IoTDB-Quality用户文档

作者：数据质量组

组织：清华大学软件学院

时间：2021年 9月 20日

目录

1 开始 **1**

1.1 概述 . . . 1

1.2 系统对标 . . . 1

1.3 常见问题 . . . 2

2 数据画像 **3**

2.1 Distinct . . . 3

2.2 Histogram . . . 4

2.3 Integral . . . 5

2.4 Mad . . . 7

2.5 Median . . . 9

2.6 MinMax . . . 10

2.7 Mode . . . 12

2.8 MovingAverage . . . 13

2.9 PACF . . . 14

2.10 Percentile . . . 15

2.11 Period . . . 17

2.12 QLB . . . 18

2.13 Re_sample . . . 19

2.14 Sample . . . 22

2.15 Segment . . . 24

2.16 Skew . . . 25

2.17 Spline . . . 27

2.18 Spread . . . 31

2.19 Stddev . . . 32

2.20 TimeWeightedAvg . . . 33

2.21 ZScore . . . 34

3 数据质量 **37**

3.1 Completeness . . . 37

3.2 Consistency . . . 39

3.3 Timeliness . . . 42

3.4 Validity . . . 44

4 数据修复 **47**
4.1 ValueFill 47
4.2 TimestampRepair 49
4.3 ValueRepair 51

5 数据匹配 **54**
5.1 Cov 54
5.2 CrossCorrelation 55
5.3 Dtw 56
5.4 PatternSymmetric 57
5.5 Pearson 58
5.6 SelfCorrelation 60
5.7 SeriesAlign(TODO) 61
5.8 SeriesSimilarity(TODO) 61
5.9 ValueAlign(TODO) 61

6 异常检测 **62**
6.1 ADWIN 62
6.2 IQR 63
6.3 KSigma 65
6.4 LOF 66
6.5 Range 68
6.6 TwoSidedFilter 69

7 频域相关 **72**
7.1 Conv 72
7.2 Deconv 72
7.3 DWT 74
7.4 FFT 75
7.5 HighPass 78
7.6 IFFT 80
7.7 LowPass 81

8 字符串处理 **84**
8.1 RegexMatch 84
8.2 RegexReplace 85
8.3 Replace 86
8.4 Split 87

9 序列发现 **89**

9.1 ConsecutiveSequences 89

9.2 ConsecutiveWindows 90

10 复杂事件处理 **92**

10.1 AND(TODO) 92

10.2 EventMatching(TODO) 92

10.3 EventNameRepair(TODO) 92

10.4 EventTag(TODO) 92

10.5 EventTimeRepair(TODO) 92

10.6 MissingEventRecovery(TODO) 92

10.7 SEQ(TODO) 92

第 1 章 开始

1.1 概述

1.1.1 什么是IoTDB-Quality

Apache IoTDB (Internet of Things Database) 是一个时序数据的数据管理系统，可以为用户提供数据收集、存储和分析等特定的服务。

对基于时序数据的应用而言，数据质量至关重要。**IoTDB-Quality**基于IoTDB用户自定义函数(UDF)，实现了一系列关于数据质量的函数，包括数据画像、数据质量评估与修复等，有效满足了工业领域对数据质量的需求。

1.1.2 快速开始

1. 下载包含全部依赖的jar包和注册脚本；
2. 将jar包复制到IoTDB程序目录的 `ext\udf` 目录下；
3. 运行 `sbin\start-server.bat` （在Windows下）或 `sbin\start-server.sh` （在Linux或MacOS下）以启动IoTDB服务器；
4. 将注册脚本复制到IoTDB的程序目录下（与 `sbin` 目录同级的根目录下），修改脚本中的参数（如果需要）并运行注册脚本以注册UDF。

