PUBLIC

SAP HANA Platform 2.0 SPS 00
Document Version: 1.0 – 2016-11-30

SAP HANA Series Data Developer Guide

Content

1 Series Data in SAP HANA

When you collect data at a measurable interval such as time, the data is called **series data**. Analysis of series data allows you to draw meaningful conclusions and predictions from the patterns and trends present in the values.

Series table operations are performed like column table operations with a few limitations and exceptions to consider:

Create series tables	Create series tables using the CREATE TABLE statement.
Load, insert, update, and delete series data	Load, insert, update, and delete series data using SQL statements such as LOAD, INSERT, UPDATE, and DELETE.
Import and export series data	Export series data to other systems, including exporting in BINARY format.
Aggregate series data	Aggregate series data by some meaningful characteristic. For example, collect weather data and aggregate it by city or country.
Modify series data and series tables	Use the IMPORT, INSERT, UPDATE, DELETE, REPLACE, or UPSERT statements to update data in series data. Use IMPORT and INSERT statements for periodic bulk loads of data, and the INSERT, UPDATE, DELETE, REPLACE, or UPSERT statements for low frequency changes, such as updating erroneous readings. Convert to smaller and larger time intervals using horizontal disaggregation and horizontal aggregation of series data. For example, to analyze how energy requirements on residences are affected by weather, you could convert a series table that stores energy measurements for every 15 minutes to a series table that stores energy measurements for every hour. Converting to a larger time interval decreases the number of a rows in the series table, while converting to a smaller interval increases the number of rows. You can alter a series table using the ALTER TABLE statement; however, the SERIES clause of an existing series table cannot be altered.
Join and compare data between series tables	Perform meaningful comparisons of similar series data across multiple series tables. For example, you could compare the energy consumption of two different buildings in your organization. You can also join series tables, and all logical join types are supported. However, both series tables must have compatible series definitions. Both can be non-equidistant, or equidistant with the same delta/offset. For example: `SELECT SE1.time, SE1.total_consumption, SE2.total_consumption FROM StoreEnergy SE1 JOIN StoreEnergy SE2 ON SE1.time = SE2.time WHERE SE1.store_id='Store1' AND SE2.store_id='Store2';`

Equality cannot be used to compare the columns when dissimilar series definitions are used, so a complex predicate that detects whether two intervals overlap is required. Two non-equidistant series can be joined but they only match when the particular instant being measured matches perfectly in both inputs.

When joining two different series tables that have different intervals, you may need to adjust the resolution of one or both tables using horizontal aggregation or horizontal disaggregation. The example below illustrates how to use the SERIES_ROUND function in a query block to accommodate the join.

```
SELECT SE.store_id, SE.time, SE.total_consumption, HW.temp
    FROM StoreEnergy SE join Store S ON SE.store_id = S.store_id
    JOIN HourlyWeather HW ON HW.location_id = S.location_id
    AND SERIES_ROUND(SE.time, SERIES TABLE HourlyWeather) =
HW.tsstart;
```

Analyze and summarize series data

Examine and compare different series to look for data anomalies and patterns, and to detect relationships between series. For example, you could analyze changes between two series to determine how changes to one series can help predict changes in another series.

Related Information

SAP HANA SQL and System Views Reference Guide (PDF)

2 Quick reference: Series data SQL support and system views

While most SQL statements and functions in SAP HANA can be used with series tables, a subset of these features are especially designed for use with series data.

The following lists provide an overview of the SQL grammar (with simple examples) and system views that are series-data centric. Full documention for each feature listed is found in the SAP HANA SQL Reference Guide. See http://help.sap.com/hana_platform/.

SQL statements

SERIES clause of the CREATE COLUMN TABLE statements

Use the SERIES clause of the CREATE COLUMN TABLE statements to define a series table. For example:

```
CREATE COLUMN TABLE ExampleStockTrades(
     ticker_symbol CHAR(5),
     trade_time TIMESTAMP,
     price DECIMAL(10,2),
     volume INTEGER)
     SERIES(SERIES KEY(ticker_symbol) NOT EQUIDISTANT
         MINVALUE '2013-01-01'
         MAXVALUE '2014-01-01'
         PERIOD FOR SERIES(trade_time));
```

SERIES REORGANIZE clause of the ALTER COLUMN TABLE statement

When data is first inserted into an equidistant piecewise series table, use this clause to reorder rows to maximize compression.

SQL functions

AUTO_CORR function

Use the AUTO_CORR function to compute all autocorrelation coefficients and return an array of values.. The following example returns [0.285714,-0.351351,-0.5625,-0.25,1,1,1,1]:

```
CREATE COLUMN TABLE correlationTable (TS_ID
VARCHAR(10), DATE DAYDATE, VALUE DOUBLE);
INSERT INTO correlationTable VALUES ('A',
'2014-10-01', 1);
INSERT INTO correlationTable VALUES ('A',
'2014-10-02', 2);
INSERT INTO correlationTable VALUES ('A',
'2014-10-03', 3);
```

```
INSERT INTO correlationTable VALUES ('A',
'2014-10-04', 4);
INSERT INTO correlationTable VALUES ('A',
'2014-10-05', 5);
INSERT INTO correlationTable VALUES ('A',
'2014-10-06', 1);
INSERT INTO correlationTable VALUES ('A',
'2014-10-07', 2);
INSERT INTO correlationTable VALUES ('A',
'2014-10-08', 3);
INSERT INTO correlationTable VALUES ('A',
'2014-10-09', 4);
INSERT INTO correlationTable VALUES ('A',
'2014-10-10', 5);
SELECT TS_ID, AUTO_CORR(VALUE, 8 SERIES (PERIOD FOR
SERIES(DATE)
        EQUIDISTANT INCREMENT BY INTERVAL 1 DAY
MISSING ELEMENTS NOT ALLOWED))
    FROM correlationTable
    GROUP BY TS_ID ORDER BY TS_ID;
```

