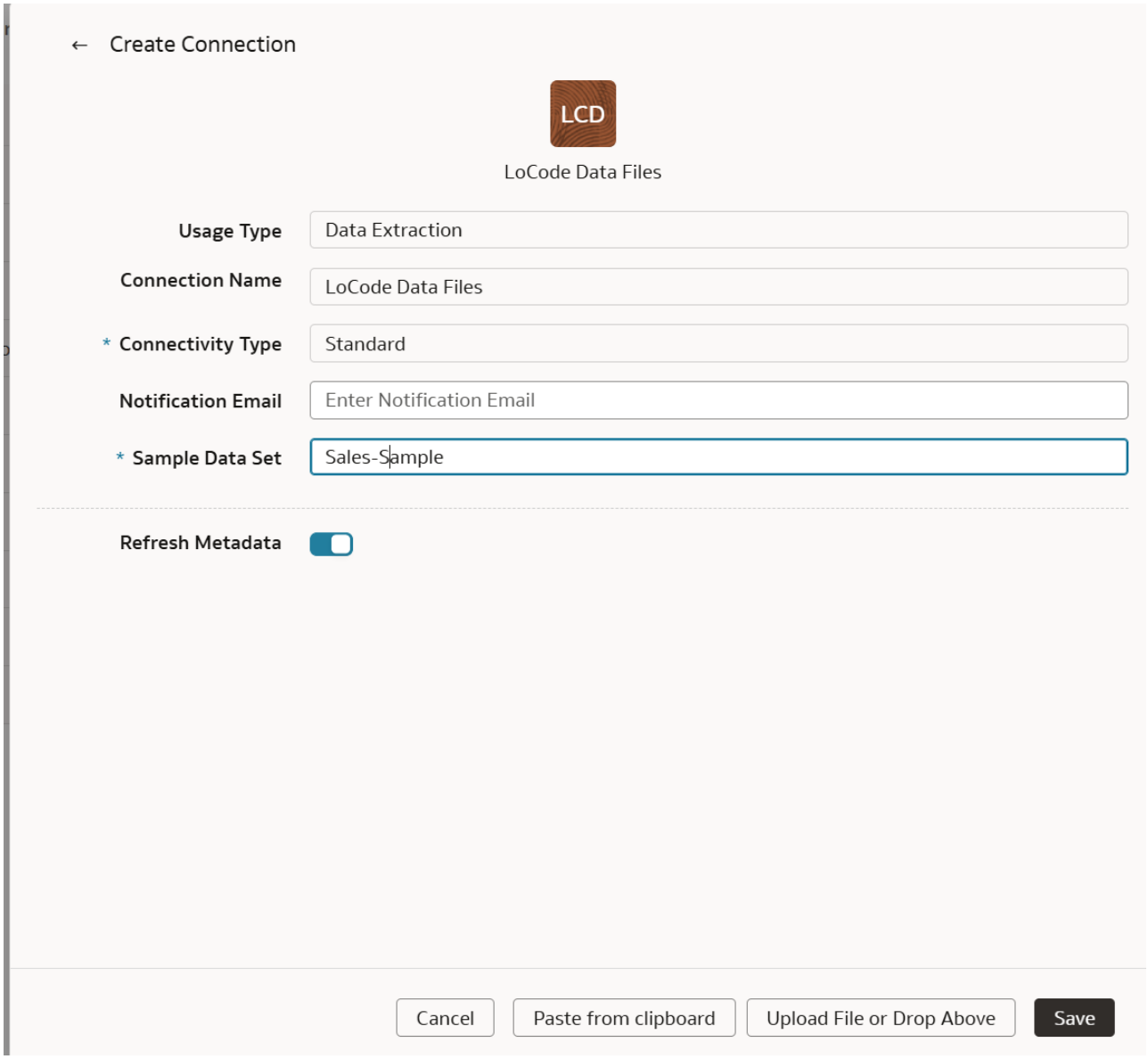
The source data files provide sample data sets, such as Sales, Customers, that you can use for Data Augmentation Scripts (DAS).

c. Enable **Refresh Metadata** to ensure that the metadata is refreshed when you save the connection.

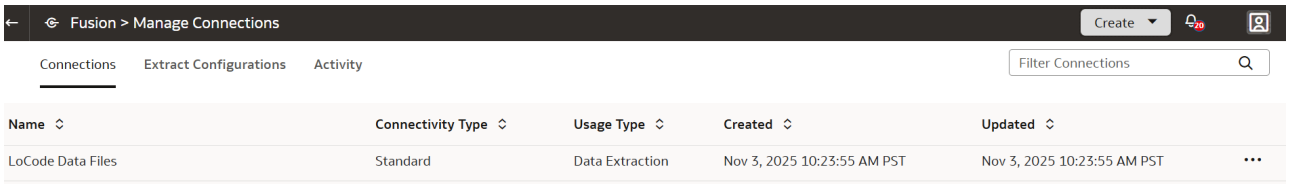
You can later refresh the metadata from the Actions menu on the Manage Connections page, if required.

> **Note**
>
> You can't create augmentations for the Data Augmentation Scripts (DAS) application unless you perform a metadata extract.

d. Click **Save**.



After the connection is created, you can view the source data files under Connections on the Manage Connections page.



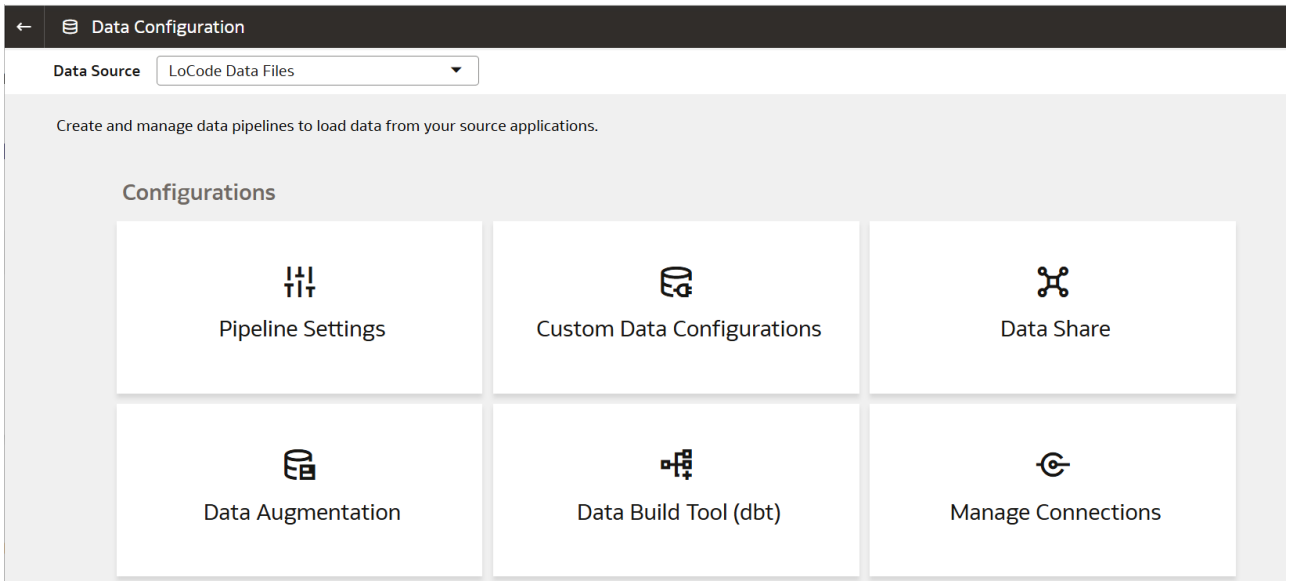
After your connection is created, you can create a custom data pipeline with Data Augmentation Scripts (DAS). See Create a Data Augmentation Scripts Application.

Set Up Pipeline Parameters for Data Augmentation Scripts

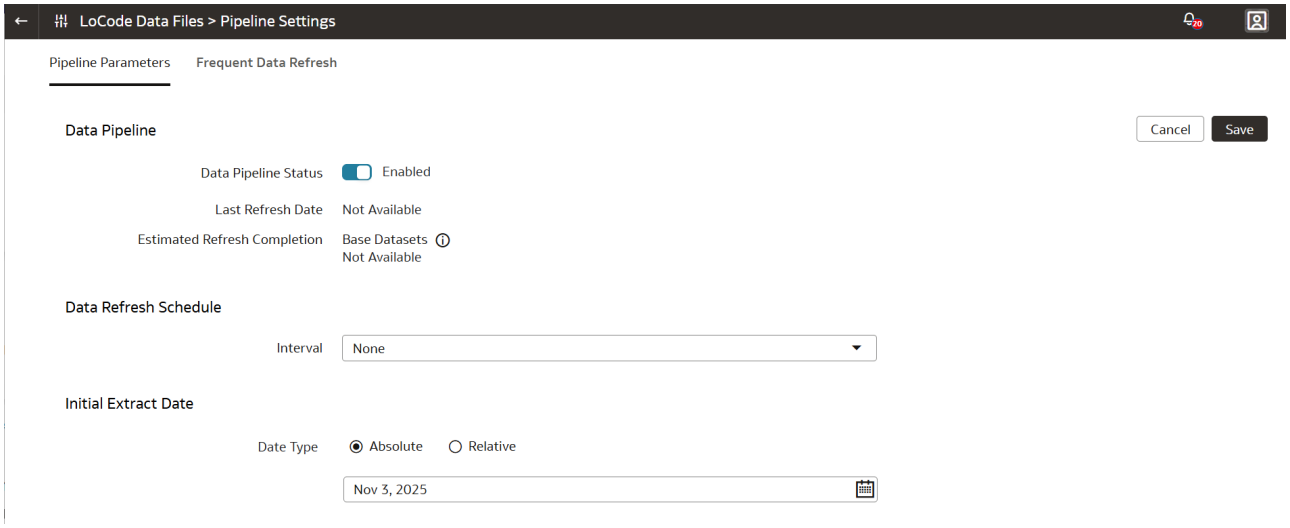
Set up pipeline parameters before proceeding to work with the Data Augmentation Scripts (DAS) application.

For more information, see Set Up the Pipeline Parameters.

1. Sign in to your service.
2. In Oracle Fusion Data Intelligence, **Console**, click **Data Configuration** under **Application Administration**.
3. On the Data Configuration page, select **LoCode Data Files** for **Data Source**, then click **Pipeline Settings** under **Configurations**.



4. On the Pipelines Settings page:
   a. Ensure that **Data Pipeline Status** is **Enabled**.
   b. (Optional) Select an **Interval** for **Data Refresh Schedule**.
   c. Select a **Date Type**. Enter a date if you select **Absolute**.



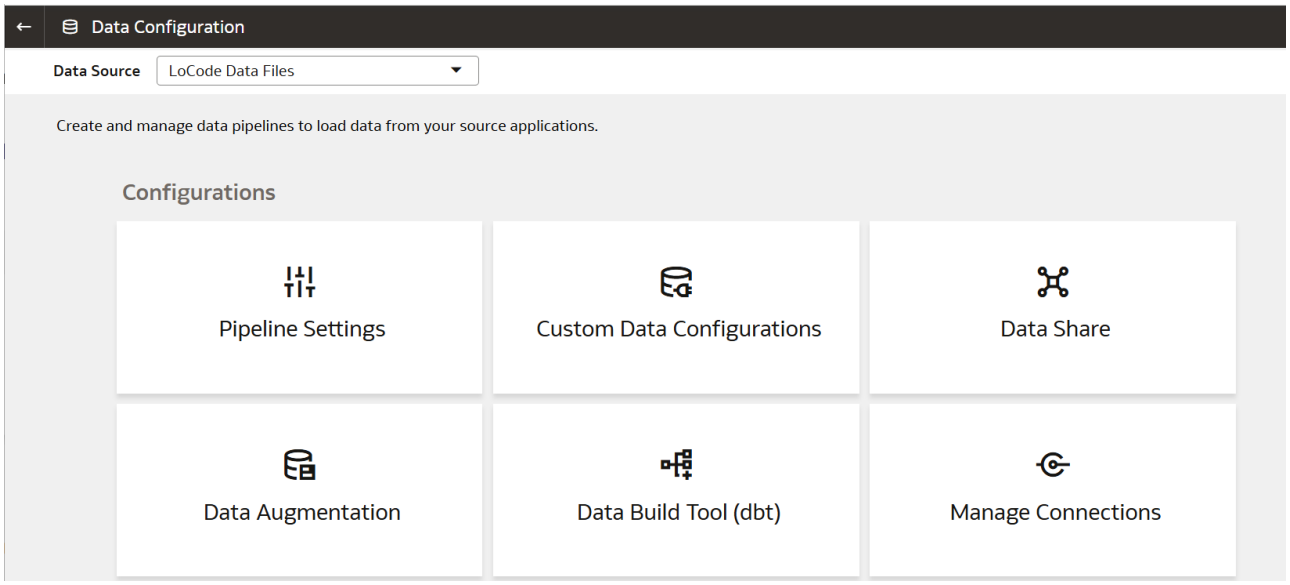
5. Click **Save**.

Create Augmentation for Data Augmentation Scripts

Create an augmentation prior to creating the Data Augmentation Scripts (DAS) application.

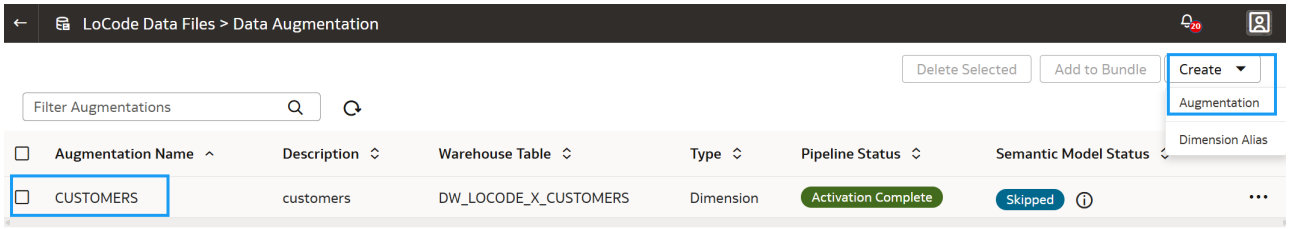
You must have at least one existing data augmentation before you set up the Data Augmentation Scripts (DAS) application. See Augment Your Data.

1. Sign in to your service.
2. In Oracle Fusion Data Intelligence, **Console**, click **Data Configuration** under **Application Administration**.
3. On the Data Configuration page, select Data Augmentation Scripts Data Files for **Data Source**, then click **Data Augmentation** under **Configurations**.



4. On the Data Augmentation page, select the Augmentation, click **Create** and then select **Augmentation**.

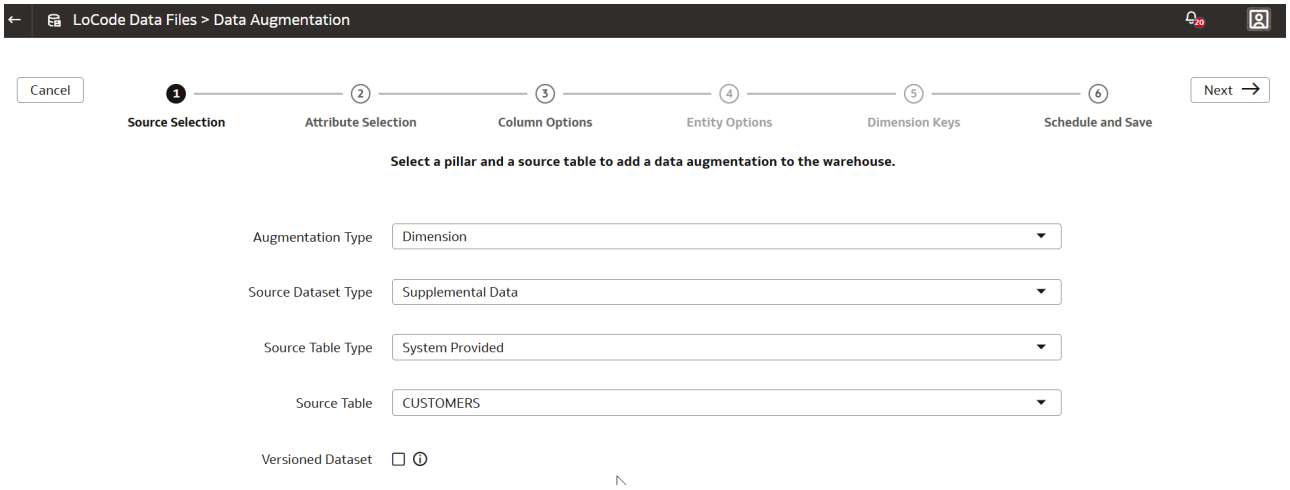
   The dimension data augmentation is for the Data Augmentation source, `CUSTOMERS`. Depending on your source, you can select another dimension for data augmentation.