1.1.3 联系我们

- Email: iotdb-quality@protonmail.com

1.2 系统对标

1.2.1 InfluxDB v2.0

InfluxDB是一个流行的时序数据库。InfluxQL是它的查询语言，其部分通用函数与数据画像相关。这些函数与IoTDB-Quality数据画像函数的对比如下（*Native*指该函数已经作为IoTDB的Native函数实现，*Built-in UDF*指该函数已经作为IoTDB的内建UDF函数实现）：

IoTDB-Quality的数据画像函数	InfluxQL的通用函数
Native	COUNT()
Distinct	DISTINCT()
Integral	INTEGRAL()
Native	MEAN()
Median	MEDIAN()
Mode	MODE()
Spread	SPREAD()
Stddev	STDDEV()
Native	SUM()
Built-in UDF	BOTTOM()
Native	FIRST()
Native	LAST()
Native	MAX()
Native	MIN()
Percentile	PERCENTILE()
Sample	SAMPLE()
Built-in UDF	TOP()
Histogram	HISTOGRAM()
Mad	
Skew	SKEW()
TimeWeightedAVG	TIMEWEIGHTEDAVG()
SelfCorrelation	
CrossCorrelation	

InfluxDB可使用Kapacitor提供的UDF功能实现自定义异常检测。由于Kapacitor可以使用python脚本，因此缺乏可用于异常检测的原生函数。

1.3 常见问题

1.3.1 函数名是否大小写敏感

函数名是大小写不敏感的，用户可以根据自己的使用习惯，选择大写、小写或是大小写混合。

第 2 章 数据画像

2.1 Distinct

2.1.1 函数简介

本函数可以返回输入序列中出现的所有不同的元素。

函数名： DISTINCT

输入序列： 仅支持单个输入序列，类型可以是任意的

输出序列： 输出单个序列，类型与输入相同。

提示：

- 输出序列的时间戳是无意义的。输出顺序是任意的。
- 缺失值和空值将被忽略，但 NaN 不会被忽略。

2.1.2 使用示例

输入序列：

```
+-----------------------------+---------------+
|                         Time|root.test.d2.s2|
+-----------------------------+---------------+
|2020-01-01T08:00:00.001+08:00|          Hello|
|2020-01-01T08:00:00.002+08:00|          hello|
|2020-01-01T08:00:00.003+08:00|          Hello|
|2020-01-01T08:00:00.004+08:00|          World|
|2020-01-01T08:00:00.005+08:00|          World|
+-----------------------------+---------------+
```

用于查询的SQL语句：

```
select distinct(s2) from root.test.d2
```

输出序列：

```
+-----------------------------+------------------------+
|                         Time|distinct(root.test.d2.s2)|
+-----------------------------+------------------------+
|1970-01-01T08:00:00.001+08:00|                   Hello|
|1970-01-01T08:00:00.002+08:00|                   hello|
|1970-01-01T08:00:00.003+08:00|                   World|
+-----------------------------+------------------------+
```

2.2 Histogram

2.2.1 函数简介

本函数用于计算单列数值型数据的分布直方图。

函数名： HISTOGRAM

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

参数：

- `start`：表示所求数据范围的下限，默认值为-Double.MAX_VALUE。
- `end`：表示所求数据范围的上限，默认值为Double.MAX_VALUE，`start`的值必须小于或等于`end`。
- `count`: 表示直方图分桶的数量，默认值为1，其值必须为正整数。

输出序列： 直方图分桶的值，其中第i个桶（从1开始计数）表示的数据范围下界为 $start + (i-1) \cdot \frac{end-start}{count}$，数据范围上界为 $start + i \cdot \frac{end-start}{count}$。

提示：

- 如果某个数据点的数值小于`start`，它会被放入第1个桶；如果某个数据点的数值大于`end`，它会被放入最后1个桶。
- 数据中的空值、缺失值和`NaN`将会被忽略。