BINNING window function

Handy for computing historgrams, use the BINNING function to distribute values into bins of equal width. The following example distributes the value of the input set into four bins that all have an equal width:

```
CREATE TABLE weather (station INT, ts DATE,
temperature FLOAT);
INSERT INTO weather VALUES (1, '2014-01-01', 0);
INSERT INTO weather VALUES (1, '2014-01-02', 3);
INSERT INTO weather VALUES (1, '2014-01-03', 4.5);
INSERT INTO weather VALUES (1, '2014-01-04', 6);
INSERT INTO weather VALUES (1, '2014-01-05', 6.3);
INSERT INTO weather VALUES (1, '2014-01-06', 5.9);
SELECT *, BINNING(VALUE => temperature, BIN_COUNT =>
4) OVER () AS bin_num FROM weather;
```

CROSS_CORR function

Use the CROSS_CORR function to compute all cross-correlation coefficients between two columns. The following example returns five values [-1, -0.9285714, -0.6, 0.5, -1]:

```
CREATE COLUMN TABLE TSeries( key INTEGER, ts
TIMESTAMP, val1 DOUBLE, val2 DOUBLE, PRIMARY KEY(key,
ts) )
    SERIES( SERIES KEY (key) EQUIDISTANT INCREMENT BY
INTERVAL 1 DAY PERIOD FOR SERIES(ts) );
INSERT INTO TSeries VALUES (1, '2014-1-1', 1, 3);
INSERT INTO TSeries VALUES (2, '2014-1-3', 2, 4);
INSERT INTO TSeries VALUES (3, '2014-1-4', 4, 2);
INSERT INTO TSeries VALUES (4, '2014-1-5', 3, 1);
SELECT CROSS_CORR(val1, val2, 10 ORDER BY ts) FROM
TSeries;
```

CUBIC_SPLINE_APPROX window function

The CUBIC_SPLINE_APPROX window function replaces null values by interpolating the gaps based on calculated cubic splines and linearly extrapolating any leading or trailing null values. The following example performs cubic spline interpolation on a series using a series table. In this

example the not-a-knot boundary condition is used, and it returns [null, 1, 2, 7, 10, 5, null].

```
CREATE COLUMN TABLE "InterpolationTable"
    (ts_id VARCHAR(20), date DAYDATE, val DOUBLE)
    SERIES(SERIES KEY(ts_id) PERIOD FOR SERIES(date)
        EQUIDISTANT INCREMENT BY INTERVAL 1 DAY
MISSING ELEMENTS ALLOWED);
INSERT INTO "InterpolationTable" VALUES
('A','2013-09-29', null);
INSERT INTO "InterpolationTable" VALUES
('A','2013-09-30', 1);
INSERT INTO "InterpolationTable" VALUES
('A','2013-10-01', 2);
INSERT INTO "InterpolationTable" VALUES
('A','2013-10-02', null);
INSERT INTO "InterpolationTable" VALUES
('A','2013-10-03', 10);
INSERT INTO "InterpolationTable" VALUES
('A','2013-10-04', 5);
INSERT INTO "InterpolationTable" VALUES
('A','2013-10-05', null);
SELECT CUBIC_SPLINE_APPROX(val,
'SPLINE_TYPE_NOT_A_KNOT')
    OVER(SERIES TABLE "InterpolationTable") FROM
"InterpolationTable";
```

DFT function

Use the DFT function to compute the Discrete Fourier Transform (DFT) of a column for the first *N* values and returns an array with exactly N elements. The following example computes the Discrete Fourier Transform of a column in an equidistant series:

```
SELECT DFT(FRACTION_OF_MIN_MAX_RANGE, 4
SERIES(EQUIDISTANT INCREMENT BY 1 PERIOD FOR
SERIES(element_number))).REAL
    FROM SERIES_GENERATE_INTEGER(1,0,10);
```

FIRST_VALUE function

Use the FIRST_VALUE functions to return the first value in a series. The following example returns the first value in COL1 column when the table is ordered by COL2 (or 9):

```
CREATE TABLE T (COL1 DOUBLE, COL2 DOUBLE);
INSERT INTO T VALUES(9, 1);
INSERT INTO T VALUES(4, 5);
INSERT INTO T VALUES(7, 3);
SELECT FIRST_VALUE (COL1 ORDER BY COL2) FROM T;
```

LAST_VALUE functions

Use the LAST_VALUE function to return the last value in a series. The following example returns the last value in COL1 column when the table is ordered by COL2 (or 4):

```
CREATE TABLE T (COL1 DOUBLE, COL2 DOUBLE);
INSERT INTO T VALUES(1, 1);
INSERT INTO T VALUES(4, 5);
INSERT INTO T VALUES(7, 3);
SELECT LAST_VALUE (COL1 ORDER BY COL2) FROM T;
```

LINEAR_APPROX window function

Use the LINEAR_APPROX function to replace null values in equidistant series data by interpolating the gaps and extrapolating any leading or trailing null values. The following example returns [1, 2, 6, 10]:

```
CREATE COLUMN TABLE "InterpolationTable" (TS_ID
VARCHAR(20), date DAYDATE, val DOUBLE);
INSERT INTO "InterpolationTable" VALUES
('A','2013-09-30', 1);
INSERT INTO "InterpolationTable" VALUES
('A','2013-10-01', 2);
INSERT INTO "InterpolationTable" VALUES
('A','2013-10-02', null);
INSERT INTO "InterpolationTable" VALUES
('A','2013-10-03', 10);
SELECT date, val, LINEAR_APPROX (val,
'EXTRAPOLATION_LINEAR')
    OVER (PARTITION BY TS_ID ORDER BY date) FROM
"InterpolationTable";
```

NTH_VALUE functions

Use the NTH_VALUE function to return the Nth value in a series. The following example returns the second value in the COL1 column when the table is ordered by COL2 (or 700):

```
CREATE TABLE T (COL1 DOUBLE, COL2 DOUBLE);
INSERT INTO T VALUES(900, 10);
INSERT INTO T VALUES(400, 50);
INSERT INTO T VALUES(700, 30);
INSERT INTO T VALUES(200, 40);
SELECT NTH_VALUE (COL1, 2 ORDER BY COL2) FROM T;
```