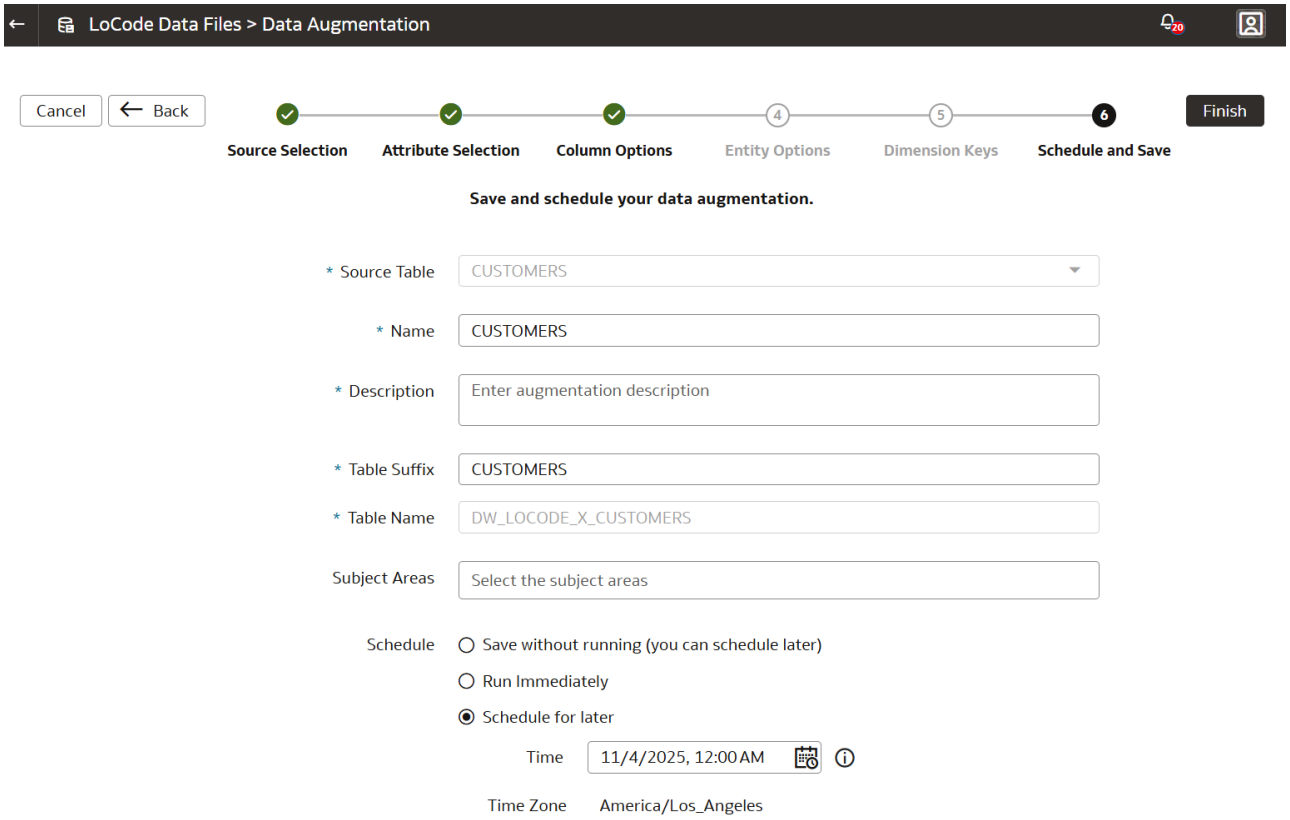
5. In the Data Augmentation wizard, add a new dimension:

   See Create Dimension Augmentation Type

   a. For Augmentation Type, select **Dimension**.
   b. For Source Dataset Type, select **Supplemental Data**.
   c. For Source Table Type, select **System Provided**.
   d. For Source Table, select **CUSTOMERS**.



6. Click **Next**.
7. For **Attribute Selection**, select the attributes for the source table.
8. Select the default **Column Options**.



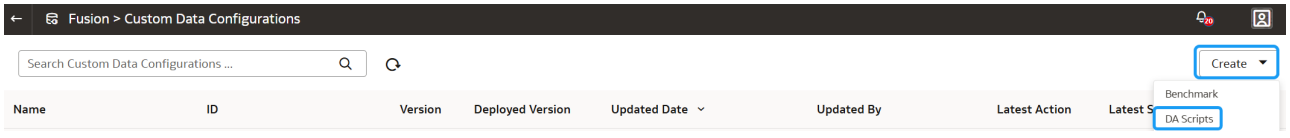
9. Continue to **Schedule and Save**.
10. Click **Finish**.

    The CUSTOMERS Data Augmentation (DA) initiates. Wait a few minutes for the augmentation to complete. The Pipeline Status updates to **Activation Complete**.

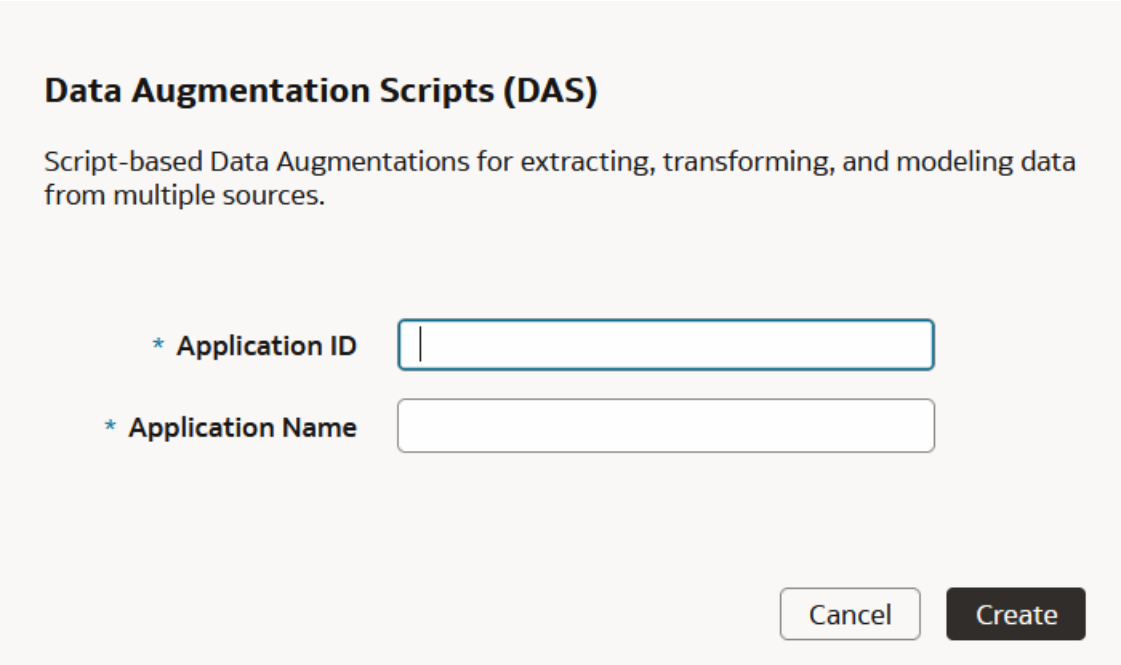
Create a Data Augmentation Scripts Application

Create a Data Augmentation Scripts (DAS) application to use your custom logic to bring data from the source to your application data files.

1. Sign in to your service.
2. In Oracle Fusion Data Intelligence, **Console**, click **Data Configuration** under **Application Administration**.
3. On the Data Configuration page, under **Configurations**, click **Custom Data Configurations**.
4. On the Custom Data Configurations page, click **Create**, and then select **DA Scripts**.

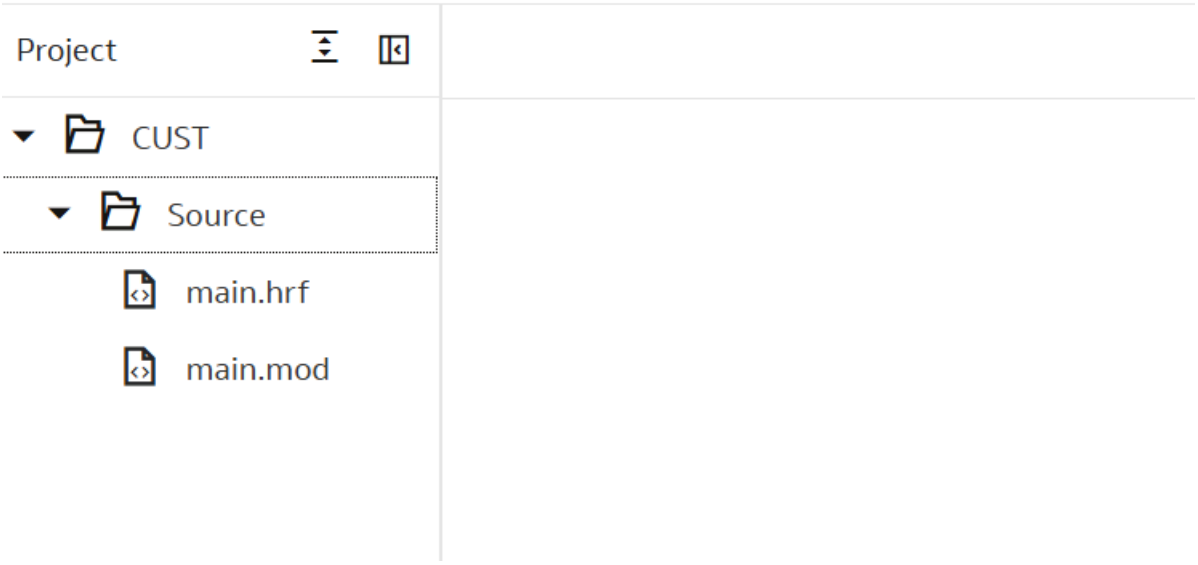


5. In the Data Augmentation Scripts (DAS) dialog, enter an ID in **Application ID**, name in **Application Name**, and then click **Create**.



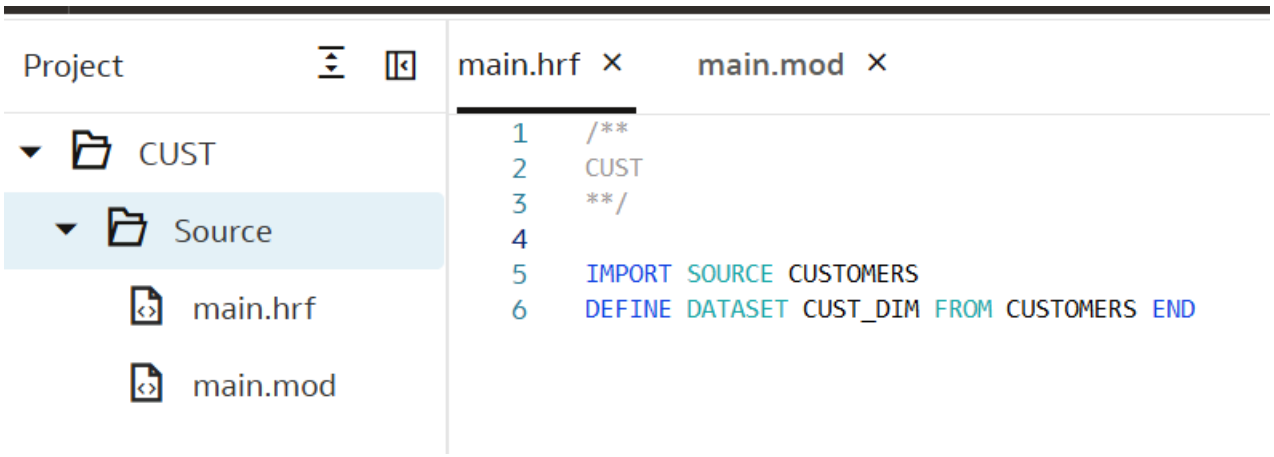
6. On the DA Scripts page, in the left menu, click **Source**.
7. Under Source, select **main.hrf**, and in the editor, enter the logic to bring data.

   In the following example, you create a data augmentation script for CUST_D using the sample Data Augmentation Scripts source files.

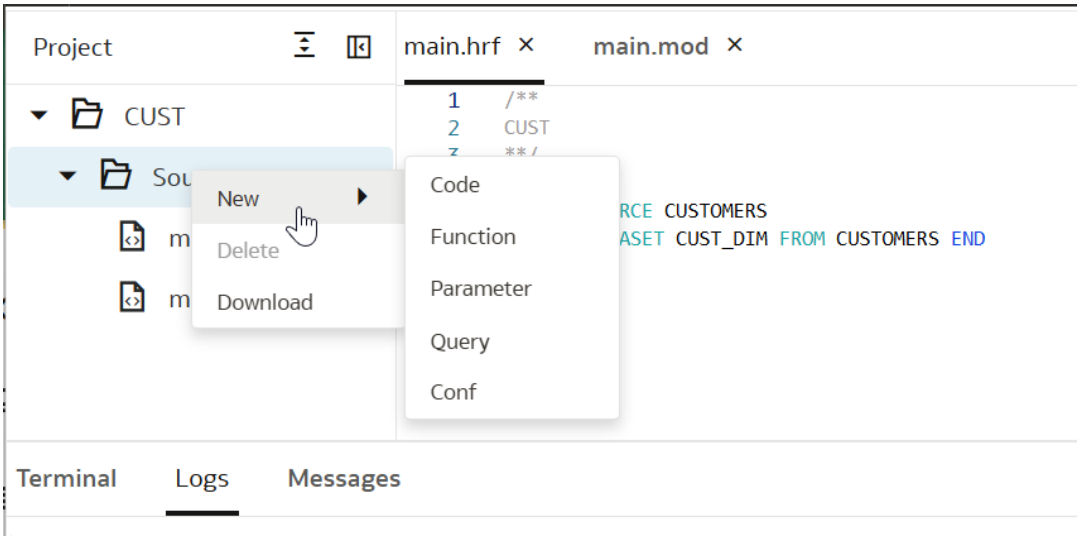


To build your dataset, you can reference data from your sources into the data warehouse using the `IMPORT` command.

See IMPORT Statement.



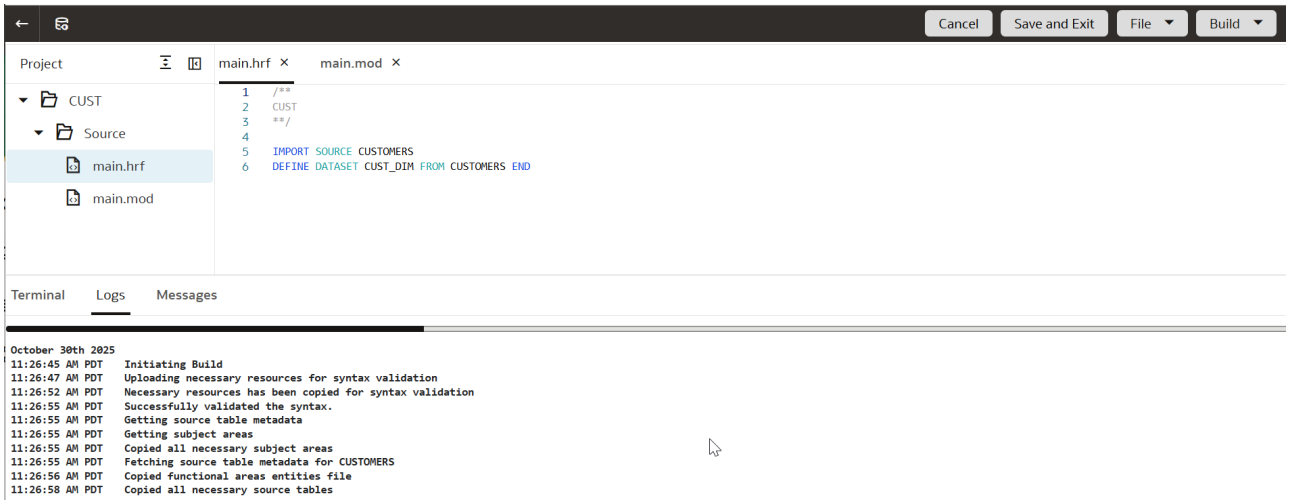
8. Right-click the Source folder, select **New** and then select the type of file you want to create.



See Supported Files.

9. Click **Save and Exit**.
10. Click **Build** and then select **Build project**.

The project compiles and validates your syntax and to ensure that the source metadata are mapped properly.



11. Click the **Actions** icon next to the DAS Scripts application you created, and then click **Publish**.

    This initiates the deployment and execution of Extract, Process, and Load phases into the data warehouse.

A

Data Augmentation Scripts Application Development Details

Use these features, syntax, and other functions of Data Augmentation Scripts to build your application.

Topics:

B

Syntax Notations

The following table lists the syntax notations and their corresponding meaning:

Symbols	Meaning
`::=`	Indicates *is defined as*.
`|`	Indicates alternatives. Separate alternatives using vertical bars. Example: `a	b` means for *a or b*.
`{rule1 | rule2}`	Insert alternatives within curly parenthesis in complex production rules.
`[rule]`	Indicate options by using square brackets. Example: `[ a ]` stands for an optional `a` value.
…	Indicates repetition. Example: `a…` means rule `a` can be repeated multiple times.

C

Comments and Escape Sequences

Comments help to keep your code readable and organized, while escape sequences enable handling special characters.

Comments

Comments include explanations or notes about the code.

Comments can be either:

- **Single-line comments:**
  Single-line comments have a double forward slash `//`.

  Example:

  ```
  // Example of a valid single line comment.
  ```

- **Multi-line or block comments:**
  Multi-line or block comments have `/*` (start of a block comment) and `*/` (end of a block comment),

  Example:

  ```
  /* Example of a
  valid block comment
  */
  ```

Escape Sequences

Escape sequence are special characters, such as new lines, tabs, or quotation marks, that you can't directly enter into a string. They start with a backslash `\` followed by a character.