2.2.2 使用示例

输入序列：

```
+-----------------------------+-----------+
|                         Time|root.test.d1.s1|
+-----------------------------+-----------+
|2020-01-01T00:00:00.000+08:00|        1.0|
|2020-01-01T00:00:01.000+08:00|        2.0|
|2020-01-01T00:00:02.000+08:00|        3.0|
|2020-01-01T00:00:03.000+08:00|        4.0|
|2020-01-01T00:00:04.000+08:00|        5.0|
|2020-01-01T00:00:05.000+08:00|        6.0|
|2020-01-01T00:00:06.000+08:00|        7.0|
|2020-01-01T00:00:07.000+08:00|        8.0|
|2020-01-01T00:00:08.000+08:00|        9.0|
|2020-01-01T00:00:09.000+08:00|       10.0|
|2020-01-01T00:00:10.000+08:00|       11.0|
|2020-01-01T00:00:11.000+08:00|       12.0|
|2020-01-01T00:00:12.000+08:00|       13.0|
|2020-01-01T00:00:13.000+08:00|       14.0|
|2020-01-01T00:00:14.000+08:00|       15.0|
|2020-01-01T00:00:15.000+08:00|       16.0|
|2020-01-01T00:00:16.000+08:00|       17.0|
|2020-01-01T00:00:17.000+08:00|       18.0|
```

```
|2020-01-01T00:00:18.000+08:00|                19.0|
|2020-01-01T00:00:19.000+08:00|                20.0|
+-----------------------------+--------------------+
```

用于查询的SQL语句：

```
select histogram(s1,"start"="1","end"="20","count"="10") from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------------------------------------+
|                         Time|histogram(root.test.d1.s1, "start"="1", "end"="20", "count"="10")|
+-----------------------------+-----------------------------------------------------------------------------+
|1970-01-01T08:00:00.000+08:00|                                                                            2|
|1970-01-01T08:00:00.001+08:00|                                                                            2|
|1970-01-01T08:00:00.002+08:00|                                                                            2|
|1970-01-01T08:00:00.003+08:00|                                                                            2|
|1970-01-01T08:00:00.004+08:00|                                                                            2|
|1970-01-01T08:00:00.005+08:00|                                                                            2|
|1970-01-01T08:00:00.006+08:00|                                                                            2|
|1970-01-01T08:00:00.007+08:00|                                                                            2|
|1970-01-01T08:00:00.008+08:00|                                                                            2|
|1970-01-01T08:00:00.009+08:00|                                                                            2|
+-----------------------------+-----------------------------------------------------------------------------+
```

2.2.2.1 Zeppelin示例

链接: <http://101.6.15.213:18181/#/notebook/2GC1HE97R>

2.3 Integral

2.3.1 函数简介

本函数用于计算时间序列的数值积分，即以时间为横坐标、数值为纵坐标绘制的折线图中折线以下的面积。

函数名： INTEGRAL

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

参数：

- `unit`：积分求解所用的时间轴单位，取值为"1S", "1s", "1m", "1H", "1d"（区分大小写），分别表示以毫秒、秒、分钟、小时、天为单位计算积分。

缺省情况下取"1s"，以秒为单位。

输出序列： 输出单个序列，类型为DOUBLE，序列仅包含一个时间戳为0、值为积分结果的数据点。

提示：

- 积分值等于折线图中每相邻两个数据点和时间轴形成的直角梯形的面积之和，不同时间单位下相当于横轴进行不同倍数放缩，得到的积分值可直接按放缩倍数转换。
- 数据中 NaN 将会被忽略。折线将以临近两个有值数据点为准。

2.3.2 使用示例

2.3.2.1 参数缺省

缺省情况下积分以1s为时间单位。

输入序列：

```
+-----------------------------+---------------+
|                         Time|root.test.d1.s1|
+-----------------------------+---------------+
|2020-01-01T00:00:01.000+08:00|              1|
|2020-01-01T00:00:02.000+08:00|              2|
|2020-01-01T00:00:03.000+08:00|              5|
|2020-01-01T00:00:04.000+08:00|              6|
|2020-01-01T00:00:05.000+08:00|              7|
|2020-01-01T00:00:08.000+08:00|              8|
|2020-01-01T00:00:09.000+08:00|            NaN|
|2020-01-01T00:00:10.000+08:00|             10|
+-----------------------------+---------------+
```