RANDOM_PARTITION window function

Use the RANDOM_PARTITION function to partition an input set randomly into three disjoint subsets by assigning a partition number to each row of the input set. The three subsets are called training, validation, and testing. The union of these three subsets may not be the complete initial dataset. The following example creates random partitions with explicit partition sizes.

```
CREATE TABLE weather (station INT, ts DATE,
temperature FLOAT);
INSERT INTO weather VALUES (1, '2014-01-01', 0);
INSERT INTO weather VALUES (1, '2014-01-02', 3);
INSERT INTO weather VALUES (1, '2014-01-03', 4.5);
INSERT INTO weather VALUES (1, '2014-01-04', 6);
INSERT INTO weather VALUES (1, '2014-01-05', 6.3);
INSERT INTO weather VALUES (1, '2014-01-06', 5.9);
INSERT INTO weather VALUES (2, '2014-01-01', 1);
INSERT INTO weather VALUES (2, '2014-01-02', 3.4);
INSERT INTO weather VALUES (2, '2014-01-03', 5);
INSERT INTO weather VALUES (2, '2014-01-04', 6.7);
INSERT INTO weather VALUES (2, '2014-01-05', 4.6);
INSERT INTO weather VALUES (2, '2014-01-06', 6.9);
SELECT *, RANDOM_PARTITION(0.5, 0.2, 0.3, 0) OVER
(PARTITION BY station ORDER BY ts) AS part_num FROM
weather;
```

SERIES_DISAGGREGATE function

Use the SERIES_DISAGGREGATE function to generate a complete series table with rows disaggregated into a target interval. The following example generates a series of dates ranging from 1999-01-01 to 2001-01-04.

```
CREATE COLUMN TABLE sourceseries(id INT, ts
TIMESTAMP, val DECIMAL(8,2))
    SERIES(SERIES KEY(id) EQUIDISTANT INCREMENT BY
INTERVAL 1 YEAR
        MINVALUE '1999-01-01' MAXVALUE '2003-01-01'
PERIOD FOR SERIES (ts));
CREATE COLUMN TABLE targetseries(id  INT, ts
TIMESTAMP, val DECIMAL(8,2))
    SERIES(SERIES KEY(id) EQUIDISTANT INCREMENT BY
INTERVAL 3 MONTH
        MINVALUE '1999-01-01' MAXVALUE '2001-01-01'
PERIOD FOR SERIES (ts));
SELECT * from SERIES_DISAGGREGATE(
    SERIES TABLE sourceseries, SERIES TABLE
targetseries);
```

SERIES_ELEMENT_TO_PERIOD function

Use the SERIES_ELEMENT_TO_PERIOD function to return elements within the specified period. The following example returns the 5th element from a series from 0 to 10 in increments of 2. It returns the result 8.

```
SELECT SERIES_ELEMENT_TO_PERIOD(5, 2, 0, 10) "val"
FROM DUMMY;
```

SERIES_FILTER window function

Use the SERIES_FILTER window function to returns value from an equidistant series, with the specified filter applied. The following example returns values filtered by the SINGLESMOOTH and DOUBLESMOOTH filters.

```
CREATE COLUMN TABLE weather (ts DATE, temperature
FLOAT);
INSERT INTO weather VALUES ('2014-01-01', 0);
INSERT INTO weather VALUES ('2014-01-02', 3);
INSERT INTO weather VALUES ('2014-01-03', 4.5);
INSERT INTO weather VALUES ('2014-01-04', 6);
INSERT INTO weather VALUES ('2014-01-05', 6.3);
INSERT INTO weather VALUES ('2014-01-06', 6.9);
INSERT INTO weather VALUES ('2014-01-07', NULL);
INSERT INTO weather VALUES ('2014-01-08', NULL);
SELECT ts, temperature,
       SERIES_FILTER(VALUE => temperature,
METHOD_NAME => 'SINGLESMOOTH', ALPHA => 0.2) OVER
(ORDER BY ts) AS SES,
       SERIES_FILTER(VALUE => temperature,
METHOD_NAME => 'DOUBLESMOOTH', ALPHA => 0.2, BETA =>
0.3) OVER (ORDER BY ts) AS DES
FROM weather;
```

SERIES_GENERATE function

Use the SERIES_GENERATE function to generate a new series table based on the specified series table definition. The following example generates a series of timestamps ranging from 1999-01-01 to 1999-01-02 that increments by 30 second intervals using a series table named testseries.

```
CREATE COLUMN TABLE testseries(id  INT, ts TIMESTAMP,
val DOUBLE)
```

```
    SERIES (SERIES KEY(id) EQUIDISTANT INCREMENT BY
INTERVAL 30 SECOND
        MINVALUE '1999-01-01'  MAXVALUE '1999-01-02'
PERIOD FOR SERIES (ts));
SELECT * FROM SERIES_GENERATE(SERIES TABLE
testseries);
```

SERIES_PERIOD_TO_ELEMENT function

Use the SERIES_PERIOD_TO_ELEMENT function to return the period associated with the specified series element number. The following example returns the next element rounded down from 2014-01-05 12:00:00 from a date series from 2014-01-01 to 2014-12-32 in increments of 1 day. It returns the result 5.

```
SELECT SERIES_PERIOD_TO_ELEMENT( '2014-01-05
12:00:00', 'INTERVAL 1 DAY', '2014-01-01',
'2014-12-31', ROUND_HALF_DOWN)
     "element" FROM DUMMY;
```

SERIES_ROUND function

Use the SERIES_ROUND function to round series data in calculations. The following example shows how to round up a time to the next 10 minute interval. It returns the result 4:30:00 AM.

```
SELECT SERIES_ROUND('04:25:01', 'INTERVAL 10 MINUTE')
"result" FROM DUMMY;
```

WEIGHTED_AVG window function

Use the WEIGHTED_AVG function to compute a weighted moving average using arithmetically decreasing weights. The following example returns the weighted moving average for the temperature column values.

```
CREATE TABLE weather (ts DATE, temperature FLOAT);
INSERT INTO weather VALUES ('2014-01-01', 0);
INSERT INTO weather VALUES ('2014-01-02', 3);
INSERT INTO weather VALUES ('2014-01-03', 4.5);
INSERT INTO weather VALUES ('2014-01-04', 6);
SELECT ts, temperature, WEIGHTED_AVG(temperature)
OVER (ORDER BY ts ROWS BETWEEN 1 PRECEDING AND
CURRENT ROW) FROM weather ORDER BY ts
```

Views

SERIES_TABLES view

The SERIES_TABLES view contains configuration settings for each series table, such as period and calendar settings, and whether the series table is equidistant.