Examples:

- `\n`: newline
- `\t`: tab
- `\r`: carriage return
- `\'`: single quote
- `\"`: double quote
- `\b`: backspace
- `\f`: form feed
- `\v`: vertical tab

D

File Types and Data Types

The Data Augmentation Scripts application supports these documented file types and data types.

Review these file types and data types that Data Augmentation Scripts supports.

- File Types
- Data Types

File Types

You can create files in their respective folders for different purposes. You can create these types of files:

- **.param**: The `.param` file (Parameter defiition file). contains one or more parameter definitions. Data Augmentation Scripts creates this file in the Parameter folder under the project root directory.
  For details on parameter definitions, see PARAMETER.
- **.mod**: This is a read-only file. The `.mod` file contains the Data Augmentation Scripts application definition. Data Augmentation Scripts creates this file in the Source folder under the project root directory.
  Syntax

  ```
  module_definition ::= MODULE module_name
  SOURCETYPE source_type
  NAMESPACE namespace
  PREFIX prefix
  ```

  Example of a `main.mod` file:

  ```
  MODULE TIME
  SOURCETYPE FUSION
  NAMESPACE TIME_
  PREFIX DW_FA_X_
  ```

- **.hrf**: The `.hrf` file contains the Data Augmentation Scripts program. The application creates this file is in the Source folder under the project root directory.
  Example of `customers_d.hrf` file:

  ```
  IMPORT SOURCE CUSTOMERS
  DEFINE DATASET CUSTOMERS_D
    ROWSOURCE CUSTOMERS;
    THIS = CUSTOMERS;
  END
  ```

- **.func** : The `.func` file consists of user-defined function definitions. Data Augmentation Scripts creates this file in the Function folder under the project root directory.
  For more details, see User Defined Functions (UDFs) or Macros.

- **.qry**: Use the `.qry` file to create a view using the query that's written in the file. Data Augmentation Scripts creates this file in the Query folder under the project root directory. For more details, see Query Files.
- **.conf**: To improve the performance of reporting queries in the database, you can specify constructs such as indexes, column groups, and partitions in this file. Data Augmentation Scripts creates this file in the Conf folder under the project root directory. For more details, see Column Groups, Indexes, and Partitions.

Data Types

The Data Augmentation Scripts application functionality supports these data types:

Table D-1 Data Augmentation Scripts Supported Data Types

Data Type	Declaration	Value Example
NUMBER(precision, scale optional)optional	NUMBER NUMBER (28,2)	3 28 -28
LARGEINT	LARGEINT	5000 75000
BIGDECIMAL(precision, scale)	BIGDECIMAL(28,2)	3.14 1.00
DATE 'yyyy-MM-dd' or DATE "yyyy-MM-dd"	DATE '1843-03-02' or DATE "1843-03-02"	‘1843-03-02' "1843-03-02"
TIMESTAMP'yyyy-MM-dd HH:mm:ss.SSSS' or TIMESTAMP "yyyy-MM-dd HH:mm:ss.SSSS"	TIMESTAMP '1843-03-02 04:28:59' or TIMESTAMP "1843-03-02 04:28:59"	‘1843-03-02 04:28:59' "1843-03-02 04:28:59"
VARCHAR2(unsigned_integer)	VARCHAR2(38)	"Hello" "Hello World” ‘Hello World’ “Hello World 101”

E

Program Structure

A program is essentially a set of code in the Data Augmentation Scripts application that consists of specific elements.

The structure of a program includes the following elements:

- IMPORT
- INCLUDE
- ALIAS
- PARAMETER
- STATEMENT

Syntax

```
program ::= {   application_source_definition
                                              | import_definition
                                              | include_definition
                                              | alias_definition
                                              | parameter_definition
                                              | list_variable_assignment
                                              | statement
                                            } ...
```

For application source definition details, see Table and Column Prefixes.

IMPORT Statement

You can use the IMPORT statement to load objects, such as modules, entities, or source tables into the current application, which you can then use as a source to build the pipeline.

Syntax

```
import_definition ::= IMPORT
                      {
                        MODULE {module_artifact | module_artifact_list}
                        | ENTITY {extended_item | extended_item_list}
                        | source_definition
                       }
```

Instructions for importing modules and entities, source definitions, optional attributes, filters, and aliases are in the following sections.

Import Modules and Entities

```
module_artifact ::= module_name
module_artifact_list ::= '[' module_name [, module_name] ... ']'
```

```
extended_item ::= entity_name
extended_item_list ::= '[' entity_name [, entity_name] ... ']'
```

This code defines the modules or entities to import.

> **Note**
>
> To use a data warehouse table or dataset from another module, you must first import the corresponding module.

Example:

```
IMPORT MODULE [FA_GL, FA_AP]
IMPORT ENTITY Item
```

Use a terminal to check the available modules, entities, and their definition.

Source Definition

The following code defines a source with optional attributes, filters, aliases, and delete specifications:

```
source_definition ::= [source_type] SOURCE
                      {
                        source_reference_list
                        | source_reference [ soft_delete_spec ]
[override_list]
                        [ then_delete_specification ]
                        [ WITH source_attribute ]
                        [ FILTEREDBY '(' boolean_returned_expression ')' ]
                        ([ AS filtered_source_name ] |
[TABLEPREFIX'['string']'] ) [ COLPREFIX'['string']']
                       }
```

Source Type

```
source_type ::= VERSIONED | UPDATEABLE | ENTITYCHANGETRACKING
```

The default for `source_type` is `UPDATEABLE`.

Source types are:

- **VERSIONED**: During each incremental run, Data Augmentation Scripts extracts all data from the source and fully refreshes.
  Deleted records aren't retained in the data warehouse. When the `SOURCE` is of type `VERSIONED` and you specify `Last Update Date` (LUD) , then Data Augmentation Scripts ignores it.
- **UPDATEABLE**: During each incremental load, Data Augmentation Scripts extracts new and changed records from the source.

Unchanged and deleted records are retained in the staging area. When the `Last Update Date` (LUD) isn't part of IMPORT SOURCE definition, Data Augmentation Scripts extracts all records from the source system and updates the data in the staging area.

- **ENTITYCHANGETRACKING**: During each incremental load, Data Augmentation Scripts extracts new and changed records based on their natural key from the source system.

Source Reference

```
source_reference_list ::= '[' source_reference [, source_reference] ... ']'
source_reference ::= source_name
```

Data Type Override

```
Override_list ::= OVERRIDE '[' column_name -DATATYPE data_type [, column_name
-DATATYPE data_type]… ']'
```

You can convert the source column data types.

Example:

```
OVERRIDE [createddate -DATATYPE TIMESTAMP , amount -DATATYPE NUMBER(20,2) ]
```

Base Delete Specification

```
then_delete_specification ::= THEN DELETE "[" delete_source [,
delete_source] ... "]"
delete_source ::= table_name column_list MATCHING column_list
```

Example:

```
THEN DELETE [DEL_SALES [SALES_ID] MATCHING [SALES_ID]]
```

For more details about base deletions, see Deletions.

Soft Delete Specification

```
soft_delete_spec ::= DELETETYPE '[' SOFT [create_soft_delete_column] ']';
create_soft_delete_column ::= column_name
```

Example:

```
IMPORT SOURCE SALES DELETETYPE[ SOFT ] THEN DELETE [ SALESDEL [SALES_ID]
MATCHING [SALES_ID] ]
```

> **Note**
>
> You can use `soft_delete_spec` with `then_delete_specification` or `track_deletes_dataset`.

For more details about soft delete, see Deletions.

Source Attributes

```
source_attribute ::= [primary_key_spec]
              [ ied_key_spec ]
              [ entity_id_spec]
              [lud_key_spec] [track_deletes_dataset])
              [stability_period]
```

You can use these source attributes:

- `primary_key_spec ::= PRIMARYKEY column_list`: Defines the primary key.
- `ied_key_spec ::= IED column_list`: Defines the initial extract date column.
- `lud_key_spec ::= LUD { column_list | '[' NULL ']' }` Defines the last update date columns.

> **Note**
>
> If you define an incremental key in the source metadata, Data Augmentation Scripts automatically uses it as the Last Update Date (LUD). To override this behavior, use `LUD[NULL]`.

- `entity_id_spec ::= ENTITYID column_list` Defines natural keys. You must define, `ENTITYID` with `source_type ENTITYCHANGETRACKING`.
- `track_deletes_dataset ::= TRACKDELETES [ IN '[' identifier ']' ]`

  ```
  stability_period ::= STABILITYPERIOD '[' number_of_days ,
  tracking_date_column ']'
  number_of_days ::= unsigned_integer
  tracking_date_column ::= column_name
  ```

For more details about Stability Period and Base Data Delete Dataset, see Deletions.

Column List

```
column_list ::= '[' column_name [, column_name] ... ']'
```

Use this code to list columns for key specifications.

Alias Name

```
filtered_source_name ::= alias_name
```

Table and Column Prefix

Use `TABLEPREFIX` and `COLPREFIX` for defining table and column prefixes.

For details, see Table and Column Prefixes.

Examples

Example 1

```
IMPORT SOURCE fiscalCalendar WITH PRIMARYKEY [fiscal_year] FILTEREDBY
('fiscal_year D
2020') AS RecentFiscalYears
```

Example 2: VERSIONED source

```
IMPORT VERSIONED SOURCE PRODUCT WITH PRIMARYKEY[PROD_ID]
```

Example 3: ENTITYCHANGETRACKING source

```
IMPORT ENTITYCHANGETRACKING SOURCE BusinessUnit  OVERRIDE [CreationDate -
DATATYPE TIMESTAMP ,
LegalEntityId -DATATYPE VARCHAR2(20) ]

WITH PRIMARYKEY [BusinessUnitId,StartDate,EndDate] IED[CreationDate] LUD
[LastUpdateDate]
ENTITYID[BusinessUnitId]

FILTEREDBY(Status='A' OR Status='U') AS BUnit
```

INCLUDE

Use the INCLUDE definition to include the supported file types in the current application.

For more details on supported files and their usage, see File Types and Data Types.

Syntax

```
include_definition ::= INCLUDE {HRF | FUNCTION | PARAMETER } file_reference
```

Example

```
INCLUDE PARAMETER "constant.param"

INCLUDE FUNCTION "coalesceUDF.func"

INCLUDE HRF "dimension.hrf"
```

> **Note**
>
> You don't need to use the `INCLUDE` definition for `.conf` and `.qry` files because you can directly use them in the code within `.hrf` files.

ALIAS

You create an alias for a table to make referencing it in your code simpler and more concise. Using an alias enables you to streamlines queries and improves readability, especially in complex statements.

Syntax

```
alias_definition ::= ALIAS table_source_type table_name AS table_alias
table_source_type ::= {EXTERNAL | FACTORY | LOCAL}
```

Example

```
ALIAS LOCAL CUSTOMERS_D AS CUST
```

Table Source Types

Table source types include:

- **LOCAL**: You define local aliases on the datasets in the same Data Augmentation Scripts application.
  Example:

  ```
  DEFINE DATASET CHANNELS_D FROM CHANNELS END
  ALIAS LOCAL CHANNELS_D AS CHAN
  ```

- **EXTERNAL**: You define external aliases on the imported source tables.
  Example:

  ```
  IMPORT SOURCE CUSTOMERS
  ALIAS EXTERNAL CUSTOMERS AS CUST
  ```

- **FACTORY**: You define factory aliases on tables in the other modules.
  Example:

  ```
  IMPORT MODULE FA_GL
  ALIAS FACTORY DW_LEDGER_D AS LEDGER
  ```

  In this example, `DW_LEDGER_D` is a warehouse table in the module `FA_GL`

> **Note**
>
> - If multiple tables share the same name across different table source types (`LOCAL`, `FACTORY`, `EXTERNAL`) and you define an alias without specifying `table_source_type`, Data Augmentation Scripts assigns the alias to the table based on the preference order: LOCAL > FACTORY > EXTERNAL. This means the alias is first created for a LOCAL dataset, followed by a FACTORY table, and then an EXTERNAL source.
> - After you define an alias, you must use the alias in the code instead of the actual table name.

PARAMETER

You define a parameter as a named variable with a specific data type and, optionally, a default value for use within queries. Parameters enable you to pass dynamic values into your queries efficiently.

Syntax

```
parameter_definition ::= DEFINE PARAMETER parameter_name ,
data_type ,default_value END
```

Example

```
DEFINE PARAMETER NOVALUE_NUMBER,NUMBER(38,0),-99999 END
DEFINE PARAMETER VAR_ETL_FALSEVALUE,VARCHAR2(16),"F" END
DEFINE PARAMETER VAR_ETL_NOVALUE_CHAR_NAME,VARCHAR2(32),"~No Value~" END
```

You can store parameters in separate files, which you can then use in the Data Augmentation Scripts code with the INCLUDE command or reference the PARAMETER definition in the transformation code itself.

For more details, see File Types and Data Types.

> **Note**
>
> - Use unique parameter names. You can't define parameters with the same names as those that are already existing system parameters.
> - You can directly use factory Parameters in the Data Augmentation Scripts application. You can check the list of available factory parameters from the terminal.

Statement

A statement defines a unit of execution that establishes macros, schemas, tables or datasets, or list variables. Use statements to organize and control distinct functional components within your code.

Syntax

```
statement ::= macro_declaration
                                        | generic_table_definition
                                        | list_variable_assignment
                                        | schema_definition
                                        | disable_delete_propagation
```

Generic Dataset Definition

The Data Augmentation Scripts dataset is a structured collection of data that you gather, process, or transform to use as an input or building block for creating other datasets in the Autonomous Data Warehouse.

Dataset Definition

A dataset definition has the following characteristics:

- Defines the data structures and properties.
- Defines the inputs (sources or datasets) through transformations.
- Specifies how it should be made available in the data warehouse.
- Includes instructions for data loading, refreshing, and deletion.

Syntax

```
generic_table_definition ::= DEFINE [ export_specification ] [ table_type ]
DATASET table_name
                                    { code_block | code_block_load |
from_clause }
                                     END
```

Export Specification

Export specification defines the accessibility of the dataset, determining whether it is restricted, controlled, or available.

By default, the export specification is PUBLIC.

Export specifications are:

- **PRIVATE**: Accessible only within the coding scope, and not materialized in target.
- **PROTECTED**: Accessible within a controlled scope, such as used to create VIEW or for debugging, but not publicly accessible.
- **PUBLIC**: Accessible in the target data warehouse.

Syntax

```
export_specification ::= PRIVATE | PROTECTED | PUBLIC
```

Table Type

Table Type defines the dataset's update capability, versioning, change tracking, storage method, and persistence.

The default update is UPDATABLE, which updates as deltas.

Syntax

```
table_type ::= UPDATEABLE | VERSIONED | ENTITYCHANGETRACKING | INLINE | VIEW|
AGGREGATIONONLY
```

The table types are:

- **UPDATEABLE**: Handles modifications (insert and update) as deltas. For delete, refer to Delete Handling
- **VERSIONED**: Maintains the latest version of data changes.
- **ENTITYCHANGETRACKING**: Tracks changes at the entity level for auditing or synchronization. (Internal)
- **INLINE**: Provides structure and data in the definition. See Inline Dataset.
- **VIEW**: Represents a read only, computed dataset derived from other datasets. See VIEW Dataset.
- **AGGREGATIONONLY**: Contains only summarized data in a single row. See AGGREGATION ONLY Dataset.

Example: In the following code, the versioned dataset `CUSTOMER_DIM` is refreshed entirely with each source load and available in the target data warehouse:

```
DEFINE PUBLIC VERSIONED DATASET CUSTOMER_DIM FROM CUSTOMERS END
```

VIEW Dataset

Example:

```
IMPORT SOURCE PRODUCTS
DEFINE DATASET PROD_DIM
 ROWSOURCE PROD_DIM;
 THIS = PRODUCTS;
END
DEFINE VIEW DATASET CURRENT_PRODUCT_D
 ROWSOURCE PROD_DIM WHERE DATEDIFF(PROD_DIM.PROD_EFF_TO , DATE '2020-01-01')
> 0;
 THIS = PROD_DIM;
END
```

> **Note**
>
> - View is always accessible as PUBLIC. For this reason, you can't specify `export_type` as `PRIVATE` or `PROTECTED`.
> - You can input only data sets, not sources.
> - You're not required to enter the primary key.
> - You can also define views using SQL. See VIEW QUERY .

Code Block

Code Block defines data sources, structures, and relationships, as well as instructions for data loading, refreshing, and deletion.

Syntax

```
code_block ::= [ builtin_schema_statement | template_schema_statement ]
         rowsource_specification [column_mapping_assignment]...
```

```
[ default_row_specification ]
[ aggregate_specification ]
[ primary_key_specification ]
[ entity_id_specification ]
[ incremental_refresh_directive ]
[ delete_specification ]...
```

Schema Statements

Example:

```
builtin_schema_statement | template_schema_statement
```

For details about schema statements, refer to Schema Definition.