用于查询的SQL语句：

```
select integral(s1) from root.test.d1 where time <= 2020-01-01 00:00:10
```

输出序列：

```
+-----------------------------+-----------------------------+
|                         Time|integral(root.test.d1.s1)|
+-----------------------------+-----------------------------+
|1970-01-01T08:00:00.000+08:00|                         57.5|
+-----------------------------+-----------------------------+
```

其计算公式为：

$$\frac{1}{2}[(1+2)\times 1+(2+5)\times 1+(5+6)\times 1+(6+7)\times 1+(7+8)\times 3+(8+10)\times 2]=57.5$$

2.3.2.2 指定时间单位

指定以分钟为时间单位。

输入序列同上，用于查询的SQL语句如下：

```
select integral(s1, "unit"="1m") from root.test.d1 where time <= 2020-01-01 00:00:10
```

输出序列：

```
+-----------------------------+-----------------------+
|                         Time|integral(root.test.d1.s1)|
+-----------------------------+-----------------------+
|1970-01-01T08:00:00.000+08:00|                  0.958|
+-----------------------------+-----------------------+
```

其计算公式为：

$$\frac{1}{2\times 60}[(1+2)\times 1+(2+3)\times 1+(5+6)\times 1+(6+7)\times 1+(7+8)\times 3+(8+10)\times 2]=0.958$$

2.4 Mad

2.4.1 函数简介

本函数用于计算单列数值型数据的精确或近似绝对中位差，绝对中位差为所有数值与其中位数绝对偏移量的中位数。

如有数据集 $\{1,3,3,5,5,6,7,8,9\}$，其中位数为5，所有数值与中位数的偏移量的绝对值为 $\{0,0,1,2,2,2,3,4,4\}$，其中位数为2，故而原数据集的绝对中位差为2。

函数名： MAD

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

参数：

- `error`：近似绝对中位差的基于数值的误差百分比，取值范围为[0,1)，默认值为0。如当`error`=0.01时，记精确绝对中位差为a，近似绝对中位差为b，不等式 $0.99a \leq b \leq 1.01a$ 成立。当`error`=0时，计算结果为精确绝对中位差。

输出序列： 输出单个序列，类型为DOUBLE，序列仅包含一个时间戳为0、值为绝对中位差的数据点。

提示： 数据中的空值、缺失值和`NaN`将会被忽略。

2.4.2 使用示例

2.4.2.1 精确查询

当`error`参数缺省或为0时，本函数计算精确绝对中位差。

输入序列：

```
+-----------------------------+------------+
|                         Time|root.test.s0|
+-----------------------------+------------+
|2021-03-17T10:32:17.054+08:00|   0.5319929|
|2021-03-17T10:32:18.054+08:00|   0.9304316|
|2021-03-17T10:32:19.054+08:00|  -1.4800133|
|2021-03-17T10:32:20.054+08:00|   0.6114087|
```

```
|2021-03-17T10:32:21.054+08:00|    2.5163336|
|2021-03-17T10:32:22.054+08:00|   -1.0845392|
|2021-03-17T10:32:23.054+08:00|    1.0562582|
|2021-03-17T10:32:24.054+08:00|    1.3867859|
|2021-03-17T10:32:25.054+08:00| -0.45429882|
|2021-03-17T10:32:26.054+08:00|    1.0353678|
|2021-03-17T10:32:27.054+08:00|    0.7307929|
|2021-03-17T10:32:28.054+08:00|    2.3167255|
|2021-03-17T10:32:29.054+08:00|     2.342443|
|2021-03-17T10:32:30.054+08:00|    1.5809103|
|2021-03-17T10:32:31.054+08:00|    1.4829416|
|2021-03-17T10:32:32.054+08:00|    1.5800357|
|2021-03-17T10:32:33.054+08:00|    0.7124368|
|2021-03-17T10:32:34.054+08:00| -0.78597564|
|2021-03-17T10:32:35.054+08:00|    1.2058644|
|2021-03-17T10:32:36.054+08:00|    1.4215064|
|2021-03-17T10:32:37.054+08:00|    1.2808295|
|2021-03-17T10:32:38.054+08:00|   -0.6173715|
|2021-03-17T10:32:39.054+08:00|  0.06644377|
|2021-03-17T10:32:40.054+08:00|     2.349338|
|2021-03-17T10:32:41.054+08:00|    1.7335888|
|2021-03-17T10:32:42.054+08:00|    1.5872132|
.............
Total line number = 10000
```