M_SERIES_TABLES view

The M_SERIES_TABLES view contains statistics on the physical contents of each series table.

SERIES_KEY_COLUMNS view

The SERIES_KEY_COLUMNS view contains the series key column for each series table.

3 Series Data Loads

Loading data into series table requires the use of statements such as INSERT and LOAD.

Series columns must match the series definition with respect to the following properties when inserting new values:

- Equidistant/delta
  For an equidistant series, the values of the column must be even multiples of the delta.
- MINVALUE/MAXVALUE
  When defined, the values of the series columns should not exceed this range.
- Missing values
  For equidistant series, there should be no gaps in the rows associated with the elements of the series when the series is defined to exclude missing values.

For example, the query below illustrates how to add a single measurement to a series table. The row represents a reading of a particular sensor at a specified timestamp.

```
INSERT INTO RawWeather(sensor_id,time,value)
    VALUES('Therm#01', TO_TIMESTAMP('2014-01-11 13:30:00'), 18.3);
```

Rows can be inserted individually or from a subquery. The example below illustrates how some of the DailyWeather columns can be computed from the HourlyWeather table.

```
INSERT INTO DailyWeather(station_id,date,maxtemp,mintemp,meantemp)
    SELECT station_id, to_date(tsstart), max(temp), min(temp), avg(temp)
        FROM HourlyWeather GROUP BY station_id, to_date(tsstart);
```

You can use the IMPORT statement to bulk-load a series table by the series column.

Related Information

SAP HANA SQL and System Views Reference Guide (PDF)

3.1 Query operations using column functions

The syntax for column functions allows you to add new columns to a result set that are computed from entire columns.

For example, consider the query below.

```
SELECT store_id, time, total_consumption,
    LINEAR_APPROX(total_consumption)
    OVER (PARTITION BY store_id ORDER BY ts)
    AS approximation_consumption
```

```
    FROM StoreEnergy SE;
```

Missing values in the total_consumption column are permitted in the StoreEnergy table in case an energy sensor fails. These missing values are represented by NULL. To perform an analysis, replace these missing values using linear approximation between the adjacent non-NULL values. The LINEAR_APPROX function operates on an entire series at a time, producing a new series that replaces missing values by interpolating between adjacent non-NULL values.

Every column function must be used in either the SELECT clause or the ORDER BY clause of a subquery. Each column function has a parameter list that defines the expected inputs. Each input parameter is either a column input or a scalar input, and this is defined as part of the column function's signature.

Standard SQL operations can be used to express many important analysis questions. In addition to these operations, the following scalar functions can be used with series tables:

- SERIES_ROUND
- SERIES_PERIOD_TO_ELEMENT
- SERIES_ELEMENT_TO_PERIOD

Related Information

SAP HANA SQL and System Views Reference Guide (PDF)

4 Aggregation

Aggregate functions can be used with series data to perform vertical aggregation, and horizontal aggregation and disaggregation.

Vertical aggregation refers to aggregation across different series. Horizontal aggregation refers to aggregation over longer periods. Horizontal disaggregation is the process of breaking up larger periods into smaller periods.

Related Information

SAP HANA SQL and System Views Reference Guide (PDF)

4.1 Vertical Aggregation

You can combine data from multiple series tables when working with series data using the GROUP BY clause and aggregate functions.

The query below illustrates a query that performs vertical aggregation over three series, identified by an IN predicate. The GROUP BY clause includes time but does not include the store_id column. A single series is created.

```
SELECT time, SUM(solar_generation), SUM(total_consumption)
    FROM StoreEnergy WHERE store_id in ('Store1', 'Store2', 'Store3')
    GROUP BY time;
```

The same technique can be used to create a series table grouping by the city of each store, as illustrated in the example below. The GROUP BY clause includes both the city and the time. This query returns a series table with zero or more series, one for each distinct city that has a store.

```
SELECT L.city, SE.time, SUM(solar_generation), SUM(total_consumption)
    FROM StoreEnergy SE JOIN Store S ON SE.store_id = S.store_id
    JOIN Location L ON S.location_id = L.location_id
    GROUP BY L.city, SE.ts;
```

When executing vertical aggregation over multiple series, the series may not have the same intervals encoded.

All combined series must be equidistant with the same offset and delta. When grouping two time series, the interval is only equal when the value of the series column is exactly equal. There is no need to compare two intervals that may overlap.

Related Information

SAP HANA SQL and System Views Reference Guide (PDF)

4.2 Horizontal Aggregation

Horizontal aggregation transforms an equidistant series with a finer interval into a new series with a coarser interval.

When moving from a finer interval to a coarser interval, a computation on the source time to map to the target must be performed. You can either define a SERIES_ROUND function to provide this functionality, or use conversion functions such as TO_DATE, YEAR, and MONTH.

Aggregate functions can be used when performing horizontal aggregation. This analysis is supported by using the GROUP BY clause and aggregate function semantics.

The example below illustrates how to perform horizontal aggregation from hours to days.

```
SELECT station_id, TO_DATE(tsstart) AS date,
    MAX(temp) AS maxtemp, MIN(temp) AS mintemp, AVG(temp) AS meantemp,
    SUM(rain) AS totalrain, SUM(snow) AS totalsnow
    FROM HourlyWeather
    GROUP BY station_id, TO_DATE(tsstart);
```

For example, the following illustrates how to perform a horizontal aggregation from fifteen minute intervals to one hour intervals:

```
SELECT DT.store_id, DT.time, SUM(solar_generation), SUM(total_consumption)
    FROM (SELECT SE.store_id, SERIES_ROUND(SE.time, 'INTERVAL 1 HOUR')
        AS time, SE.solar_generation, SE.total_consumption FROM StoreEnergy SE
    ) DT GROUP BY DT.store_id, DT.time;
```

The example below uses the SHINE database example to illustrate horizontal aggregation. A new series table is defined to have monthly values. generate_exchange_rate_history generates exchange rates every five minutes, which are aggregated to monthly rates using SERIES_ROUND and MEDIAN. The results are inserted into the table.