Row Source Specification

Syntax

```
rowsource_specification ::= ROWSOURCE { table_source | dataset_source |
inline_source }

table_source ::= rowsource_expression

rowsource_expression ::= table_reference join_condition...
[ row_filtering_conditional ]

join_condition ::= {join_type table_reference ON join_expr | join_type_cross
table_reference }

join_type ::= { INNER | LEFT OUTER | RIGHT OUTER | FULL OUTER } JOIN
join_type_cross ::= CROSS JOIN

join_expr ::= boolean_returned_expression

row_filtering_conditional ::= WHERE boolean_returned_expression

dataset_source ::= { UNION | UNION-ALL } table_reference_list
table_reference_list ::= '[ ' table_reference [ , table_reference ]... ' ]'
table_reference ::= table_name | table_alias

inline_source ::= INTABLE '( ' column_and_type inline_value ' )'
column_and_type ::= '[ ' column_name : data_type [, column_name :
data_type ]... ' ]'

inline_value ::= VALUES '( ' row_expression [, row_expression ]... ' )'
row_expression ::= '[ ' constant_value [, constant_value ]... ' ]’
```

Example: Inner Join

```
IMPORT SOURCE CUSTOMERS
IMPORT SOURCE COUNTRIES

DEFINE VERSIONED DATASET GAMING_CUSTOMER_REACHED_G
 ROWSOURCE CUSTOMERS INNER JOIN COUNTRIES ON (CUSTOMERS.COUNTRY_ID =
```

```
COUNTRIES.COUNTRY_ID);
 THIS = COUNTRIES[COUNTRY_REGION,COUNTRY_SUBREGION];
 THIS = CUSTOMERS;
 PRIMARYKEY[CUST_ID];
END
```

Set Operations

A set operation works with two or more datasets and combines them into a single result set as a `ROWSOURCE`.

- You're required to specify column-mapping to define the structure of the resulting table.
- You need to explicitly specify a primary key if you want one, because there's no default primary key.

Example: UNION

```
IMPORT SOURCE PRODUCTS
DEFINE DATASET PRODUCTS_D FROM
PRODUCTS[PROD_ID,PROD_NAME,PROD_EFF_FROM,PROD_EFF_TO,PROD_VALID] END
DEFINE PRIVATE DATASET PC_PRODUCTS_TMP
 ROWSOURCE PRODUCTS WHERE
 PRODUCTS.PROD_CATEGORY = 'Hardware' AND PRODUCTS.PROD_SUBCATEGORY LIKE
'%PCs';
 THIS =
PRODUCTS[PROD_ID,PROD_NAME,PROD_CATEGORY_DESC,PROD_LIST_PRICE,PROD_MIN_PRICE];
 THIS[PRIORITY] = 2;
END

DEFINE PRIVATE DATASET CONSOLE_PRODUCTS_TMP
 ROWSOURCE PRODUCTS WHERE PRODUCTS.PROD_SUBCATEGORY_DESC LIKE '%Game%' AND
PRODUCTS.PROD_VALID = 'A';
 THIS =
PRODUCTS[PROD_ID,PROD_NAME,PROD_CATEGORY_DESC,PROD_LIST_PRICE,PROD_MIN_PRICE];
 THIS[PRIORITY] = 1;
END

DEFINE DATASET ENTERTAINMENT_PRODUCTS_C
 ROWSOURCE UNION[PC_PRODUCTS_TMP,CONSOLE_PRODUCTS_TMP];
 THIS = PC_PRODUCTS_TMP;
 PRIMARYKEY[PROD_ID];
END
```

INLINE Data Set

An inline table defines both its structure and its data, including a predefined set of records, directly within the table definition. Use inline tables to embed small datasets in your code without relying on external sources.

Example:

```
DEFINE INLINE DATASET MYINLINEDATA
 ROWSOURCE INTABLE([
    PROMO_CATEGORY_ID:VARCHAR2(128),
    PROMO_CATEGORY:VARCHAR2(60),
    PROMO_DISCOUNT_RATE:BIGDECIMAL(38, 12)]
```

```
    VALUES
       ([1 ,'No Promotion',0],
       [2 ,'Television',11.5],
       [3 ,'Internet', 15.4])
    );
PRIMARYKEY[PROMO_CATEGORY_ID];
END
```

> **Note**
>
> You could directly insert numeric values into a `VARCHAR2` column with or without enclosing them in quotes.

Column Mapping Assignment

Assign column mapping for column selections, properties, and transformations.

You can omit column mapping when there's only one input (in `ROWSOURCE`) and all input columns are selected.

Syntax

```
column_mapping_assignment ::= THIS [ column_list ] '=' { table_name
                                                       |
value_returned_expression }
                                                       [ '-' DATATYPE
data_type ]
                                                       [ '-' { INTERNAL |
VARIABLE } ] ;
```

See Value Returned Expressions.

- **INTERNAL**: A column with restricted access, hidden from users, that you can mainly use for debugging purposes.
- **VARIABLE**: A transient data holder within the dataset, that you can use for intermediate transformations because it's not included in the final output.

Example:

```
IMPORT SOURCE CUSTOMERS
DEFINE DATASET CUSTOMERS_D_COL
 ROWSOURCE CUSTOMERS;
 THIS[FROM_EFFECTIVE_DATE,TO_EFFECTIVE_DATE] =
CUSTOMERS[CUST_EFF_FROM,CUST_EFF_TO];
 THIS[CUST_MARITAL_STATUS] = COALESCE
(CUSTOMERS.CUST_MARITAL_STATUS,'UNKNOWN') -INTERNAL;
 THIS =
CUSTOMERS[CUST_ID,CUST_FIRST_NAME,CUST_LAST_NAME,CUST_CITY,COUNTRY_ID];
 THIS[CUST_FULL_NAME] = CONCAT_WS (' ',
THIS.CUST_FIRST_NAME,THIS.CUST_LAST_NAME);
 THIS[BIRTHYEAR] = CUSTOMERS[CUST_YEAR_OF_BIRTH] -VARIABLE;
 THIS[VOTED_AGE_FLAG] = CASE WHEN THIS.BIRTHYEAR > 2018 THEN 'Y' ELSE 'N' END
```

```
-DATATYPE VARCHAR2(1);
END
```

You can omit column mapping when there's only one input (in `ROWSOURCE`) and all input columns are selected.

Example: The following is an example of omitting column mapping:

```
IMPORT SOURCE CUSTOMERS
//THIS = CUSTOMERS is omitted
DEFINE DATASET CUSTOMERS_DO
 ROWSOURCE CUSTOMERS;
END
```

Default Row Specification

Default row specification requires you to define a predefined record, ensuring there is always a default entry in the table and maintaining schema integrity.

Syntax:

```
default_row_specification ::= DEFAULTROW default_column_list ;
default_column_list ::= '[' (THIS[<column_list>] '=' (PARAMETER '['
<parameter_name> ']' |
<constant_value>); )+ ']'
```

Example:

```
IMPORT SOURCE CUSTOMERS
/* not listed columns are assigned NULL */
DEFINE DATASET CUSTOMERS_DEFAULT
 ROWSOURCE CUSTOMERS;
 THIS = CUSTOMERS;
 DEFAULTROW
 [ THIS[CUST_ID] = -99999;
 THIS[CUST_FIRST_NAME] = 'Unknown';
 ]
 PRIMARYKEY[CUST_ID];
END
```

If you omit a column's default value, it defaults to `NULL`.

Aggregate Specification

Aggregate specification denotes a list of `group_by` columns when aggregation is part of the transformation.

Syntax:

```
aggregate_specification ::= GROUPBY column_list ;
```

Example:

```
IMPORT SOURCE CUSTOMERS
IMPORT SOURCE COUNTRIES

DEFINE VERSIONED DATASET GAMING_CUSTOMER_REACHED_G
 ROWSOURCE CUSTOMERS INNER JOIN COUNTRIES ON (CUSTOMERS.COUNTRY_ID =
COUNTRIES.COUNTRY_ID)
 THIS = COUNTRIES[COUNTRY_REGION,COUNTRY_SUBREGION];
 THIS[REACHED_VOLUME] = COUNT(CUSTOMERS.CUST_ID);
 GROUPBY[COUNTRY_REGION,COUNTRY_SUBREGION];
 PRIMARYKEY[COUNTRY_REGION,COUNTRY_SUBREGION];
END
```

Primary Key Specification

The primary key is optional and inferred from the input when there's only a single input. For `PRIVATE VERSIONED` datasets, which do not require `PRIMARYKEY`, you must explicitly define a primary key for a multi-input dataset.

Syntax:

```
primary_key_specification ::= PRIMARYKEY column_list ;
entity_id_specification ::= ENTITYID column_list ; (internal)
column_list ::= '[' column_name [, column_name]… ']'
```

Example:

```
IMPORT SOURCE CUSTOMERS
DEFINE DATASET CUSTOMERS_D
 ROWSOURCE CUSTOMERS;
 THIS = CUSTOMERS;
 PRIMARYKEY[CUST_ID];
END
```

Delete Specification and Soft Delete

This specification defines the driven dataset and matching key for deleting the target table.

Syntax:

```
delete_specification ::= [soft_delete_spec ] deletesource_specification
soft_delete_spec ::= DELETETYPE '[' SOFT [create_soft_delete_column] ']';
create_soft_delete_column ::= column_name
deletesource_specification ::= DELETESOURCE table_name  column_list MATCHING
column_list;
```

Example:

```
IMPORT SOURCE CUSTOMER_DELETE_LOG
IMPORT SOURCE CUSTOMERS
DEFINE DATASET CUSTOMERS_D
   ROWSOURCE CUSTOMER_HD;
```

```
   THIS = CUSTOMER_HD;

   DELETESOURCE CUSTOMER_DELETE_LOG[CUST_ID] MATCHING [CUST_ID];
END
```

You've defined the dataset `CUSTOMER_DELETE_LOG` to identify and delete the matched records in the dataset `CUSTOMERS_D`.

Incremental Refresh Directive

This directive defines how a dataset refreshes with the `UPDATEABLE` table type, using the last updated date (LUD) from the source table.

The incremental refresh directive identifies the incremental input for changes and is crucial for optimizing refresh efficiency in multi-input datasets.

Syntax:

```
incremental_refresh_directive ::= REFRESH ON {CHANGES|UPSERTS|DELETES} IN
table_reference_list ;
table_reference_list ::= '[' table_reference [, <table_reference]… ']'
table_reference ::= table_name | table_alias
```

The incremental refresh directives are as follows:

- **CHANGES**: Supports both upserts (updates) and deletes.
- **UPSERTS**: Supports only upserts (updates); no deletes.
- **DELETES**: Supports only deletes.

Example:

```
IMPORT SOURCE SALES
IMPORT SOURCE PRODUCTS
DEFINE UPDATEABLE DATASET SALES_FACT_OJ
    ROWSOURCE SALES INNER JOIN PRODUCTS ON SALES.PROD_ID = PRODUCTS.PROD_ID;
    THIS = SALES;
    THIS[PROD_NAME] = PRODUCTS.PROD_NAME;
    PRIMARYKEY [CUST_ID,PROD_ID,PROMO_ID,CHANNEL_ID,TIME_ID];
    REFRESH ON CHANGES IN [SALES];
END
```

See Incremental.

Code Block Load - Full and Incremental Load Instructions

When the full load transformation logic is separate from subsequent loads, the dataset definition has separate instructions for each load type.

> **Note**
>
> - You're required to enter the full Load block.
> - Without an Incremental Load block, the dataset stops refreshing after the full load.
> - The `VIEW` or `INLINE` table types don't support `code_block_load`.

Syntax:

```
code_block_load ::= ON FULL LOAD
                    rowsource_specification
                    [column_mapping_assignment]...
                    [ INCREMENTAL LOAD
                     rowsource_specification
                     [column_mapping_assignment]...
                    ]
                    ENDLOAD
                    [ default_row_specification ]
                    [ aggregate_specification ]
                    [ primary_key_specification ]
                    [ entity_id_specification ]
                    [ incremental_refresh_directive ]
                    [ delete_specification ]...
```

Example:

```
IMPORT SOURCE CUSTOMERS
DEFINE DATASET CUSTOMERS_D_FULL
    ON FULL LOAD
        ROWSOURCE CUSTOMERS WHERE CUSTOMERS.CUST_VALID = 'A';
    INCREMENTAL LOAD
        ROWSOURCE CUSTOMERS;
    ENDLOAD
END
```

FROM - Compact Format

`FROM` is a compact form for dataset definition instruction that you can use to specify sources and select input columns.

While `ROWSOURCE` offers more flexibility for generating transformation input, you can use `FROM` to simply list the instructions in the same line.

Syntax:

```
from_clause ::= FROM [ table_source_type ] { table_name | table_name
column_list | table_name EXCLUDE column_list }
column_list ::= '[' column_name [ ',' column_name ]… ']'
```

Example:

```
IMPORT SOURCE CUSTOMERS
DEFINE DATASET CUSTOMERS_D FROM CUSTOMERS
[CUST_ID,CUST_FIRST_NAME,CUST_LAST_NAME,CUST_CITY,CUST_YEAR_OF_BIRTH,COUNTRY_I
D] END
```

AGGREGATIONONLY

AGGREGATIONONLY is a table type in dataset definitions, designed to store only aggregated data.

It ensures that the dataset contains a single row of summarized values rather than raw transactional records. Each column must use an aggregation function, such as SUM or AVG, to derive its values.

Key Features

- **No Primary Key Required** : Aggregation-only datasets contain a single row of summarized data, eliminating the need for a primary key.
- **Mandatory Aggregation**: Columns in these datasets must use aggregation functions (Example `SUM()`, `AVG()`).
- **No Group-By Allowed**: If `GROUPBY` is specified, the dataset is treated as a regular dataset instead of aggregation-only.

Example

```
DEFINE AGGREGATIONONLY DATASET SALES_AGG
 ROWSOURCE SALES WHERE SALES.QUANTITY_SOLD > 10 ;
 THIS[AVG_SALES_AMT] = AVG(SALES[AMOUNT_SOLD]);
END
```

Multiple Aggregated Metrics

An aggregation-only dataset can store multiple metrics, such as Average, Sum, and Min.

Example:

```
DEFINE AGGREGATIONONLY DATASET SALES_AGG1
    ROWSOURCE SALES WHERE SALES.QUANTITY_SOLD > 10;
    THIS[AVG_SALES_AMT] = AVG(SALES[AMOUNT_SOLD]);
    THIS[SUM_SALES_AMT] = SUM(SALES[AMOUNT_SOLD]);
    THIS[MIN_SALES_AMT] = MIN(SALES[AMOUNT_SOLD]);
    THIS[MAX_SALES_AMT] = MAX(SALES[AMOUNT_SOLD]);
END
```

Full and Incremental Loads in Aggregation-Only Datasets

Aggregation-only datasets support code-block-load (both full loads and incremental loads). See Code Block Load - Full and Incremental Load Instructions .

Example:

```
DEFINE AGGREGATIONONLY DATASET SALES_AGG2
    ON FULL LOAD
```

```
        ROWSOURCE SALES WHERE SALES.QUANTITY_SOLD > 20;
        THIS[AVG_SALES_AMT] = AVG(SALES[AMOUNT_SOLD]);
        THIS[SUM_SALES_AMT] = SUM(SALES[AMOUNT_SOLD]);
        THIS[MIN_SALES_AMT] = MIN(SALES[AMOUNT_SOLD]);
        THIS[MAX_SALES_AMT] = MAX(SALES[AMOUNT_SOLD]);
    INCREMENTAL LOAD
        ROWSOURCE SALES WHERE SALES.QUANTITY_SOLD > 10;
        THIS[AVG_SALES_AMT] = AVG(SALES[AMOUNT_SOLD]);
       THIS[SUM_SALES_AMT] = SUM(SALES[AMOUNT_SOLD]);
        THIS[MIN_SALES_AMT] = MIN(SALES[AMOUNT_SOLD]);
        THIS[MAX_SALES_AMT] = MAX(SALES[AMOUNT_SOLD]);
    ENDLOAD
END
```

In this example:

- **Full Load** uses SALES with QUANTITY_SOLD > 20, where all the records are truncated and reloaded.
- **Incremental Load** uses SALES, with QUANTITY_SOLD > 10.

Derived Aggregation-Only Datasets

To simplify calculations, you can derive aggregation-only datasets from other aggregation-only datasets.

Example:

```
DEFINE AGGREGATIONONLY DATASET SALES_AGG3
 ROWSOURCE SALES_AGG2;
 THIS = SALES_AGG2.AVG_SALES_AMT;
END
```

> **Note**
>
> - You must mark a dataset that;s created exclusively from an aggregation-only source as `AGGREGATIONONLY`.
> - CROSS JOIN is the only join type allowed with aggregation-only inputs.
> - The `Refresh On Changes` directive is not allowed on aggregation-only dataset.

Joining Aggregation-Only Datasets with Transactional Data

You can combine aggregation-only datasets with transactional data using `CROSS JOIN`.

Example:

```
DEFINE DATASET SALES_AGG4
 ROWSOURCE SALES CROSS JOIN SALES_AGG_DERIVED WHERE SALES.QUANTITY_SOLD > 10;
 THIS = SALES_AGG_DERIVED;
 THIS = SALES;
 PRIMARYKEY[PROD_ID,CHANNEL_ID,CUST_ID,TIME_ID,PROMO_ID];
 REFRESH ON CHANGES IN[SALES];
END
```

> **Note**
>
> - CROSS JOIN is the only join type allowed with aggregation-only inputs.
> - The `Refresh On Changes` directive is not allowed on aggregation-only dataset.

Deletions

You apply deletions during subsequent loads to reflect removals from the source (tracked or audited) or to clean up as explicitly instructed.

Deletion propagation includes:

- **Implicit Propagation**: Deletions flow through the lineage to downstream datasets.
- **Explicit Prevention**: Deletions are restricted from propagating.

Topics include:

Track Upstream Removals

`TRACKDELETES`: Automatically detects removals by comparing the current dataset with the original upstream copy.

Syntax:

See IMPORT Statement.

You can track removals in:

- The imported source itself (the default behavior).
- A named track set, that's tracked separately with a unique name.