用于查询的SQL语句：

```
select mad(s0) from root.test
```

输出序列：

```
+-----------------------------+-----------------+
|                         Time| mad(root.test.s0)|
+-----------------------------+-----------------+
|1970-01-01T08:00:00.000+08:00|0.6806197166442871|
+-----------------------------+-----------------+
```

2.4.2.2 近似查询

当 error 参数取值不为0时，本函数计算近似绝对中位差。

输入序列同上，用于查询的SQL语句如下：

```
select mad(s0, "error"="0.01") from root.test
```

输出序列：

```
+-----------------------------+-------------------------------------+
|                         Time|mad(root.test.s0, "error"="0.01")|
+-----------------------------+-------------------------------------+
```

```
|1970-01-01T08:00:00.000+08:00|                0.6806616245859518|
+-----------------------------+----------------------------------+
```

2.5 Median

2.5.1 函数简介

本函数用于计算单列数值型数据的精确或近似中位数。

函数名： MEDIAN

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- `error`：近似中位数的基于排名的误差百分比，取值范围[0,1)，默认值为0。如当`error`=0.01时，计算出的中位数的真实排名百分比在0.49~0.51之间。当`error`=0时，计算结果为精确中位数。

输出序列： 输出单个序列，类型为DOUBLE，序列仅包含一个时间戳为0、值为中位数的数据点。

2.5.2 使用示例

输入序列：

```
+-----------------------------+-----------+
|                         Time|root.test.s0|
+-----------------------------+-----------+
|2021-03-17T10:32:17.054+08:00|  0.5319929|
|2021-03-17T10:32:18.054+08:00|  0.9304316|
|2021-03-17T10:32:19.054+08:00| -1.4800133|
|2021-03-17T10:32:20.054+08:00|  0.6114087|
|2021-03-17T10:32:21.054+08:00|  2.5163336|
|2021-03-17T10:32:22.054+08:00| -1.0845392|
|2021-03-17T10:32:23.054+08:00|  1.0562582|
|2021-03-17T10:32:24.054+08:00|  1.3867859|
|2021-03-17T10:32:25.054+08:00| -0.45429882|
|2021-03-17T10:32:26.054+08:00|  1.0353678|
|2021-03-17T10:32:27.054+08:00|  0.7307929|
|2021-03-17T10:32:28.054+08:00|  2.3167255|
|2021-03-17T10:32:29.054+08:00|   2.342443|
|2021-03-17T10:32:30.054+08:00|  1.5809103|
|2021-03-17T10:32:31.054+08:00|  1.4829416|
|2021-03-17T10:32:32.054+08:00|  1.5800357|
|2021-03-17T10:32:33.054+08:00|  0.7124368|
|2021-03-17T10:32:34.054+08:00| -0.78597564|
|2021-03-17T10:32:35.054+08:00|  1.2058644|
|2021-03-17T10:32:36.054+08:00|  1.4215064|
```

```
|2021-03-17T10:32:37.054+08:00|    1.2808295|
|2021-03-17T10:32:38.054+08:00|   -0.6173715|
|2021-03-17T10:32:39.054+08:00|  0.06644377|
|2021-03-17T10:32:40.054+08:00|     2.349338|
|2021-03-17T10:32:41.054+08:00|    1.7335888|
|2021-03-17T10:32:42.054+08:00|    1.5872132|
.............
Total line number = 10000
```