> **Note**
>
> SHINE cannot be used in a production environment.

```
DROP TABLE sourceseries;
CREATE COLUMN TABLE sourceseries(
    MON TIMESTAMP,
    EUR_USD_FACTOR DECIMAL(9,6),
    MONTHLY_EXPENSE_IN_USD DECIMAL(13,2)
)
SERIES (EQUIDISTANT INCREMENT BY INTERVAL 1 MONTH
    MINVALUE '2012-01-01'
    MAXVALUE '2012-12-01'
    PERIOD FOR SERIES (MON));
INSERT INTO sourceseries
    SELECT min(ts) mon, median(FACTOR), 1000000*median(FACTOR)
```

```
        FROM
"SAP_HANA_DEMO"."sap.hana.democontent.epm.seriesdata::generate_exchange_rate_hist
ory"()
        WHERE TCURR='USD' AND TS < '2013-01-01'
        GROUP BY SERIES_ROUND(ts, 'INTERVAL 1 MONTH', ROUND_FLOOR)
        ORDER BY mon;

SELECT * FROM sourceseries ORDER BY mon;
```

Related Information

SAP HANA SQL and System Views Reference Guide (PDF)
SAP HANA Interactive Education (SHINE) (PDF)

4.3 Horizontal Disaggregation

Transforming an equidistant time series with a coarser delta to one with a finer delta can be performed using the SERIES_DISAGGREGATE function.

The SERIES_DISAGGREGATE function accepts parameters that define source (coarser) and target (finer) series descriptors to generate a result table. This function can be used with numeric arguments to generate a series with numeric periods. This function can also be used to implement custom distribution algorithms that preserve the precision of the distributed values and distribute without a remainder.

The query below illustrates how to use the SERIES_DISAGGREGATE_TIMESTAMP table to disaggregate a series table from days to hours.

```
SELECT * FROM SERIES_DISAGGREGATE_TIMESTAMP(
    'INTERVAL 1 MONTH',
    'INTERVAL 1 DAY',
    TO_DATE('2013-01-01'),
    TO_DATE('2014-01-01'));
```

In the query above, the mean temperature of the day is repeated for each hour. The totalrain column is multiplied by the ratio to evenly divide the precipitation across all hours of the day. Custom weights could be applied by joining to a series giving the weights, and the NUM_GENERATED_PERIODS_IN_SOURCE_PERIOD value can be used to implement custom rounding algorithms to carefully deal with rounding issues.

The result sets of the SERIES_DISAGGREGATE functions have one row for each entry in the series that is defined by a generated series descriptor with a column giving the timestamp(s) of the source series and target series as defined by the corresponding series descriptor. The length of the source and target intervals are returned along with the ratio of their lengths. These are returned separately to allow users to carefully deal with rounding issues if required.

When performing horizontal disaggregation, consider the following rules for creating the values in the target series:

- Copy the source value to all generated rows
- Distribute the source value evenly

- Apply weights or smoothing

Different logic is needed for different types of data. When a single column stores only one kind of data then this logic can be applied on a per-column basis.

The example below uses the SHINE database example to illustrate horizontal aggregation followed by horizontal disaggregation. A new series table is defined to have monthly values. generate_exchange_rate_history generates exchange rates every five minutes, which are aggregated to monthly rates using SERIES_ROUND and MEDIAN. The results are inserted into the table and then disaggregated from one month to one day.

> **Note**
>
> SHINE cannot be used in a production environment.

```
DROP TABLE sourceseries;
CREATE COLUMN TABLE sourceseries(
    MON TIMESTAMP,
    EUR_USD_FACTOR DECIMAL(9,6),
    MONTHLY_EXPENSE_IN_USD DECIMAL(13,2)
)
SERIES (EQUIDISTANT INCREMENT BY INTERVAL 1 MONTH
    MINVALUE '2012-01-01'
    MAXVALUE '2012-12-01'
    PERIOD FOR SERIES (MON));
INSERT INTO sourceseries
    SELECT min(ts) mon, median(FACTOR), 1000000*median(FACTOR)
        FROM
"SAP_HANA_DEMO"."sap.hana.democontent.epm.seriesdata::generate_exchange_rate_hist
ory"()
        WHERE TCURR='USD' AND TS < '2013-01-01'
        GROUP BY SERIES_ROUND(ts, 'INTERVAL 1 MONTH', ROUND_FLOOR)
        ORDER BY mon;

SELECT * FROM sourceseries ORDER BY mon;
SELECT s.mon AS original_month, g.generated_period_start
    AS generated_day, s.MONTHLY_EXPENSE_IN_USD * g.FRACTION_OF_SOURCE_PERIOD AS
DAILY_EXPENSE_IN_USD
    FROM sourceseries AS s
    LEFT JOIN SERIES_DISAGGREGATE_DATE('INTERVAL 1 MONTH', 'INTERVAL 1 DAY',
'2012-01-01', '2013-10-01') g
    ON s.mon = g.SOURCE_PERIOD_START
    ORDER BY generated_day;
```

The following example illustrates how to evenly distribute integer values from hourly to ten minute intervals and assign the remainder to the last interval.