Example:

```
IMPORT SOURCE SALES WITH TRACKDELETES
IMPORT SOURCE SALES WITH TRACKDELETES IN [SALESREMOVALS] AS SALES_DEL1 //
Named Track Set
```

> **Note**
>
> If the upstream system already tracks or audits deletions, you can leverage that audit to improve performance and reduce the cost of comparing and tracking.

Hard Deletions

You can permanently delete records from data warehouse datasets through propagation or explicit application.

You can apply deletions:

- **With Tracked Imported Source (default behavior): TRACKDELETES**
  Example: Propagate automatically in SALES_F :

```
IMPORT SOURCE SALES WITH TRACKDELETES
DEFINE DATASET SALES_F FROM SALES END
```

- **Using a Named Tracked Set**
  You can apply this to both the imported source and the dataset by using:
  - **Then Delete**: Explicit directive that you can apply to an Imported Source in IMPORT SOURCE statement.
    See IMPORT Statement.

    Example:

```
IMPORT SOURCE DEL_SALES //delete records
//Clean up SALES with DEL_SALES
IMPORT SOURCE SALES THEN DELETE [DEL_SALES [SALES_ID] MATCHING
[SALES_ID]]

// Delete for keys in DEL_SALES are automatically propagated to
NET_SALES_F
DEFINE DATASET NET_SALES_F FROM SALES END
```

  - **Delete Source**: Explicit deletion that you can apply to a dataset in the Dataset Definition.
    The delete set (substraend) can be a delete named track set or can be any dataset.

    Syntax: See Generic Dataset Definition.

> **Note**
>
> DELETESOURCE overrides all propagation for that dataset.

    Example: Subtraend from TRACKDELETES:

```
IMPORT SOURCE SALES WITH TRACKDELETES IN [REMOVALS]
DEFINE DATASET SALES_F
   ROWSOURCE SALES;
   THIS = SALES;
   DELETESOURCE REMOVALS [SALES_ID] MATCHING [SALES_ID]
END
```

    Example: Subtraend from imported source, using DELETESOURCE in the dataset:

```
IMPORT SOURCE SALES_REMOVALS //Audited from upstream, no tracking
IMPORT SOURCE SALES
DEFINE DATASET SALES_F
   ROWSOURCE SALES;
   THIS = SALES;
   DELETESOURCE SALES_REMOVALS [SALES_ID] MATCHING [SALES_ID];
END
```

Soft Deletions

Instead of being physically removed, Data Augmentation Scripts flags removal records as deleted. The flag name defaults to `ISDELETED` if you don't specify otherwise.

> **Note**
>
> - You must filter out soft-deleted records when necessary from queries and downstream datasets.
> - Soft deletes retain data but require efficient query filtering to maintain system efficiency.

Syntax: See IMPORT Statement.

Similar to hard deletes, soft deletes utilize `TRACKDELETE`, `THEN DELETE`, and `DELETESOURCE`.

- **TRACKDELETE and Soft Delete**
  The following are examples of soft delete on importing.

  Example: flag name not specified (defaulted) with `TRACKDELETES`.

  ```
  IMPORT SOURCE CUSTOMERS DELETETYPE[ SOFT ] WITH PRIMARYKEY[CUST_ID]
  TRACKDELETES
  DEFINE UPDATEABLE DATASET CUSTOMERS_SD_D FROM CUSTOMERS END
  ```

  Example: flag name provided

  ```
  IMPORT SOURCE SALES DELETETYPE[ SOFT[ISTRANDELETED] ] TRACKDELETES[ IN
  [THDELETE] ]
  DEFINE UPDATEABLE DATASET CUSTOMERS_SD_D FROM CUSTOMERS END
  ```

- **THEN DELETE and Soft Delete**
  Example: Soft delete on importing, using other imported data.

  ```
  IMPORT SOURCE SALESDEL
  IMPORT SOURCE SALES DELETETYPE[ SOFT ] THEN DELETE [ SALESDEL [SALES_ID]
  MATCHING [SALES_ID] ]
  DEFINE DATASET SALES_SD_F FROM SALES END
  ```

- **DELETESOURCE and Soft Delete**
  Syntax: See Generic Dataset Definition.

  Example: Soft delete on dataset.

  ```
  IMPORT SOURCE SALESDEL
  IMPORT SOURCE SALES
  DEFINE UPDATEABLE DATASET SALES_F
      ROWSOURCE SALES;
      THIS = SALES;
      DELETETYPE[ SOFT[SALESDELETED] ];
      DELETESOURCE SALESDEL[SALE_ID] MATCHING [SALE_ID];
  END
  ```

Propagation Control

Propagation control consists of:

- **DISABLE DELETEPROPAGATION**
  You can use disable `DELETEPROPAGATION` to prevent deletions from cascading to downstream datasets.
  - **Default Behavior**: By default, Data Augmentation Scripts applies removals to downstream datasets, unless you define propagation.
  - **Disabled Propagation**: Data Augmentation Scripts doesn't propagate deletions automatically, but still applies explicit deletions that use `DELETESOURCE`.

  Syntax:

  ```
  disable_delete_propagation ::= DISABLE DELETEPROPAGATION FOR { ALL
  DATASETS | DATASETS '[' table_name ']' }
  ```

  Example: Disable delete propagation for all datasets.

  ```
  DISABLE DELETEPROPAGATION FOR ALL DATASETS
  ```

  Example: Disable delete propagation for a product dimension dataset.

  ```
  DISABLE DELETEPROPAGATION FOR DATASETS[PRODUCTS_D]
  ```

- **REFRESH ON DELETES**
  You can controls how deletions impact the dataset.

  Syntax: See Generic Dataset Definition.

  Example: In the following example, for incremental runs, only Sales deletes are considered.

  ```
  IMPORT SOURCE SALES WITH TRACKDELETES
  IMPORT SOURCE PRODUCTS
  DEFINE DATASET SALES_F
     ROWSOURCE PRODUCTS  INNER JOIN SALES  ON SALES.PROD_ID =
  PRODUCTS.PROD_ID;
     THIS = SALES;
     REFRESH ON DELETES IN[SALES];
     REFRESH ON UPSERTS IN[PRODUCTS]
  END
  ```

- **STABILITYPERIOD**
  You can restrict processing data changes within a defined timeframe (or sliding window) since the previous load. You can apply the restriction to either the record's Initial Extract Date (IED) or Last Updated Date (LUD) of upstream data.

  Syntax: See IMPORT Statement.

  The following is the change tracking behavior based on Last Updated Date (LUD):

  - If you haven't identified a LUD for the source, only changes to records created in the last n days at extract time are tracked.

– If you've identified a LUD defined for the source, both changes since the last extract and changes to records created in the last n days at extract time are tracked.

The following example specifies the Initial Extract Date- (IED) but not a Last Updated Date (LUD) :

```
IMPORT SOURCE CUSTOMERS WITH IED [CUST_EFF_FROM] STABILITYPERIOD[30,
CUST_EFF_FROM] TRACKDELETES
DEFINE DATASET CUSTOMERS_D
   ROWSOURCE CUSTOMERS;
   THIS = CUSTOMERS;
END
```

The example code specifies that only changes (including deletions) in SALES from the last 30 days be processed through propagation in each load.

> **Note**
>
> Even when you don't explicitly define the Last Updated Date (LUD) column in an IMPORT definition, if an incremental key is defined in the source metadata, Data Augmentation Scripts automatically uses it as the Last Updated Date (LUD). To override this behavior, you can use LUD[NULL].

The following example ignores the Last Updated Date (LUD):

```
IMPORT SOURCE CUSTOMERS WITH IED [CUST_EFF_FROM] LUD[null]
STABILITYPERIOD[30, CUST_EFF_FROM] TRACKDELETES
```

Choose the Best Deletion Strategy

The following table maps a deletion scenario with the corresponding recommended deletion method:

Scenario	Recommended Deletion Method
Data must be completely removed.	Hard delete (`THEN DELETE`, `DELETESOURCE`, `TRACKDELETES`)
Need to retain deleted records for historical tracking.	Soft delete (`ISDELETED` flag)
Source system doesn't track deletions.	`TRACKDELETES`
Deletions must be explicitly provided by a named dataset.	`DELETESOURCE`
Avoid cascading deletions in downstream datasets.	`DISABLE DELETEPROPAGATION`
Optimize performance by applying necessary deletions on imported sources	`THEN DELETE`

Schema Definition

Schema Definition provides control over dataset structures while maintaining flexibility.

Schema definitions have the following characteristics:

- Allows predefined or overridden data types and formats for datasets.

- Provides uniformity across multiple datasets while permitting necessary adjustments.
- Enables you to define a schema separately as a template or embedded within a dataset as an inline statement.
- Allows you to reference by name within a dataset when you've a schema defined separately.
- Becomes part of a code block when you've embedded it. See Code Block.

Syntax

For information on schema statements (`template_schema_statement` or `builtin_schema_statement`) in the dataset definition `code_block`, see Generic Dataset Definition.

Schema Definition Rules

- Column data types are mandatory; `PRIMARYKEY` is optional.
- By default, all columns are nullable unless you explicitly specify otherwise.
- If you don't define `PRIMARYKEY` in the schema, it must be provided by the source dataset.

Conflict Resolution

- **Primary Key**
  - When you define `PRIMARYKEY` in both the schema and the dataset, the dataset value takes precedence.
  - For datasets with row source joins, you can omit `PRIMARYKEY` from the dataset if you've already specified it in the schema.
- **Data Type**
  - When you define a column's data type in both the schema and the dataset, the data type that you provide in the dataset overrides the dataset in the schema.

Column Mismatch Handling

When a column is present in either the schema or the dataset but not in both, its inclusion and properties are determined as follows:

- **Extra Columns in the Dataset**: If a column appears in the dataset but not in the schema, its properties are derived from the source.
- **Extra Schema Columns**: If you define a column in the schema but don't map it in the dataset, it's ignored unless it's part of the primary key. If so, a warning is issued.

Schema Template Definition

You can use `DEFINE SCHEMA`, the dataset definition code block, to define a schema separately from a dataset definition, refer to it, and apply it to any dataset.

Syntax:

```
schema_definition ::= DEFINE SCHEMA schema_name
                                                    '[ '
                                                      column_name data_type
[ PRIMARYKEY ] [ nullable_flag ]
                                                     [,column_name data_type
[ PRIMARYKEY ] [ nullable_flag ]] ...
                                                    ']'
```

```
                                                        END
schema_name ::= identifier
template_schema_statement ::= SCHEMA schema_name ; //used in Dataset
Definition
```

Example of a dataset defined with the `template_schema_statement`:

```
IMPORT SOURCE CUSTOMERS
DEFINE SCHEMA CUSTOMERS_D_SCHEMA
    [
    CUST_ID NUMBER(38,0) PRIMARYKEY,
    CUST_LAST_NAME VARCHAR2(32),
    CUST_CITY_ID NUMBER(38,0),
    CUST_VALID VARCHAR2(32),
    CUST_EFF_FROM DATE NOT NULL,
    CUST_EFF_TO DATE
    ]
END
//Usage in Dataset Definition
DEFINE DATASET CUSTOMERS_D
 SCHEMA CUSTOMERS_D_SCHEMA;
 ROWSOURCE CUSTOMERS;
 THIS = CUSTOMERS;
END
```

Inline Schema Definition

Within the dataset definition code block, you can define a schema directly within a dataset definition, applying it only to that specific dataset.

Syntax:

```
builtin_schema_statement ::= SCHEMA
                                                          '[ '
                                                             column_name
data_type [ PRIMARYKEY ] [ nullable_flag ]
                                                             [,column_name
data_type [ PRIMARYKEY ] [ nullable_flag ]] ...
                                                           ']' ;
```

Example of a dataset defined with the `builtin_schema_statement`:

```
DEFINE DATASET INS_CUSTOMERS_D
    SCHEMA
    [
        CUST_ID          NUMBER(38,0)    PRIMARYKEY,
        CUST_LAST_NAME.  VARCHAR2(32),
        CUST_CITY_ID     NUMBER(38,0),
        CUST_VALID       VARCHAR2(32),
        CUST_EFF_FROM    DATE          NOT NULL,
        CUST_EFF_TO      DATE
    ];

    ROWSOURCE CUSTOMERS;
```

```
    THIS = CUSTOMERS;
END
```

Code Block Load and Schema Definition

When you separate the full load code from the incremental load within a dataset definition, the inline schema is defined at the beginning of the `code_block_load`.

Example of a `code_block_load` defined with `template_schema_statement`:

```
DEFINE DATASET FL_CUSTOMERS_D
    SCHEMA CUSTOMERS_D_SCHEMA;
    ON FULL LOAD
        ROWSOURCE CUSTOMERS WHERE CUSTOMERS.CUST_VALID = 'A';
        THIS = CUSTOMERS;
    INCREMENTAL LOAD
        ROWSOURCE CUSTOMERS;
        THIS = CUSTOMERS;
    ENDLOAD
END
```

Example of a `code_block_load` defined with `builtin_schema_statement`:

```
DEFINE VERSIONED DATASET INSCH_FL_CUSTOMERS_CF
    SCHEMA
    [
        CUST_ID          NUMBER(38,0)    PRIMARYKEY,
        CUST_LAST_NAME   VARCHAR2(32),
        CUST_CITY_ID     NUMBER(38,0),
        CUST_VALID       VARCHAR2(32),
        CUST_EFF_FROM    DATE          NOT NULL,
        CUST_EFF_TO      DATE
    ];
    ON FULL LOAD
        ROWSOURCE CUSTOMERS WHERE CUSTOMERS.CUST_VALID  = 'A';
        THIS = CUSTOMERS;
    INCREMENTAL LOAD
        ROWSOURCE CUSTOMERS;
        THIS = CUSTOMERS;
    ENDLOAD
END
```

Transposition Table Definition

You can use a Transposition Table Definition to restructure datasets by rotating rows into columns (PIVOT) or columns into rows (UNPIVOT).

Syntax

```
transposition_table_definition ::= DEFINE [export_specification]
[transposition_table_type] DATASET table_name
                                  { transposition_code_block |
pivot_code_block_load | unpivot_code_block_load }
```

```
                                    END

transposition_table_type ::= VERSIONED
```

For more information, see Export Specification.

Common Elements

```
transposition_rowsource_specification ::= ROWSOURCE
{ transposition_table_source | inline_source };

transposition_table_source ::= table_reference [ row_filtering_conditional ]

segment ::= WITHIN table_name [EXCLUDE] column_list;
```

Transposition Code Block

The transposition code block contains either a PIVOT or UNPIVOT block, which determines how the data is transformed.

```
transposition_code_block ::= pivot_code_block | unpivot_code_block
```

See:

- Pivot Definition
- Unpivot Definition

Pivot Definition

```
pivot_code_block ::=
                   transposition_rowsource_specification
                   pivot_section
                   [ column_mapping_assignment ]...
                   [ primary_key_specification ]
```

> **Note**
>
> You're allowed to only map columns for columns generated from PIVOT section.

```
pivot_section ::= PIVOT
                                          '('
                                             [ segment ]
                                             pivot_transposition...
                                          ')'

pivot_transposition ::= THIS [ column_list ] '=' { pivot_function_list |
pivot_scalar_list }
                        FOR { table_name.column_name IN
constant_value_expression | table_column_tuple IN
constant_value_tuple_expression }
```

```
pivot_function_list ::= '[' aggregate_function [ column_prefix ] [,
aggregate_function [ column_prefix ] ] ... ']'

pivot_scalar_list ::= '[' column_name [ column_prefix ] [, column_name
[ column_prefix ] ] ... ']'
column_prefix ::= '-' COLPREFIX string

table_column_tuple ::= '(' table_name.column_name [,
table_name.column_name] ... ')'

constant_value_expression ::= '(' constant_value [, constant_value] ... ')'

constant_value_tuple_expression ::= '(' constant_value_tuple [,
constant_value_tuple] ... ')'
constant_value_tuple ::= '(' constant_value, constant_value [,
constant_value] ... ')'
```

Note

- If a `WITHIN` statement is skipped, all the columns that aren't used in transpositions will be used to partition (group) the data.
- Within the `PIVOT` call, transpositions could be aggregated or scalar. If you use aggregation, then all transposition statements must have aggregation.
- If column names on the left side aren't explicitly mentioned, `-COLPREFIX` property is mandatory for aggregation-based transpositions and optional for scalar transpositions.
- `ROWSOURCE` can't contain joins or unions but it can contain filters.
- Other transformations (except aggregations) are allowed after the `PIVOT` block and only by using the columns from `PIVOT` section.
- No `GROUP BY` is allowed in the dataset.
- Primary key is optional. If optional, the within clause columns are considered as primary key. Primary key can be specified, especially for excluding functionally-dependent columns.