用于查询的SQL语句：

```
select median(s0, "error"="0.01") from root.test
```

输出序列：

```
+----------------------------+-------------------------------------+
|                        Time|median(root.test.s0, "error"="0.01")|
+----------------------------+-------------------------------------+
|1970-01-01T08:00:00.000+08:00|                1.021884560585022|
+----------------------------+-------------------------------------+
```

2.6 MinMax

2.6.1 函数简介

本函数将输入序列使用min-max方法进行标准化。最小值归一至0，最大值归一至1.

函数名： MINMAX

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

参数：

- `method`：若设置为"batch"，则将数据全部读入后转换；若设置为"stream"，则需用户提供最大值及最小值进行流式计算转换。默认为"batch"。
- `min`：使用流式计算时的最小值。
- `max`：使用流式计算时的最大值。

输出序列： 输出单个序列，类型为DOUBLE。

2.6.2 使用示例

2.6.2.1 全数据计算

输入序列：

```
+----------------------------+------------+
|                        Time|root.test.s1|
+----------------------------+------------+
|1970-01-01T08:00:00.100+08:00|         0.0|
|1970-01-01T08:00:00.200+08:00|         0.0|
```

```
|1970-01-01T08:00:00.300+08:00|           1.0|
|1970-01-01T08:00:00.400+08:00|          -1.0|
|1970-01-01T08:00:00.500+08:00|           0.0|
|1970-01-01T08:00:00.600+08:00|           0.0|
|1970-01-01T08:00:00.700+08:00|          -2.0|
|1970-01-01T08:00:00.800+08:00|           2.0|
|1970-01-01T08:00:00.900+08:00|           0.0|
|1970-01-01T08:00:01.000+08:00|           0.0|
|1970-01-01T08:00:01.100+08:00|           1.0|
|1970-01-01T08:00:01.200+08:00|          -1.0|
|1970-01-01T08:00:01.300+08:00|          -1.0|
|1970-01-01T08:00:01.400+08:00|           1.0|
|1970-01-01T08:00:01.500+08:00|           0.0|
|1970-01-01T08:00:01.600+08:00|           0.0|
|1970-01-01T08:00:01.700+08:00|          10.0|
|1970-01-01T08:00:01.800+08:00|           2.0|
|1970-01-01T08:00:01.900+08:00|          -2.0|
|1970-01-01T08:00:02.000+08:00|           0.0|
+-----------------------------+---------------+
```

用于查询的SQL语句：

```
select minmax(s1) from root.test
```

输出序列：

```
+-----------------------------+--------------------+
|                         Time|minmax(root.test.s1)|
+-----------------------------+--------------------+
|1970-01-01T08:00:00.100+08:00| 0.16666666666666666|
|1970-01-01T08:00:00.200+08:00| 0.16666666666666666|
|1970-01-01T08:00:00.300+08:00|                0.25|
|1970-01-01T08:00:00.400+08:00| 0.08333333333333333|
|1970-01-01T08:00:00.500+08:00| 0.16666666666666666|
|1970-01-01T08:00:00.600+08:00| 0.16666666666666666|
|1970-01-01T08:00:00.700+08:00|                 0.0|
|1970-01-01T08:00:00.800+08:00|  0.3333333333333333|
|1970-01-01T08:00:00.900+08:00| 0.16666666666666666|
|1970-01-01T08:00:01.000+08:00| 0.16666666666666666|
|1970-01-01T08:00:01.100+08:00|                0.25|
|1970-01-01T08:00:01.200+08:00| 0.08333333333333333|
|1970-01-01T08:00:01.300+08:00| 0.08333333333333333|
|1970-01-01T08:00:01.400+08:00|                0.25|
|1970-01-01T08:00:01.500+08:00| 0.16666666666666666|
|1970-01-01T08:00:01.600+08:00| 0.16666666666666666|
|1970-01-01T08:00:01.700+08:00|                 1.0|
|1970-01-01T08:00:01.800+08:00|  0.3333333333333333|
|1970-01-01T08:00:01.900+08:00|                 0.0|
|1970-01-01T08:00:02.000+08:00| 0.16666666666666666|
```

```
+-----------------------------+---------------+
```