> **Note**
>
> The following example should not be used in a production environment. Consider rounding issues when creating your own distribution algorithm.

```
CREATE TABLE ExampleTable (
    ts TIMESTAMP,
    val INTEGER
);
INSERT INTO ExampleTable VALUES ('2015-01-01 01:00:00', 7);

SELECT r.generated_period_start AS GeneratedTimestamp,
  TO_INTEGER(t.val * fraction_of_source_period) + CASE row_number_in_period
    WHEN elements_in_period THEN
```

```
        t.val - elements_in_period * TO_INTEGER(t.val *
fraction_of_source_period)
    ELSE 0
    END AS DistributedValue
    FROM ExampleTable as t,
      (SELECT source_period_start, generated_period_start,
fraction_of_source_period,
         SERIES_PERIOD_TO_ELEMENT(SOURCE_PERIOD_END, 'INTERVAL 10 MINUTE',
         SOURCE_PERIOD_START, SOURCE_PERIOD_END) - 1 AS
elements_in_period,
         SERIES_PERIOD_TO_ELEMENT(GENERATED_PERIOD_START, 'INTERVAL 10 MINUTE',
         SOURCE_PERIOD_START, GENERATED_PERIOD_START) AS row_number_in_period
       FROM SERIES_DISAGGREGATE_TIMESTAMP('INTERVAL 1 HOUR', 'INTERVAL 10
MINUTE',
         '2015-01-01 01:00:00', '2015-01-01 02:00:00')
       ) r
    WHERE t.ts = r.source_period_start
    ORDER BY r.generated_period_start;
```

The query above produces the following results:

	GENERATEDTIMESTAMP	DISTRIBUTEDVALUE
1	Jan 1, 2015 1:00:00.0 AM	1
2	Jan 1, 2015 1:10:00.0 AM	1
3	Jan 1, 2015 1:20:00.0 AM	1
4	Jan 1, 2015 1:30:00.0 AM	1
5	Jan 1, 2015 1:40:00.0 AM	1
6	Jan 1, 2015 1:50:00.0 AM	2

Related Information

SAP HANA Interactive Education (SHINE) (PDF)

5 Equidistant Piecewise Series Data

Series data that retains a degree of regularity without being equidistant across the entire dataset can be represented using equidistant piecewise tables.

Equidistant piecewise data consists of regions where equal increments exist between successive points, but these increments are not always consistent for all regions in the table. Different series or different parts of a series may have different intervals.

Store equidistant piecewise data in an equidistant piecewise table, or data that has the equidistant piecewise property specified. Data stored in an equidistant piecewise table can be queried and manipulated with enhanced performance.

There following column types are required in an equidistant piecewise table.

- Series key columns
  One or more columns that identify the series that each row belongs to.
- Series column
  A DATE, DATETIME, or TIMESTAMP column that represents the time for each row.
- Alternate Series columns
  A DATE, DATETIME, or TIMESTAMP that is also associated with each row. These columns can be used to record the time for each row, or represent the end of an interval associated with each row. Good compression can be achieved for these columns when successive values within the series differ by a constant amount.
- Value columns
  The data recorded for the specified series at the specifies point in time.

To store this data efficiently, the series column and any alternate series columns must be encoded using an arithmetic formula involving highly-compressible columns. Additional columns that are not directly requested by the user must be present in the column table that stores the equidistant piecewise data to support this encoding.

A view can be created on top of the column table that stores the equidistant piecewise data. This view must contain the original columns that comprised the equidistant piecewise data set. You can work with the equidistant piecewise data set by working directly with the view.

The initial timestamp column is encoded in an equidistant piecewise table by four additional columns. Optimizations for equidistant data are enabled by a column naming convention. For example, assume that a timestamp column is named ts. To efficiently store this data in a table, the timestamp must be encoded in the following columns:

Column	Description
ts.0	This column has the same data type as the user ts column. Insertions, updates, and deletions are performed here.
ts.b	This column has the same data type as the user ts column and is the base timestamp for this encoding.

Column	Description
ts.m	BIGINT. This column is the number of 100 nanosecond increments forming a unit.
ts.x	INTEGER. This column is the number of units between the base timestamp and the encoded timestamp.
ts.f	BIGINT. Reserved for internal use.

Alternative timestamps are encoded in the following columns:

Column	Description
alt_ts.0	This column has the same data type as the user alt_ts columns.
alt_ts.a	BIGINT. This column is the number of nanoseconds between the ts column value and the alt_ts column value.

GENERATED ALWAYS columns that are computed using the SERIES_ROUND function store a lower-resolution version of the timestamp but are highly compressible. These optional columns support the fast processing of predicates over the ts and alt_ts columns. When columns of the generated rounded format are present, the number of rows for which the exact timestamp must be computed are limited. Additionally, when you have queries that involve a GROUP BY over the timestamp or alternate timestamp columns, or when using analytic or attribute views over the equidistant piecewise table, these columns can do the aggregation, improving performance.

Views over equidistant piecewise tables hide supporting columns, allowing you to work with User columns. These views cannot be used as inputs for analytic or attribute views. They also cannot be used for bulk loading of data. You can create a view that computes the original timestamp from the encoded values. In this case, the expression that represents the timestamp must conform exactly to the following form:

```
ts := COALESCE("ts.0",ADD_SECONDS("ts.b","ts.m"*"ts.x"*0.00000001))
alt_ts := COALESCE("alt_ts.0",ADD_SECONDS("ts.b","alt_ts.a"*0.0000001))
```

If you are using Core Data Services (CDS) to develop your persistence model, create an entity that will have the equidistant piecewise property. When this entity is activated in a catalog, the appropriate column table and view is automatically created.

You can create a table and view to store equidistant piecewise data using SQL. Use CDS to generate model SQL statements that can then be run manually. Right-click on the table and view that were generated for the entity, and open the Definition. Export the SQL to use this SQL in your application.

Equidistant piecewise tables cannot include single or double quotes in their name. No primary key should be created on any series tables. Instead, the application should ensure that duplicate values are not inserted. Do not use a HISTORY COLUMN table to store equidistant piecewise data.

Use a non-equidistant table when:

- There is no pattern to the series column values.
- You accept lower compression on the series data in exchange for improved query performance.

Use an equidistant table when:

- You want to use an integer series column.
- You have data points at regular intervals where the date/time types are an even number of intervals from '0001-01-01 00:00:00'.
- You want maximum compression and query performance.

Use an equidistant piecewise table when:

- Your series column is a timestamp, seconddate, or date.
- You have many runs of data where the timestamps for consecutive data points differ by a constant interval.
- The timestamps are not necessarily aligned to any particular value.
- You accept lower query performance in exchange for significantly improved compression of the series column.