Examples

Calculate average income for each month using country and city.

```
DEFINE VERSIONED DATASET DW_CITY_PIVOT
    ROWSOURCE CITIES;

    PIVOT
    (
        /* ------------------------ SPECIFY PARTITION
------------------------- */
        WITHIN CITIES[COUNTRY, CITY, CITY_CODE];

        /* ------------------------SPECIFY TRANSPOSITIONS
---------------------- */
```

```
        // Provide target column names on LHS for each of the month values
        THIS[AV_INC_JAN, AV_INC_FEB, AV_INC_MAR] = AVG(CITIES.INCOME) FOR
CITIES.MONTH IN ('Jan', 'Feb', 'Mar');
    );

    PRIMARYKEY [COUNTRY, CITY];
END
```

Calculate total population and average population for each year using country and city.

```
DEFINE VERSIONED DATASET DW_CITY_PIVOT
    ROWSOURCE CITIES;

    PIVOT
    (
        WITHIN CITIES[COUNTRY, CITY, CITY_CODE];

        // Target column names are generated using -COLPREFIX property. The
YEAR values are suffixed to -COLPREFIX using underscore(_).
        THIS = [ SUM(CITIES.POPULATION) -COLPREFIX 'SUM_POP',
AVG(CITIES.POPULATION) -COLPREFIX 'AVG_POP' ]
                      FOR CITIES.YEAR IN (2000, 2010, 2020);

         // E.g. Generated columns SUM_POP_2000, AVG_POP_2000,
SUM_POP_2010 , ...
    );

    THIS[SUM_CHANGE_2010_2020] = THIS.SUM_POP_2000/THIS.SUM_POP_2010;

    PRIMARYKEY [COUNTRY, CITY];
END
```

Calculate total population for each (year, month) combination using country and city.

```
DEFINE VERSIONED DATASET DW_CITY_PIVOT
    ROWSOURCE CITIES;

    PIVOT
    (
        WITHIN CITIES[COUNTRY, CITY, CITY_CODE];

        // Multiple column combinations.
        THIS = SUM(CITIES.POPULATION) -COLPREFIX 'SUM_POP'
                    FOR (CITIES.YEAR, CITIES.MONTH) IN (
                                                          (2000,'Jan'),
(2000,'Feb'), (2000,'Mar'),
                                                          (2010,'Feb'),
(2010,'Mar'),
                                                          (2020,'Jan'),
(2020,'Feb'), (2020,'Mar')
                                                       );

       // Column name examples: SUM_POP_2000_JAN, SUM_POP_2000_FEB
    );
```

```
    PRIMARYKEY [COUNTRY, CITY];
END
```

Non-aggregation pivot

```
DEFINE VERSIONED DATASET DW_CUSTOMERS_PIVOT
    ROWSOURCE CUSTOMERS;

    PIVOT
    (
        WITHIN CUSTOMERS[ID];

        // For non-aggregation pivot, -COLPREFIX is optional.
        // In this case, use the values from IN clause as column names.
       THIS = CUSTOMERS.AttributeValue FOR CUSTOMERS.Attribute IN
('FirstName', 'LastName', 'DOB');
    );

    PRIMARYKEY [ID];
END
```

Combine all PIVOTs in a dataset

```
DEFINE VERSIONED DATASET DW_CITY_PIVOT
    ROWSOURCE CITIES;

    PIVOT
    (
        /* ------------------------- SPECIFY PARTITION
------------------------- */
        WITHIN CITIES[COUNTRY, CITY, CITY_CODE];
        // or WITHIN ALL;


        /* ----------------------SPECIFY TRANSPOSITIONS
---------------------- */
        // Provide target column names on LHS for each of the month values
        THIS[AV_INC_JAN, AV_INC_FEB, AV_INC_MAR] = AVG(CITIES.INCOME) FOR
CITIES.MONTH IN ('Jan', 'Feb', 'Mar');

        THIS =  [ SUM(CITIES.POPULATION) -COLPREFIX 'SUM_POP',
AVG(CITIES.POPULATION) -COLPREFIX 'AVG_POP' ] FOR CITIES.YEAR IN (2000, 2010,
2020);

        // Multiple column combinations
        THIS = [SUM(CITIES.POPULATION) -COLPREFIX 'SUM_POP']
                  FOR (CITIES.YEAR, CITIES.MONTH) IN (
                                                       (2000,'Jan'),
(2000,'Feb'), (2000,'Mar'),
                                                       (2010,'Feb'),
(2010,'Mar'),
                                                       (2020,'Jan'),
(2020,'Feb'), (2020,'Mar')
                                                      );
    );
```

```
    THIS[SUM_CHANGE_2010_2020] = THIS.SUM_POP_2000/THIS.SUM_POP_2010;

    PRIMARYKEY [COUNTRY, CITY];
END
```

Unpivot Definition

```
unpivot_code_block ::= transposition_rowsource_specification
                       unpivot_section
                       [ column_mapping_assignment ] ...
                       primary_key_specification
```

> **Note**
>
> You're allowed to assign columns only for columns generated from the UNPIVOT section.

```
unpivot_section ::= UNPIVOT [ INCLUDE NULLS ]
                    '('
                        [ segment ]
                        unpivot_transposition ...
                    ')';

unpivot_transposition ::= THIS column_tuple_list '=' table_name column_list ;
column_tuple_list ::= '[' column_tuple [, column_tuple ]... ']'
column_tuple ::= '(' column_key_value [, column_key_value ]... , identifier
')'
column_key_value ::= identifier ':' constant_value
```

> **Note**
>
> - The `WITHIN` statement is optional. If it's skipped, all remaining columns that aren't used in transpositions must be used in partition.
> - `NULL` values are excluded by default. You can include them by explicitly using `INCLUDE NULLS`.
> - Aggregation isn't allowed in the `UNPIVOT` block.
> - `ROWSOURCE` can't contain joins or unions but it can contain filters. Other transformations (except aggregations) are allowed after the `UNPIVOT` section and only by using the columns from the `UNPIVOT` section.
> - No `GROUP BY` is allowed in the dataset.
> - Primary key is mandatory and must include the columns from the `WITHIN` statement (it can exclude functionally dependent columns) and the key column from the `UNPIVOT` assignment.

Examples

Show product and sales amount as (name, value) pairs using id and fiscal year.

```
DEFINE VERSIONED DATASET DW_SALES_UNPIVOT
    ROWSOURCE SALES;

    UNPIVOT INCLUDE NULLS // Nulls can be included/excluded. Exclude, by
default
    (
        WITHIN SALES[ID, FISCAL_YEAR];

        // Target Columns SALES_AMT and PRODUCT are specified on LHS
        // The corresponding display values for PRODUCT column are specified
in the LHS as well
        // Pairs on LHS map to columns on RHS, in sequence
        THIS[(PRODUCT: 'A', SALES_AMT), (PRODUCT : 'B', SALES_AMT)] =
SALES[PROD_A_AMT, PROD_B_AMT];

    );

    PRIMARYKEY [ID, FISCAL_YEAR, PRODUCT];
END
```

Show product, sales amount, sales quantity using ID and fiscal year.

```
DEFINE VERSIONED DATASET DW_SALES_UNPIVOT
    ROWSOURCE SALES;

    UNPIVOT INCLUDE NULLS
    (
        WITHIN SALES[ID, FISCAL_YEAR];
         THIS[(PRODUCT : 'A', SALES_AMT), (PRODUCT :'B', SALES_AMT),
(PRODUCT : 'A', SALES_QTY), (PRODUCT : 'B', SALES_QTY)]
              = SALES[PROD_A_AMT, PROD_B_AMT, PROD_A_QTY, PROD_B_QTY];
    );

    THIS[SALES_RATIO] = THIS.SALES_AMT/THIS.SALES_QTY;

    PRIMARYKEY [ID, FISCAL_YEAR,PRODUCT];
END
```

Show year, month and population using country and city.

```
DEFINE VERSIONED DATASET DW_SALES_UNPIVOT
    ROWSOURCE SALES;
    UNPIVOT
    (
        WITHIN SALES[COUNTRY, CITY, CITY_CODE];

        THIS[(YEAR : 2000, MONTH : 'Jan', POPULATION), (YEAR : 2000, MONTH :
'Feb', POPULATION), (YEAR : 2000, MONTH : 'Mar', POPULATION)] =
SALES[SUM_POP_2000_JAN, SUM_POP_2000_FEB, SUM_POP_2000_MAR];
    );
```

```
    PRIMARYKEY [COUNTRY, CITY, YEAR, MONTH];
END
```

Block Load Definitions

Use block load for different load logic for full and incremental.

Pivot Load

```
pivot_code_block_load ::= 'ON FULL LOAD'
                              transposition_rowsource_specification
                              pivot_section
                              [ column_mapping_assignment ]...
                              [
                                  'INCREMENTAL LOAD'
                                  transposition_rowsource_specification
                                  pivot_section
                                  [ column_mapping_assignment ]...
                              ]
                              'ENDLOAD'
                              [ primary_key_specification ]
```

Example of Pivot Code block load:

```
DEFINE VERSIONED DATASET DW_CITY_PIVOT
    ON FULL LOAD
        ROWSOURCE CITIES;
        PIVOT
        (
            WITHIN CITIES[COUNTRY, CITY, CITY_CODE];
            THIS[AV_INC_JAN, AV_INC_FEB, AV_INC_MAR] = AVG(CITIES.INCOME) FOR
CITIES.MONTH IN ('Jan', 'Feb', 'Mar');
        );

    INCREMENTAL LOAD
        ROWSOURCE CITIES_COUNTIES;
        PIVOT
        (
            WITHIN CITIES_COUNTIES[COUNTRY, CITY, CITY_CODE];
            THIS[AV_INC_JAN, AV_INC_FEB, AV_INC_MAR] =
AVG(CITIES_COUNTIES.INCOME) FOR CITIES_COUNTIES.MONTH IN ('Jan', 'Feb',
'Mar');
        );
    ENDLOAD

    PRIMARYKEY [COUNTRY, CITY];
END
```

Unpivot Load

```
unpivot_code_block_load ::= 'ON FULL LOAD'
                                transposition_rowsource_specification
```

```
    unpivot_section
       [ column_mapping_assignment ]...
      [
       'INCREMENTAL LOAD'
       transposition_rowsource_specification
       unpivot_section
       [ column_mapping_assignment ]...
                                      ]
'ENDLOAD'
primary_key_specification
```

User Defined Functions (UDFs) or Macros

User-Defined Functions (UDFs) or macros in Data Augmentation Scripts enable you to create custom functions, which complement the built-in functions provided by Data Augmentation Scripts.

Syntax

```
macro_declaration ::= DEFINE FUNCTION macro_name '('
formal_parameter[,formal_parameter]... ')'

value_returned_expression
                                          END
```

Example 1: The following UDF changes the date column format to `yyyyMMdd`:

```
DEFINE FUNCTION formatDate(col)
 DATE_FORMAT(col, 'yyyyMMdd')
END
```

Example 2: The following UDF demonstrates the usage of multiple input parameters:

```
DEFINE FUNCTION getCurrencyRate(currency, currencyRate)
 CASE WHEN currency = $VAR_PARAM_GLOBAL_CURRENCY$ THEN 1 ELSE currencyRate END
END
```

Using a UDF

To use UDFs in Data Augmentation Scripts, you must first define the function and then call the defined function.

You can define UDFs in two ways:

- **Externally**: You can define UDFs in the `.func` file and then call in the current locode application by including the `.func` file using the `include_definition`.
- **Internally**: You can define UDFs within the locode program itself

You can call UDFs in the following ways:

- UDF call on single column:

```
 macro_call ::= macro_name '(' { table_name.column_name | table-name
[column_name] | THIS.column_name | THIS [column_name] } ')'
```

- UDF call on a list of columns:

```
macro-call ::= macro_name '(' variable_name ')' FOR variable_name IN table-
name [EXCLUDE] '[' column_name [, column_name]… ']'
```

> **Note**
>
> - Data Augmentation Scripts supports a UDF call on a list of columns for single parameter UDF functions.
> - The `EXCLUDE` option in UDF allows you to exclude specific elements of a dataset when applying the UDF.

Examples

- **UDF defined externally**
  You first define UDF in the `myFunctions.func` file within the Data Augmentation Scripts Project directory, and it's later used in the `main.hrf file`.

  `myFunctions.func`:

```
DEFINE FUNCTION toVc(col)
   CAST(col AS VARCHAR2(400))
END
```

  `main.hrf`:

```
INCLUDE FUNCTION "myFunctions.func"

DEFINE VERSIONED DATASET CUSTOMERS_D
   ROWSOURCE CUSTOMERS WHERE CUSTOMERS.CUST_ID=1;
   THIS = CUSTOMERS;
   THIS[CUST_POSTAL_CODE_STRING] = toVc(CUSTOMERS.CUST_POSTAL_CODE);
   PRIMARYKEY[CUST_ID];
END
```

- **UDF defined internally**
  You can define UDFs within the Data Augmentation Scripts program in the `.hrf` itself.

```
DEFINE FUNCTION toVc(col)
   CAST(col AS VARCHAR2(400))
END

DEFINE VERSIONED DATASET CUSTOMERS_D
   ROWSOURCE CUSTOMERS WHERE CUSTOMERS.CUST_ID=1;
   THIS = CUSTOMERS;
   THIS[CUST_POSTAL_CODE_STRING] = toVc(CUSTOMERS.CUST_POSTAL_CODE);
   PRIMARYKEY[CUST_ID];
END
```

F

Expressions

An expression is a combination of one or more values, parameters, and functions that evaluate to a value or to a boolean result (Example: True or False).

Syntax

```
expression_list ::= '[' expression [, expression]… ']'
expression ::= value_returned_expression | boolean_returned_expression | '('
expression ')'
```

> **Note**
>
> '(' <expression ')' implies nested functions. Example: MAX(AVG(salary))

Example

```
MAX(AVG(CUSTOMERS.CUST_CREDIT_LIMIT))
```

Value Returned Expressions

A value returned expression is an expression that evaluates to a specific value. For example, a function that calculates the sum of two numbers may return the sum of those numbers as a value. You can then use this value in other parts of the program.

Value-Returned Expression List: A comma-separated list of value-returned expressions.

```
value_returned_expression_list ::= value_returned_expression
[,value_returned_expression]...
```

Value-Returned Expression

```
value_returned_expression ::= term | value_returned_expression {'+'|'-'} term
term ::= factor | term {'*'|'/'} factor
factor ::= ['+' | '-'] primary
primary ::=   column_reference
            | constant_value
            | PARAMETER '[' parameter_name ']'
            | case_expr
            | function_expression
            | '('value_returned_expression ')'
column_reference ::= { THIS | table_name }.column_name // E.g.: CUST.CUST_ID,
                                                                THIS.AMT
(Refer to a previously defined column within the same target dataset)
                     | table_name [EXCLUDE] column_list // E.g.:
sales[amount, quantity, prod_id],
```

```
sales[amount] , sales EXCLUDE [amount, quantity, prod_id]
constant_value ::= [-] number | identifier | date | timestamp | string| NULL
```

Example

```
IMPORT SOURCE CUSTOMERS
DEFINE DATASET CUSTOMERS_D
 ROWSOURCE CUSTOMERS;
 THIS = CUSTOMERS[CUST_ID];

 // value_returned_expression - use of function CONCAT_WS with
column_reference
 THIS[CUST_FULL_NAME] = CONCAT_WS(' ',
CUSTOMERS.CUST_FIRST_NAME,CUSTOMERS.CUST_LAST_NAME);
END
```

Case Expressions

Case expressions allow you to add conditional logic by using either a simple-case or searched-case structure.

Syntax

```
case_expr ::= simple_case_expression | searched_case_expression
simple_case_expression ::= CASE input_expression
                            (WHEN when_expression THEN result_expression )...
                            [ ELSE else_result_expression ]
                           END

searched_case_expression ::= CASE
                             ( WHEN boolean_expression THEN
result_expression ) ...
                             [ ELSE else_result_expression ]
                           END
input_expression ::= expression
when_expression ::= expression
result_expression ::= value_returned_expression
else_result_expression ::= value_returned_expression
boolean_expression ::= boolean_returned_expression
```

- Use a simple CASE expression when you want to evaluate a single expression against multiple potential values.
- Use a searched CASE expression when you need to evaluate multiple, independent boolean expressions to determine the result.

Example 1: Simple Case Expression

```
CASE CUSTOMERS.CUST_VALID
                       WHEN 'I' THEN 'Inactive' WHEN 'A' THEN 'Active'
                       ELSE 'Unknown'
END
```

In this example, if the CUST_VALID column has a value of I, it returns Inactive. If the value is A, it returns Active. If neither condition is met, it returns Unknown.

Example 2: Searched Case Expression

```
CASE
     WHEN CUSTOMERS.CUST_YEAR_OF_BIRTH <= 1973 THEN 'Segment A'
     WHEN CUSTOMERS.CUST_YEAR_OF_BIRTH >1973 THEN 'Segment B'
END
```

In this example, if CUST_YEAR_OF_BIRTH is less than or equal to 1973, it assigns Segment A. Otherwise, it assigns Segment B.

```
CASE WHEN ISNULL(CUSTOMERS.CUST_VALID) THEN 'Unknown' ELSE 'Known' END
```

In this example, if CUST_VALID is NULL, it returns Unknown. Otherwise, it returns Known.

Function Expressions

A function expression represents any valid function that you can use within the framework.

Syntax

```
function_expression ::= general_function | aggregate_function |
window_function | macro_call| datatype_function
```

Function expressions include:

- **General Functions**: Standard functions used for various computations or transformations.
- **Aggregate Functions**: Functions that perform calculations over groups of rows (Example: SUM, AVG).
- **Window Functions**: Functions that compute values over a range of rows within a partition (Example: ROW_NUMBER, RANK).
- **Macro Calls**: Calls to user-defined macros applied to columns or parameters.
- **Datatype Functions**: Functions specific to certain data types (Example: casting or type-specific operations).

Aggregate Functions

An aggregate function performs an operation on sets of values and returns a single result.