2.7 Mode

2.7.1 函数简介

本函数用于计算时间序列的众数，即出现次数最多的元素。

函数名： MODE

输入序列： 仅支持单个输入序列，类型可以是任意的。

输出序列： 输出单个序列，类型与输入相同，序列仅包含一个时间戳为0、值为众数的数据点。

提示：

- 如果有多个出现次数最多的元素，将会输出任意一个。
- 数据中的空值和缺失值将会被忽略，但 NaN 不会被忽略。

2.7.2 使用示例

输入序列：

```
+-----------------------------+---------------+
|                         Time|root.test.d2.s2|
+-----------------------------+---------------+
|1970-01-01T08:00:00.001+08:00|          Hello|
|1970-01-01T08:00:00.002+08:00|          hello|
|1970-01-01T08:00:00.003+08:00|          Hello|
|1970-01-01T08:00:00.004+08:00|          World|
|1970-01-01T08:00:00.005+08:00|          World|
|1970-01-01T08:00:01.600+08:00|          World|
|1970-01-15T09:37:34.451+08:00|          Hello|
|1970-01-15T09:37:34.452+08:00|          hello|
|1970-01-15T09:37:34.453+08:00|          Hello|
|1970-01-15T09:37:34.454+08:00|          World|
|1970-01-15T09:37:34.455+08:00|          World|
+-----------------------------+---------------+
```

用于查询的SQL语句：

```
select mode(s2) from root.test.d2
```

输出序列：

```
+-----------------------------+-----------------------+
|                         Time|mode(root.test.d2.s2)|
+-----------------------------+-----------------------+
|1970-01-01T08:00:00.000+08:00|                  World|
+-----------------------------+-----------------------+
```

2.8 MovingAverage

2.8.1 函数简介

本函数计算序列的移动平均。

函数名： MOVINGAVERAGE

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

参数：

- `n`：移动窗口的长度。

输出序列：输出单个序列，类型为DOUBLE。

2.8.2 使用示例

2.8.2.1 指定窗口长度

输入序列：

```
+-----------------------------+------------+
|                         Time|root.test.s1|
+-----------------------------+------------+
|1970-01-01T08:00:00.100+08:00|         0.0|
|1970-01-01T08:00:00.200+08:00|         0.0|
|1970-01-01T08:00:00.300+08:00|         1.0|
|1970-01-01T08:00:00.400+08:00|        -1.0|
|1970-01-01T08:00:00.500+08:00|         0.0|
|1970-01-01T08:00:00.600+08:00|         0.0|
|1970-01-01T08:00:00.700+08:00|        -2.0|
|1970-01-01T08:00:00.800+08:00|         2.0|
|1970-01-01T08:00:00.900+08:00|         0.0|
|1970-01-01T08:00:01.000+08:00|         0.0|
|1970-01-01T08:00:01.100+08:00|         1.0|
|1970-01-01T08:00:01.200+08:00|        -1.0|
|1970-01-01T08:00:01.300+08:00|        -1.0|
|1970-01-01T08:00:01.400+08:00|         1.0|
|1970-01-01T08:00:01.500+08:00|         0.0|
|1970-01-01T08:00:01.600+08:00|         0.0|
|1970-01-01T08:00:01.700+08:00|        10.0|
|1970-01-01T08:00:01.800+08:00|         2.0|
|1970-01-01T08:00:01.900+08:00|        -2.0|
|1970-01-01T08:00:02.000+08:00|         0.0|
+-----------------------------+------------+
```