Consider the following guidelines when loading data:

- Insert / update / delete statements
  Views over equidistant piecewise tables are updatablewhere the views only include the user-specified columns. You can insert, update, and delete over these views as if they were regular column tables.
- Bulk loading data
  When using the IMPORT FROM statement, you must use the column table. However, it should only be necessary to import to the ts.0, alt_ts.0, series key, and value columns.
- Importing and exporting data
  Use IMPORT and EXPORT statements against views. These statements will work for equidistant piecewise views in the same way as they do for other views.

Consider the following guidelines when querying data:

- SQL Queries
  When writing SQL queries over equidistant data, the queries should select from the view because this provides a simpler representation of the data.
- Analytic Views and Attribute Views
  You cannot use SQL views as inputs to attribute or analytic views. Attribute and analytic views must be constructed against the equidistant piecewise base table. Because the timestamp is not encoded against a single column, you must use only the series key and generated rounded columns as attribute columns. You can only work with the granularity and alignment that you have specified in these columns.

Consider the following guidelines when managing equidistant piecewise tables:

- ALTER TABLE SERIES REORGANIZE statement
  Timestamps in an equidistant piecewise column table can be represented in multiple ways. Initially, they are stored in uncompressed format in the ts.0 and alt_ts.0 columns. However, as more rows get added to the table, it is desirable to change the encoding of these timestamps to achieve good compression. This operation is performed by the ALTER TABLE SERIES REORGANIZE statement. The argument to this statement is the equidistant piecewise column table to be considered. Optionally, you can specify that only certain partitions of that table are to be considered, or that only a certain number of rows per partition be considered. The ALTER TABLE SERIES REORGANIZE statement will find rows (up to the optionally specified limit) that are not optimally encoded, and change their representation in the column table in a way that preserves the value of the expression representing the timestamp in the equidistant piecewise view. This operation is not transactional. It is recommended that this statement be run when there are between thousands to hundreds of thousands of un-reorganized rows per partition in a table. If the statement is run too frequently, it will only consider small numbers of rows, and not make good

decisions on compression. If it is not run frequently enough, large numbers of rows will not have optimal compression, and the memory used to store the data will be larger than necessary. The ALTER TABLE SERIES REORGANIZE statement is processor intensive, and so should be run either frequently enough that it does not interfere with regular operation, or else during quiet periods.

- Monitoring
  Use the M_SERIES_TABLE view to determine how many rows in an equidistant piecewise table are not yet optimally encoded (reorganized). This view shows the number of rows that have been considered for optimal encoding, as well as the number of rows that have been encoded in an equidistant segment.

6 Series Generation

You can generate a complete series according to a definition using the SERIES_GENERATE function, which accepts a series descriptor that is defined either by a set of named parameters, or a reference to a series table directly specified in grammar.

The example below illustrates how to generate a result set with a single column, and one that contains one row for every period that matches the series descriptor of a table.

```
SELECT S.GENERATED_PERIOD_START AS ts
    FROM SERIES_GENERATE_TIMESTAMP(SERIES TABLE DailyWeather) S;
SELECT GENERATED_PERIOD_START
    FROM SERIES_GENERATE_TIMESTAMP('INTERVAL 15 MINUTE',
        '2013-01-01T00:00:00', '2013-01-02T00:00:00');
```

The SERIES_GENERATE function can be used as the preserved side of an outer join when working with sparse series in order to make them dense.

The example below illustrates how to use the SERIES_GENERATE function to produce dense output from sparse output.

```
SELECT S.store_id, S.ts, R.consumption, R.temperature
    FROM (Store S,
        SERIES_GENERATE_TIMESTAMP(SERIES TABLE SYSTEM.StoreEnergy) GEN)
    LEFT JOIN StoreEnergy SE ON SE.store_id = S.store_id
    AND GEN.GENERATED_PERIOD_START = SE.time;
```

Related Information

SAP HANA SQL and System Views Reference Guide (PDF)

7 Series Descriptors

A series table is like any other table but it has additional properties defined in a series descriptor.

- SERIES_KEY
  The series key is a column or multiple columns that uniquely identify a particular series within the table. The table stores a single series when series key columns are not defined. For series tables that do not allow duplicate entries, a PRIMARY KEY can be defined with the SERIES_KEY columns and a period column.
  The series key can be empty. Otherwise, every column name is the name of a column in the series table that is not a column in the PERIOD_COLUMNS.
- PERIOD_TYPE
  The period type allows you to define a series table over a number-based or time-based data type, such as TIMESTAMP, DATE, or TIME.
  The period columns must refer to columns in the table that have the data type specified by the descriptor. The PERIOD_TYPE must be one of the following data types:
  - BIGINT
  - DATE
  - DECIMAL(p,s)
  - INTEGER
  - SMALLINT
  - SECONDDATE
  - SMALLDECIMAL
  - TIME
  - TIMESTAMP
  - TINYINT
- IS_EQUIDISTANT
  Equidistant series tables are associated with times that are defined by a regular pattern. Some series table data may contain time variations or arbitrary times, making them non-equidistant.
  The equidistant series tables offer improved compression compared to non-equidistant tables.
  It is possible to convert non-equidistant series to equidistant.
  Some analysis routines are only available for equidistant data.
- IS_INSTANT
  A time series can represent information about an instant in time or intervals. In some cases, a single column may contain a values such as temperature in one row and consumption in another. In this case, the table can be defined as instant; extra semantic information can be used to convert to an interval series for further analysis where appropriate.
- INCREMENT_BY
  INCREMENT_BY defines the distance between adjacent elements in an equidistant series. The delta is defined by either an integer constant, an interval constant, or a defined number of elements between a minimum and maximum. The delta is used in the physical storage representation to optimize equidistant series table.
- IS_MISSING_ ALLOWED
  Equidistant series define a mapping from the series data type to elements. If the series table has the property there is a single contiguous range of element where every element has a matching row, then the

series table has no missing elements. Otherwise, the internal elements that have no matching row *missing elements*.