Syntax

An aggregate function can be any one of the following:

- **AVG'(' value_returned_expression ')'**: Calculates the mean of the values.
  Input: A numeric expression

  Example:

  ```
  AVG(SALES.AMOUNT_SOLD)
  ```

- **COUNT'(*)'** : Counts all the rows, including those with null values.

Input: None.

Example:

```
COUNT(*)
```

- **COUNT'(' value_returned_expression_list ')'**: Counts the rows where the input expression is non-null.
  Input: A numeric or other type of expression.

  Example:

  ```
  COUNT(SALES.CUST_ID)
  ```

- **COUNT'(' DISTINCT value_returned_expression_list ')'**: Counts the unique, non-null values in the input expression.
  Input: A numeric or categorical expression.

  Example:

  ```
  COUNT(DISTINCT SALES.CUST_ID)
  ```

- **MAX'(' value_returned_expression ')'**: Returns the maximum value in the input.
  Input: A numeric or comparable expression.

  Example:

  ```
  MAX(SALES.AMOUNT_SOLD)
  ```

- **MIN'(' value_returned_expression ')'**: Returns the minimum value in the input.
  Input: A numeric or comparable expression.

  Example:

  ```
  MIN(SALES.AMOUNT_SOLD)
  ```

- **SUM'(' value_returned_expression ')'**: Calculates the total sum of the values in the input.
  Input: A numeric expression.

  Example:

  ```
  SUM(SALES.AMOUNT_SOLD)
  ```

- **FIRST'(' value_returned_expression [, nulls_option] ')'**: Returns the first value in a group. Optionally ignores null values.
  Input: A numeric or comparable expression, and an optional null handling option.

  Example:

  ```
  FIRST(SALES.AMOUNT_SOLD,IGNORE NULLS)
  ```

- **LAST'(' value_returned_expression [, nulls_option] ')'**: Returns the last value in a group. Optionally ignores null values.
  Input: A numeric or comparable expression, and an optional null handling option.

Example:

```
LAST(SALES.AMOUNT_SOLD)
nulls_option ::= [IGNORE | RESPECT] NULLS
```

Datatype Functions

Datatype functions include operations on specific data types.

Syntax

```
datatype_function ::= string_function | numeric_function | date_function |
```

You can use these functions in the following ways:

- **Cast and Convert Function**: For datatype transformations.
- **Date Function**: For date/time operations.
- **Numeric Function**: For numeric calculations.
- **String Function**: For string manipulation.

Cast and Convert Functions

Cast and convert functions transform data between different types.

Cast and convert functions include:

- **TO_DATE'(' value_returned_expression [, format_mask] ')'** : Converts a string to a date using the specified `format_mask`.
  Inputs: A string (`VARCHAR2`) and an optional format mask.

  Example:

  ```
  TO_DATE('2024-01-03','yyyy-MM-dd')
  ```

- **TO_TIMESTAMP'(' value_returned_expression [, format_mask] ')'** : Converts a string to a timestamp using the specified `format_mask`.
  Inputs: A string (`VARCHAR2`) and an optional format mask.

  Example:

  ```
  TO_TIMESTAMP('2024-01-03 01:01:01','yyyy-MM-dd HH:mm:ss')
  ```

- **CAST'(' value_returned_expression AS data_type ')'** : Converts the expression to the specified data type.
  Inputs: Any compatible data types.

  Example:

  ```
  CAST(CUSTOMERS.CUST_YEAR_OF_BIRTH AS VARCHAR2(20))
  ```

  The following table shows the CAST compatibility between six datatypes: `VARCHAR2`, `NUMBER`, `BIGDECIMAL`, `LARGEINT`, `DATE`, and `TIMESTAMP`. The table cells indicate whether the `CAST` between the source and target datatypes is allowed (Yes) or not (No).

Source/ Target	VARCHAR2	NUMBER	BIGDECIMAL	LARGEINT	DATE	TIMESTAMP
VARCHAR2	Yes	Yes	Yes	Yes	Yes	Yes
NUMBER	Yes	Yes	Yes	Yes	No	No
BIGDECIMAL	Yes	Yes	Yes	Yes	No	No
LARGEINT	Yes	Yes	Yes	Yes	No	No
DATE	Yes	No	No	No	Yes	Yes
TIMESTAMP	Yes	No	No	No	Yes	Yes

- **INT('' value_returned_expression '')'** : Converts an expression to an integer..
  Inputs: Exact numeric types. Example: `NUMBER`, `LARGEINT`, `DOUBLE`, `BIGDECIMAL`.

  Example:

  ```
  INT('12.12') // function output will be 12
  ```

- **BIGINT('' value_returned_expression '')'** : Converts an expression to a large integer.
  Inputs: Exact numeric types. Example: `NUMBER`, `LARGEINT`, `DOUBLE`, `BIGDECIMAL`.

  Example:

  ```
  BIGINT('1234567890.1234567890') // function output will be 1234567890
  ```

Date Functions

Date functions perform operations and calculations on date and time values.

Date functions include:

- **ADD_MONTHS('' start_date, num_months '')'** :Returns the date that is `num_months` after `start_date`.
  Inputs: A date and an integer.

  Example:

  ```
  ADD_MONTHS(CUSTOMERS.CUST_EFF_FROM,1)
  ```

- **LAST_DAY('' value_returned_expression '')'**: Returns the last day of the month for the given date.
  Inputs: An expression that evaluates to a date.

  Example:

  ```
  LAST_DAY(CUSTOMERS.CUST_EFF_FROM)
  ```

- **MONTHS_BETWEEN('' end_date, start_date [, roundOff] '')'**: Returns the number of months between.
  Inputs: Two expressions that evaluate to a date or timestamp and an optional BOOLEAN expression. The result is rounded to 8 digits unless `roundOff=false`.

  Example:

  ```
  MONTHS_BETWEEN(CUSTOMERS.CUST_EFF_FROM,CUSTOMERS.CUST_EFF_TO)
  ```

- **NEXT_DAY('' value_returned_expression, day_of_week '')'**: Returns the first date after the expression that matches the specified `day_of_week`.

Inputs: A date and a string for the day of the week (Example: `SU`, `SUN`, `SUNDAY`).

Example:

```
NEXT_DAY(CUSTOMERS.CUST_EFF_FROM,'TUESDAY')
```

- **TRUNC(' value_returned_expression, fmt ')'**: Truncates the date to the unit specified by the format `fmt`.
  Inputs: A date and a format string (Example: `year`, `yyyy`, `yy`, `mon`, `month`, `mm`).

  Example:

  ```
  TRUNC(CUSTOMERS.CUST_EFF_FROM,'YYYY')
  ```

- **DATE_TRUNC(' fmt, value_returned_expression ')'**: Truncates the timestamp to the unit specified by the format `fmt`.
  Inputs: A timestamp and a format string (Example: `YEAR`, `YYYY`, `YY`, `MON`, `MONTH`, `MM`, `DAY`, `DD`, `HOUR`, `MINUTE`, `SECOND`, `WEEK`, `QUARTER`).

  Example:

  ```
  DATE_TRUNC('YEAR',CUSTOMERS.CUST_EFF_FROM)
  ```

- **DATE_SUB(' start_date, integer ')'**: Returns the date obtained by subtracting the integer from the given date.
  Inputs: An expression of date type and an integer.

  Example:

  ```
  DATE_SUB(CUSTOMERS.CUST_EFF_FROM,10)
  ```

- **DATEDIFF(' end_date, start_date ')'**: Returns the number of days between `start_date` and `end_date`.
  Inputs: Two dates.

  Example:

  ```
  DATEDIFF(CUSTOMERS.CUST_EFF_FROM,CUSTOMERS.CUST_EFF_TO)
  ```

- **DATE_ADD(' date, integer ')'**: Returns the date obtained by adding the integer to the given date.
  Inputs: A date and an integer.

  Example:

  ```
  DATE_ADD(CUSTOMERS.CUST_EFF_FROM,10)
  ```

- **DATE_FORMAT(' timestamp, fmt ')'**: Converts the timestamp to a value of string in the format specified by the date format `fmt`.
  Inputs: A timestamp and a format string. (Example: `yyyy-MM-dd,yyyy`.)

  Example:

  ```
  DATE_FORMAT(CUSTOMERS.CUST_EFF_FROM,'yyyy')
  ```

- **CURRENT_DATE '('')'**: Returns the current date.
- **CURRENT_TIMESTAMP '('')'**: Returns the current timestamp.

- **FROM_UNIXTIME' (' value_returned_expression [, fmt] ')'**: Converts UNIX time to timestamp.
  Inputs: An integer expression representing UNIX time and an optional format string expression with a valid format.

  Example:

  ```
  FROM_UNIXTIME(10033743070)
  ```

- **UNIX_TIMESTAMP' (' value_returned_expression [, fmt] ')'**: Returns UNIX time.
  Inputs: A date or timestamp, and an optional format string expression with a valid format (when the first input is of type string and not conformed to the default date or timestamp format).

  Example:

  ```
  UNIX_TIMESTAMP('2023-04-04 00:00:01', 'yyyy-MM-dd HH:mm:ss')
  ```

Numeric Functions

Numeric functions perform calculations on numeric values.

Usage Example: Use of ROUND

```
DEFINE DATASET CUSTOMERS_D
   ROWSOURCE CUSTOMERS;
   THIS[CUST_ID]=CUSTOMERS.CUST_ID;
   THIS[CUST_CREDIT_LIMIT] = ROUND(CUSTOMERS.CUST_CREDIT_LIMIT,2);
   PRIMARYKEY[CUST_ID];
END
```

Numeric functions include the following:

- **ABS'(' value_returned_expression ')'**: Returns the absolute value of the numeric input.
  Inputs: A numeric expression.

  Example: `ABS(-123)`

- **ROUND'(' value_returned_expression, number ')'**: Rounds the numeric input to the specified decimal places using HALF UP rounding. Default is 0.
  Inputs: A numeric expression and the number of decimal places.

  Example: `ROUND(-123.1111,2)`

- **SIGN'(' value_returned_expression ')'**: Returns -1.0, 0.0, or 1.0 if the input is negative, zero, or positive, respectively.
  Inputs: A numeric expression.

  Example: `SIGN(-123)`

- **CEIL'(' value_returned_expression ')'**: Returns the smallest integer greater than or equal to the input.
  Inputs: A numeric expression.

  Example: `CEIL(123.1111)`

- **EXP'(' value_returned_expression ')'**: Returns the exponential raised to the power of the numeric input.
  Inputs: A numeric expression.

  Example: `EXP(2)`

- **FLOOR'(' value_returned_expression ')'**: Returns the largest integer less than or equal to the input.
  Inputs: A numeric expression.

  Example: `FLOOR(123.1111)`
- **LN'(' value_returned_expression ')'**: Returns the natural logarithm of the numeric input.
  Inputs: A numeric expression.

  Example: `LN(2)`
- **LOG'(' base, value_returned_expression ')'**: Returns the logarithm of the input with the specified base.
  Inputs: A numeric base and a numeric expression.

  Example: `LOG(2,4)`
- **MOD'(' value_returned_expression, value_returned_expression ')'**: Returns the remainder after dividing the first input by the second.
  Inputs: Two numeric expressions.

  Example: `MOD(9,5)`
- **SQRT'(' value_returned_expression ')'**: Returns the square root of the numeric input.
  Inputs: A numeric expression.

  Example: `SQRT(9)`
- **SIN'(' value_returned_expression ')'**: Returns the sine of the numeric input (in radians).
  Inputs: A numeric expression.

  Example: `SIN(1.5708)`
- **COS'(' value_returned_expression ')'**: Returns the cosine of the numeric input (in radians).
  Inputs: A numeric expression.

  Example: `COS(1.5708)`
- **TAN'(' value_returned_expression ')'**: Returns the tangent of the numeric input (in radians).
  Inputs: A numeric expression.

  Example: `TAN(1.5708)`
- **POWER'(' value_returned_expression, value_returned_expression ')'**: Raises the first input to the power of the second input.
  Inputs: Two numeric expressions.

  Example: `POWER(4,3)`

String Functions

You can perform various character manipulations using string functions.

String functions include:

- **CHAR'(' value_returned_expression ')'**: Converts a numeric value between 0 and 255 to the character value corresponding to the ASCII code. If the input numeric value is greater than 255, the function uses number % 256 to wrap.
  Inputs: An integral numeric expression.

  Example: `CHAR(35)`
- **CONCAT_WS'(' sep , value_returned_expression_list ')'**: Joins multiple strings using a specified separator.

Inputs: A string.

Example: `CONCAT_WS(' ',CUSTOMERS.CUST_FIRST_NAME,CUSTOMERS.CUST_LAST_NAME)`

- **LOWER('(value_returned_expression ')'**: Converts all characters of the string to lowercase.
  Inputs: A string expression.

  Example: `LOWER(CUSTOMERS[CUST_FIRST_NAME])`

- **SUBSTR('(value_returned_expression, pos [, len] ')'**: Extracts a substring starting at position and length.
  Inputs: A string, starting position (pos) and length (len).

  Example: `SUBSTR(CUSTOMERS[CUST_FIRST_NAME],1,4)`

- **REGEXP_EXTRACT('(value_returned_expression, regexp , idx ')'**: Returns the substring in the input that matches the regular expression group at idx.
  Inputs: A string, a regular expression and a group index integer value.

  Example: `REGEXP_EXTRACT("100-200", "(d+)",1)`

- **CONCAT('(value_returned_expression_list ')'**: Combines multiple strings into a single string.
  Inputs: String expressions.

  Example: `CONCAT(CUSTOMERS.CUST_FIRST_NAME,CUSTOMERS.CUST_LAST_NAME)`

- **TRIM('(value_returned_expression ')'**: Removes leading and trailing spaces from a string.
  Inputs: A string expression.

  Example: `TRIM(CUSTOMERS.CUST_FIRST_NAME)`

- **UPPER('(value_returned_expression ')'**: Converts all characters of the string to uppercase.
  Inputs: A string expression.

  Example: `UPPER(CUSTOMERS[CUST_FIRST_NAME])`

- **INSTR('(str, substr ')'**: Returns the index of the first occurrence of substr in str (1-based).
  Inputs: A string and the substring to search for.

  Example: `INSTR(CUSTOMERS[CUST_FIRST_NAME],"Alex")`

- **ASCII('(value_returned_expression ')'**: Returns the ASCII numeric value of the first character in the string.
  Inputs: A string expression.

  Example: `ASCII("Alex")`

- **LEFT('(str, len ')'**: Returns the leftmost length (len) characters from the string (str). Returns an empty string if len <= 0.
  Inputs: A string (str) and a numeric length.

  Example: `LEFT(CUSTOMERS[CUST_FIRST_NAME],2)`

- **REPLACE('(str, search[, replace] ')'**: Replaces all occurrences of search from the string (str) with replace (the default is an empty string).
  Inputs: A string, search term, and optional replacement string.

  Example: `REPLACE(CUSTOMERS[CUST_VALID],"I","*")`

- **RIGHT('(str, len ')'**: Returns the rightmost length (len) characters from the string (str). Returns an empty string if len <= 0.
  Inputs: A string (str) and a numeric length.

Example: `RIGHT(CUSTOMERS[CUST_FIRST_NAME],2)`

- **SPACE(' value_returned_expression ')'**: Returns a string of spaces with length equal to the numeric value.
  Inputs: An integer expression.

  Example: `SPACE(3)`

- **LENGTH(' value_returned_expression ')'**: Returns a number representing the total number of characters in the input string, including spaces.
  Inputs: An integer expression.

  Example: `LENGTH(CUSTOMERS[CUST_FIRST_NAME])`

- **HASH(' value_returned_expression_list ')'**: Returns a number for hash value of the arguments.
  Inputs: Any expression or expressions list.

  Example: `HASH(CUSTOMERS.CUST_FIRST_NAME,CUSTOMERS.CUST_LAST_NAME)`

General Functions

A general function performs operations on data around NULL value handling and comparisons.

General functions include the following:

- **NVL '(' expr1, expr2 ')'**: Returns expr2 if expr1 is null, otherwise returns expr1.
  Inputs: Two expressions, where expr1 can be null.

  Example: `NVL(PROMOTIONS.PROMO_COST, 0)`

- **NVL2 '(' expr1, expr2, expr3 ')'**: Returns expr2 if expr1 is not null, otherwise returns expr3.
  Inputs: Three expressions.

  Example: `NVL2(PROMOTIONS.PROMO_NAME, 'Eligible', 'Not Eligible')`

- **DECODE '(' value_returned_expression, search, result [, search, result] ... [, default] ')'**: Compares an expression to a list of search values and returns the corresponding result or a default value.
  Inputs:

  ```
  value_returned_expression ::= expression
  search ::= expression //An expression that matches the type of
  value_returned_expression
  result ::= expression //An expression that shares a least common type with
  default_result and the other result expressions
  default_result ::= expressionn //An optional expression that shares a
  least common type with result
  ```

  Example: `DECODE(PROMOTIONS.PROMO_NAME, ' NO PROMOTION #', 1, ' NO PROMOTION', 2, 0)`

- **COALESCE '(' value_returned_expression_list ')'**: Returns the first non-null value in the list of expressions, otherwise, returns null.
  Inputs: Two or more expressions.

  Example: `COALESCE(PROMOTIONS.PROMO_CATEGORY, PROMOTIONS.PROMO_SUBCATEGORY, 'Unknown')`

- **NULLIF '(' value_returned_expression, value_returned_expression ')'**: Returns null if expr1 equals expr2; otherwise, returns expr1.

Inputs: Two expressions for comparison.