- PERIOD_COLUMNS
  Each row of a series table has a period of validity that represents the period in time that the row applies to. When the series table represents instants, then there is a single period column. When the table represents intervals, there can be one or two columns.
  - PERIOD_INSTANT_COLUMN
    A series table that contains instants has a single period column. This column represents the instant in time that the row applies to.
  - PERIOD_START_COLUMN and PERIOD_END_COLUMN
    When defining an interval series table, the start and end of the interval are maintained separately For equidistant series, the INCREMENT_BY allows users to find either end point given the other, so either start or end can be omitted from the series table without loss of information.
- MIN_VALUE and MAX_VALUE
  The series table can be defined with a minimum and/or maximum value. These values are used to verify that loaded data corresponds to the range definition. The values in PERIOD_START_COLUMN or PERIOD_INSTANT_COLUMN must satisfy the MIN_VALUE MAX_VALUE constraint. For values in a PERIOD_END column, the constraint is MIN_VALUE < MAX_VALUE. If not defined, the MIN_VALUE and MAX_VALUE values are NULL. The MIN_VALUE must be aligned with any INCREMENT_BY. For example, a when a series table has INCREMENT_BY of 'INTERVAL 1 MONTH', the MIN_VALUE must start on the first day of a month.
- Value columns
  All other columns that are neither SERIES_KEY columns nor PERIOD_COLUMNS are value columns. All value columns are assumed to be time-dependent, meaning that there is no functional dependency for a value attribute unless includes one of the PERIOD_COLUMNS. This normalization is not enforced by the system and physical designs that are not normalized are permitted and useful in some cases where the application is able to handle or avoid any anomalies the denormalized form allows.

The following restrictions must be met by valid series descriptors:

1. **Equidistant**
   If a series descriptor indicates a series IS_EQUIDISTANT is false then IS_INSTANT must be TRUE; INCREMENT_BY must be NULL; IS_MISSING_ALLOWED must be NULL. Otherwise, INCREMENT_BY must be a string representing a valid number or interval. The IS_MISSING_ALLOWED must be true or false.
2. **Increment by**
   The INCREMENT_BY for an equidistant series provides the fixed delta between elements. When the period type is a numeric type, the string is a number that can be converted to the period type. When the period type is a DATETIME, the string is of the following form:

   ```
   INTERVAL number units
   ```

   number is a representation of a number and *units* is either YEAR, MONTH, DAY, HOUR, MINUTE, or SECOND. The number must be a positive integer unless *units* is SECOND. The number can contain digits after the decimal point. Exponential notation is allowed.
3. **Instant**
   If a series descriptor indicates IS_INSTANT is true, then PERIOD_INSTANT_COLUMN is not NULL; PERIOD_START_COLUMN and PERIOD_END_COLUMN are both NULL. Otherwise, the PERIOD_INSTANT_COLUMN must be NULL, and at least one of PERIOD_START_COLUMN AND PERIOD_END_COLUMN must not be NULL.

Related Information

SAP HANA SQL and System Views Reference Guide (PDF)

8 Multiple Time Zone Considerations

When working with several geographical locations, reporting measurements in different time zones require special considerations.

Daylight savings time imposes a special consideration of the primary key. You cannot duplicate rows for a single time in a series table, and some time zones observe daylight savings.

The series column should be recorded in UTC or in a constant offset from UTC. UTC time uniquely identifies each row in a series.

Important Disclaimer for Features in SAP HANA Platform, Options and Capabilities

SAP HANA server software and tools can be used for several SAP HANA platform and options scenarios as well as the respective capabilities used in these scenarios. The availability of these is based on the available SAP HANA licenses and the SAP HANA landscape, including the type and version of the back-end systems the SAP HANA administration and development tools are connected to. There are several types of licenses available for SAP HANA. Depending on your SAP HANA installation license type, some of the features and tools described in the SAP HANA platform documentation may only be available in the SAP HANA options and capabilities, which may be released independently of an SAP HANA Platform Support Package Stack (SPS). Although various features included in SAP HANA options and capabilities are cited in the SAP HANA platform documentation, each SAP HANA edition governs the options and capabilities available. Based on this, customers do not necessarily have the right to use features included in SAP HANA options and capabilities. For customers to whom these license restrictions apply, the use of features included in SAP HANA options and capabilities in a production system requires purchasing the corresponding software license(s) from SAP. The documentation for the SAP HANA optional components is available in SAP Help Portal at http://help.sap.com/hana_options. If you have additional questions about what your particular license provides, or wish to discuss licensing features available in SAP HANA options, please contact your SAP account team representative.

Important Disclaimers and Legal Information

Coding Samples

Any software coding and/or code lines / strings ("Code") included in this documentation are only examples and are not intended to be used in a productive system environment. The Code is only intended to better explain and visualize the syntax and phrasing rules of certain coding. SAP does not warrant the correctness and completeness of the Code given herein, and SAP shall not be liable for errors or damages caused by the usage of the Code, unless damages were caused by SAP intentionally or by SAP's gross negligence.

Accessibility

The information contained in the SAP documentation represents SAP's current view of accessibility criteria as of the date of publication; it is in no way intended to be a binding guideline on how to ensure accessibility of software products. SAP in particular disclaims any liability in relation to this document. This disclaimer, however, does not apply in cases of willful misconduct or gross negligence of SAP. Furthermore, this document does not result in any direct or indirect contractual obligations of SAP.

Gender-Neutral Language

As far as possible, SAP documentation is gender neutral. Depending on the context, the reader is addressed directly with "you", or a gender-neutral noun (such as "sales person" or "working days") is used. If when referring to members of both sexes, however, the third-person singular cannot be avoided or a gender-neutral noun does not exist, SAP reserves the right to use the masculine form of the noun and pronoun. This is to ensure that the documentation remains comprehensible.

Internet Hyperlinks

The SAP documentation may contain hyperlinks to the Internet. These hyperlinks are intended to serve as a hint about where to find related information. SAP does not warrant the availability and correctness of this related information or the ability of this information to serve a particular purpose. SAP shall not be liable for any damages caused by the use of related information unless damages have been caused by SAP's gross negligence or willful misconduct. All links are categorized for transparency (see: http://help.sap.com/disclaimer).

go.sap.com/registration/contact.html