Example: `NULLIF(PROMOTIONS.PROMO_CATEGORY, PROMOTIONS.PROMO_SUBCATEGORY)`

- **GREATEST '(' value_returned_expression_list ')'**: Returns the largest value among the provided arguments.
  Inputs: Two or more comparable expressions.

  Example: `GREATEST(1,4,6,7,null)`

- **LEAST '(' value_returned_expression_list ')'**: Returns the smallest value among the provided arguments.
  Inputs: Two or more comparable expressions.

  Example: `LEAST(1,4,6,7,null)`

- **IFNULL '(' expression, expression ')'**: Returns col2 if col1 is null, or col1 otherwise.
  Inputs: Two expressions for comparison.

  Example: `IFNULL(PROMOTIONS.PROMO_CATEGORY, PROMOTIONS.PROMO_SUBCATEGORY)`

User Defined Functions Call

To use user defined functions in Data Augmentation Scripts, you must first define the function and then call the defined function.

For more information, see User Defined Functions (UDFs) or Macros.

Window Functions

A window function combines aggregate, ranking, or analytic functions with an over-clause for windowed calculations.

Syntax

```
window_function ::= aggregate_function over_clause | ranking_function
over_clause | analytic_function over_clause
```

Window functions include:

- `aggregate_function`: For supported aggregate functions, see Aggregate Functions
  Example: `AVG(SALES.AMOUNT_SOLD) OVER (PARTITION BY SALES.CUST_ID)`
- `ranking_function`: Performs ranking operations within a partitioned result set.
  Syntax: `ranking_function::= ROW_NUMBER'(' ')'| RANK'(' ')'| DENSE_RANK'(' ')' | PERCENT_RANK'(' ')'`
  - `ROW_NUMBER()`: Assigns a unique number to each row in the result set.
  - `RANK()`: Assigns a rank to each row, with gaps for ties.
  - `DENSE_RANK()`: Assigns a rank to each row, without gaps for ties.
  - `PERCENT_RANK()`: Calculates the relative rank of a row as a percentage

  Example: `ROW_NUMBER() OVER (PARTITION BY SALES.CUST_ID ORDER BY SALES.AMOUNT_SOLD DESC NULLS LAST)`
- `analytic_function`: Retrieves values from the previous or next rows in a result set that's based on a specified ordering.
  Syntax: `analytic_function::= {LAG | LEAD} '('value_returned_expression[, offset[, default]]')'`

```
offset::= value_returned_expression

default::= value_returned_expression
```

- `LAG()`: Returns the value of a previous row based on the offset.
- `LEAD()`: Returns the value of a subsequent row based on the offset.
- `offset`: Value should be an integer when present. The number of rows back from the current row from which to obtain a value. If not specified, the default is 1.
- `default`: Default value that's used for a null value.

Example: `LAG(SALES.AMOUNT_SOLD, 1, 0) OVER (PARTITION BY SALES.CUST_ID ORDER BY SALES.AMOUNT_SOLD DESC)`

- `over_clause`: Specifies the partitioning, ordering, and frame for window functions.
Syntax:

```
over_clause::= OVER '(' [ partition_by_clause] [order_by_clause]
[ frame_clause ]')'
partition_by_clause::= PARTITION BY value_returned_expression_list
order_by_clause ::= ORDER BY value_returned_expression [ASC|DESC] [NULLS
FIRST|LAST)]
                                (, value_returned_expression [ASC|DESC]
[NULLS(FIRST|LAST)])…

frame_clause::= (ROWS | RANGE) (frame_start | frame_between)
frame_between::= BETWEEN frame_start AND frame_end
frame-start::= UNBOUNDED PRECEDING | unsigned_integer PRECEDING  | CURRENT
ROW
frame_end::= UNBOUNDED FOLLOWING | unsigned_integer FOLLOWING  | CURRENT
ROW
```

  - `partition_by_clause`: Groups rows using PARTITION BY expression.
  - `order_by_clause`: Orders rows using ORDER BY expression.
    - Optionally, specifies whether to sort the rows in ascending or descending order.
      Syntax: `[ ASC | DESC ]`
    - Optionally, specifies whether NULL values are returned before or after non-NULL values.
      Syntax: `NULLS [ FIRST | LAST ]`
  - `frame_clause`: Defines the window frame.
  - `frame-start` and `frame_end`:
    - `UNBOUNDED PRECEDING`: Starts from the first row.
    - `unsigned_integer PRECEDING`: Starts n rows before the current row.
    - `CURRENT ROW`: Includes only the current row.
    - `UNBOUNDED FOLLOWING`: Extends to the last row.
    - `unsigned_integer FOLLOWING`: Ends n rows after the current row.

  Example:

```
SUM(SALES.AMOUNT_SOLD) OVER (PARTITION BY SALES.CUST_ID ORDER BY
SALES.TIME_ID ASC ROWS UNBOUNDED PRECEDING AND CURRENT ROW)
```

Boolean Returned Expressions

Boolean returned expressions evaluate to TRUE or FALSE based on logical conditions.

Boolean returned expressions include:

- **`value_returned_expression { '>=' | '<=' | '>' | '<' | '<>' | '=' }`**
  `value_returned_expression`: Compares two expressions using operators (>=, <=, >, <, <>, =).
  Example:

  ```
  CUSTOMERS.CUST_YEAR_OF_BIRTH >= 2000
  ```

- **`expression [NOT] BETWEEN value_returned_expression AND value_returned_expression`**: Checks if the value of the expression falls within the specified range.
  Example:

  ```
  CUSTOMERS.CUST_CREDIT_LIMIT BETWEEN 50000 AND 100000
  ```

- **`match_expression [NOT] LIKE pattern`**: Matches a string match expression against a specified pattern with the following valid wildcard characters (%, _, [], [^]).
  - **match_expression**: A string that's evaluated against a pattern.
  - **pattern**: A string specifying the pattern for matching with valid wildcard characters: %, _, [], [^].

  Example:

  ```
  CUSTOMERS.FIRST_NAME LIKE 'J%n'
  ```

- **`expression IS [NOT] NULL`**: Checks if the expression is (or is not) NULL.
  Example:

  ```
  CUSTOMERS.CUST_EMAIL IS NOT NULL
  ```

- **`ISNULL '(' expression ')'`**: Returns as `True` if the expression is `Null`, or is `False` otherwise.
  Example:

  ```
  ISNULL(CUSTOMERS.CUST_VALID)
  ```

- **`NOT boolean_returned_expression`**: Negates a boolean expression.
  Example:

  ```
  NOT CUSTOMERS.CUST_YEAR_OF_BIRTH >= 2000
  ```

- **`boolean_returned_expression AND boolean_returned_expression`**: Combines conditions using the logical `AND`.
  Example:

  ```
  CUSTOMERS.CUST_YEAR_OF_BIRTH >= 2000 AND CUSTOMERS.CUST_EMAIL IS NOT NULL
  ```

- **`boolean_returned_expression OR boolean_returned_expression`**:Combines conditions using the logical `OR`.
  Example:

  ```
  CUSTOMERS.CUST_YEAR_OF_BIRTH >= 2000 OR CUSTOMERS.CUST_EMAIL IS NOT NULL
  ```

- **`expression [NOT] IN in_expr`**: Checks if the value of the expression exists (or doesn't exist) in the list of `in_expr` values (value returned expression list).
  Example:

  ```
  CUSTOMERS.CUST_VALID IN ('A', 'I')
  ```

- **`'('boolean_returned_expression')'`**: Groups expressions for logical evaluation.
  Example:

  ```
  (CUSTOMERS.CUST_YEAR_OF_BIRTH >= 2000 AND CUSTOMERS.CUST_EMAIL IS NOT NULL)
  ```

Usage Example

```
DEFINE VERSIONED DATASET CUSTOMERS_D
   //use of boolean_returned_expression
   ROWSOURCE CUSTOMERS WHERE CUSTOMERS.CUST_YEAR_OF_BIRTH IS NOT NULL AND
CUSTOMERS.CUST_SRC_ID IS NULL;
   THIS = CUSTOMERS;
   PRIMARYKEY[CUST_ID];
END
```

G

Column Groups, Indexes, and Partitions

To improve the performance of reporting queries in the database, you can specify constructs such as column groups, indexes, and partitions (only for PROTECTED and PUBLIC datasets).

You specify column groups, indexes, and partitions in the `ADW.conf` file in a Data Augmentation Scripts application. Specifying these constructs is optional.

See:

- COLUMNGROUPS
- INDEXES
- PARTITIONS

COLUMNGROUPS

A column group is a set of columns that are treated as a single unit for query performance.

Column groups work only when the extended statistics feature is enabled in the target data warehouse. By gathering statistics on a column group, the optimizer can more accurately estimate cardinality when a query groups these columns together.

> **Note**
>
> - For a dataset, a column can belong to multiple column groups.
> - Different column groups with the same columns in same orders aren't allowed.
> - Different column groups with the same columns in different orders are allowed.

Syntax

```
column_group_block ::= COLUMNGROUPS '[' column_group_statement ... ']'
column_group_statement ::= CREATE COLUMNGROUP identifier ON table_name
column_list ;
```

Example

```
COLUMNGROUPS
[
CREATE COLUMNGROUP FAW_DW_SALES ON DW_SALES[CUST_ID, SALE_ID];
CREATE COLUMNGROUP FAW_DW_SALES2 ON DW_SALES[SALE_ID, CUST_ID];
CREATE COLUMNGROUP FAW_DW_CUSTOMER ON DW_CUSTOMER[CUST_ID, CUST_NAME];
]
```

INDEXES

Indexes enable you to a quickly look up data in table columns.

By default, local unique BTREE indexes are created based on the primary key for each PROTECTED or PUBLIC dataset.

Additionally, there are instances when:

- There are unnecessary columns in the primary key.
- The few columns on which queries can be beneficial are missing from indexing (outside the primary keys) .

To overcome these issues, you can provide instructions to create non-primary key indexes.

> **Note**
>
> Column groups:
>
> - Support BTREE index.
> - Don't allow different indexes of the same type with the same columns in same orders.
> - Allow different indexes of different types with the same columns.
> - Allow different indexes with the same columns in different orders.

Syntax

```
index_block ::= INDEXES
                                              '['
                                                 { pk_index_statement |
skip_pk_index_statement |
                                                   create_index_statement }
[ create_index_statement ] ...
                                              ']';
pk_index_statement ::= CREATE INDEX ON PRIMARYKEY FOR ALL DATASETS [ EXCEPT
table_list ] ;
skip_pk_index_statement ::= SKIP CREATE INDEX ON PRIMARYKEY FOR ALL DATASETS
[ EXCEPT table_list ] ;
create_index_statement ::= CREATE [ index_type ] unique_spec [ scope_spec ]
INDEX identifier ON table_name column_list ;
table_list ::= '[' table_name (, table_name) ']'
index_type ::= BTREE
unique_spec ::= UNIQUE | NONUNIQUE
scope_spec ::= LOCAL | GLOBAL       //default LOCAL
```

Example

```
INDEXES
[
 SKIP CREATE INDEX ON PRIMARYKEY FOR ALL DATASETS EXCEPT [PROMOTION_D];
 CREATE NON-UNIQUE INDEX city_index ON CUST_D[CUST_CITY, COUNTRY_ID];
```

```
 CREATE UNIQUE GLOBAL INDEX UN_INDEX ON CUST_D[CUST_ID, CUST_FIRST_NAME];
]
```

PARTITIONS

Partitioning allows subdividing tables, indexes, and index-organized tables into smaller pieces.

Partitions enable you to manage and access these database objects at a finer level of granularity.

Syntax

```
partition_block ::= PARTITIONS '[' { list_partition | range_partition }... ']'
list_partition ::= CREATE LIST PARTITION ON table_name '[' column_name ']';
range_partition ::= CREATE RANGE PARTITION ON table_name '[' column_name ']'
                     INTERVALPERIOD '[' period_value ']'
                     [INTERVALVALUE '[' integer ']' ]
                    ;
period_value ::= “YEAR” | “QUARTER” | “MONTH” | “DAY” | “CUSTOM”;
```

> **Note**
>
> - You must use `INTERVALVALUE` only with `period_type DAY` and `CUSTOM`. When used with `CUSTOM`, it signifies an interval in months.
> - For range partitioning, `column_name` must be of datatype `DATE` or `TIMESTAMP`.
> - A table can have only one partition.

Example

```
PARTITIONS
[
   // LIST
   CREATE LIST PARTITION ON PRODUCT_SALES[PROD_CATEGORY];

   // RANGE
   CREATE RANGE PARTITION ON PROMO_SALES[PROMO_DATE] INTERVALPERIOD["DAY"]
   INTERVALVALUE[15];
]
```

H
VIEW QUERY

View query enables you to define views using query language and create views using the dataset definition syntax.

In Data Augmentation Scripts, you create a query file with the `.qry` extension (Example: `<name>.qry`). This file include the queries to create the views in files.

For Autonomous Data Warehouse, this is similar to the *select* syntax in Oracle.

Syntax

```
raw_view_dataset ::= DEFINE RAWVIEW DATASET dataset_name [raw_view_type]
view_specification END
raw_view_type ::= RAWVIEWTYPE = 'string'
String should be a valid DB name e.g., ORACLE. Default is ORACLE. (Note:
Currently only ORACLE is supported.)
view_specification ::= RAWSQL = GETSQL(' '"' query_file_name'"' ')';
```

> **Note**
>
> - Write valid syntax. Data Augmentation Scripts doesn't parse the queries in the query file for invalid syntax.
> - Use full names and fully qualified prefixes for the local tables in the view query because input tables used in queries can be local datasets.
> - View query must not end with `';'`.

Example

Write the view query: channels.qry:

```
SELECT
CHANNELS_D.CHANNEL_CLASS,CHANNELS_D.CHANNEL_CLASS_ID,CHANNELS_D.CHANNEL_DESC,C
HANNELS_D.CHANNEL_ID,TRUNC(SYSDATE) AS CURDATE
FROM DW_LOCODE_X_APP_CHANNELS_D CHANNELS_D
```

Reference the query file in code file (.hrf) main.hrf :

```
DEFINE DATASET CHANNEL_VRAWVIEW
RAWVIEWTYPE = ORACLE;
RAWSQL = GETSQL("channels.qry");
END
```

Table and Column Prefixes

You can define table and column prefixes for instances when in some source applications, the table and column names can begin with special characters that Data Augmentation Scripts doesn't support for IMPORT.

By defining the table and column prefix, you have the option of providing prefixes for tables and column names. You can then use the prefixed object names in subsequent code.

Define prefixes at any one of these two levels:

- Application level prefix for tables and columns
- Table level prefixes for tables and columns

> **Note**
>
> - You must start prefixes with an alphabet. Other characters in the prefix don't have to be alphabets.
> - Keywords not allowed as prefix.
> - Precede `TABLEPREFIX` with `COLPREFIX` when you provide both.
> - Use the prefix names when you reference prefixed columns and tables in datasets.
> - Table-level prefixes override application-level prefixes when you provide both prefixes.
> - Alias takes precedence over application level prefixes when you define both application-level table prefix and import as Alias.

Application level prefix for tables and columns

To ensure that all the tables and columns in the application have the same prefix, you can specify prefixes at the application level.

Syntax

```
application_source_definition ::= APPLICATION SOURCE (COLPREFIX'['<string>']'
| TABLEPREFIX'['<string>']') ]
```

Example

```
APPLICATION SOURCE TABLEPREFIX["CPQ"] COLPREFIX["CPQ"]

IMPORT SOURCE _TRANSACTION_HEADER //Interpret as IMPORT SOURCE
_TRANSACTION_HEADER AS CPQ_TRANSACTION_HEADER

// When referencing prefixed tables in dataset, ensure CPQ prefix is used by
the developer
```

```
DEFINE DATASET DW_THEADER_F FROM CPQ_TRANSACTION_HEADER END

DEFINE DATASET DW_THEADER_F1
 ROWSOURCE CPQ_TRANSACTION_HEADER;
 THIS = TRANSACTION_HEADER EXCLUDE [CPQ_COL1]; // When referencing prefixed
columns in dataset, ensure CPQ prefix is used by the developer
 THIS = TRANSACTION_HEADER[CPQ_COL2];
 THIS[SALES] = TRANSACTION_HEADER[CPQ_COL3];
END
```

Table level prefixes for tables and columns

You can specify prefixes at the table and column levels.

Syntax

Check `COLPREFIX` and `TABLEPREFIX` in the `source_definition` syntax.

Example

```
IMPORT SOURCE _TRANSACTION_HEADER COLPREFIX["AA"] TABLEPREFIX["AA"]
DEFINE DATASET DW_THEADER_F
 ROWSOURCE AA_TRANSACTION_HEADER;
 THIS = AA_TRANSACTION_HEADER[AA_COL1,AA_COL2,AA_COL3];
 PRIMARYKEY[AA_COL1];
END
```

J

Keyboard Shortcuts for Data Augmentation Scripts

You can use these keyboard shortcuts to perform actions in Data Augmentation Scripts.

Task	Keyboard Shortcut
Save the active file.	Ctrl+S (Windows) Command+S (Mac)
Save all files.	Ctrl+Shift+S (Windows) Command+Shift+S (Mac)
Expand the file structure.	Ctrl+B (Windows) Command+B (Mac)
Toggle the console.	Ctrl+' (Windows) Command+' (Mac)
Compile the project.	Ctrl+Shift+B (Windows) Command+Shift+B (Mac)
Build the project.	F5

Glossary

Index