用于查询的SQL语句：

```
select movingaverage(s1, "n"="3") from root.test
```

输出序列：

```
+-----------------------------+--------------------------------------+
|                         Time|movingaverage(root.test.s1, "n"="3")|
+-----------------------------+--------------------------------------+
|1970-01-01T08:00:00.300+08:00|                   0.3333333333333333|
|1970-01-01T08:00:00.400+08:00|                                   0.0|
|1970-01-01T08:00:00.500+08:00|                  -0.3333333333333333|
|1970-01-01T08:00:00.600+08:00|                                   0.0|
|1970-01-01T08:00:00.700+08:00|                  -0.6666666666666666|
|1970-01-01T08:00:00.800+08:00|                                   0.0|
|1970-01-01T08:00:00.900+08:00|                   0.6666666666666666|
|1970-01-01T08:00:01.000+08:00|                                   0.0|
|1970-01-01T08:00:01.100+08:00|                   0.3333333333333333|
|1970-01-01T08:00:01.200+08:00|                                   0.0|
|1970-01-01T08:00:01.300+08:00|                  -0.6666666666666666|
|1970-01-01T08:00:01.400+08:00|                                   0.0|
|1970-01-01T08:00:01.500+08:00|                   0.3333333333333333|
|1970-01-01T08:00:01.600+08:00|                                   0.0|
|1970-01-01T08:00:01.700+08:00|                   3.3333333333333335|
|1970-01-01T08:00:01.800+08:00|                                   4.0|
|1970-01-01T08:00:01.900+08:00|                                   0.0|
|1970-01-01T08:00:02.000+08:00|                  -0.6666666666666666|
+-----------------------------+--------------------------------------+
```

2.9 PACF

2.9.1 函数简介

本函数通过求解Yule-Walker方程，计算序列的偏自相关系数。

函数名： PACF

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

参数：

- `lag`：最大滞后阶数。默认值为 $\min(10\log_{10}n, n-1)$ ，n 表示数据点个数。

输出序列：输出单个序列，类型为DOUBLE。

2.9.2 使用示例

2.9.2.1 指定滞后阶数

输入序列：

```
+-----------------------------+-----------+
|                         Time|root.test.s1|
+-----------------------------+-----------+
```

```
|2019-12-27T00:00:00.000+08:00|                5.0|
|2019-12-27T00:05:00.000+08:00|                5.0|
|2019-12-27T00:10:00.000+08:00|                5.0|
|2019-12-27T00:15:00.000+08:00|                5.0|
|2019-12-27T00:20:00.000+08:00|                6.0|
|2019-12-27T00:25:00.000+08:00|                5.0|
|2019-12-27T00:30:00.000+08:00|                6.0|
|2019-12-27T00:35:00.000+08:00|                6.0|
|2019-12-27T00:40:00.000+08:00|                6.0|
|2019-12-27T00:45:00.000+08:00|                6.0|
|2019-12-27T00:50:00.000+08:00|                6.0|
|2019-12-27T00:55:00.000+08:00|           5.982609|
|2019-12-27T01:00:00.000+08:00|          5.9652176|
|2019-12-27T01:05:00.000+08:00|           5.947826|
|2019-12-27T01:10:00.000+08:00|          5.9304347|
|2019-12-27T01:15:00.000+08:00|          5.9130435|
|2019-12-27T01:20:00.000+08:00|          5.8956523|
|2019-12-27T01:25:00.000+08:00|           5.878261|
|2019-12-27T01:30:00.000+08:00|          5.8608694|
|2019-12-27T01:35:00.000+08:00|           5.843478|
.............
Total line number = 18066
```

用于查询的SQL语句：

```
select pacf(s1, "lag"="5") from root.test
```

输出序列：

```
+-----------------------------+-----------------------------------+
|                         Time|pacf(root.test.s1, "lag"="5")|
+-----------------------------+-----------------------------------+
|2019-12-27T00:00:00.000+08:00|                                1.0|
|2019-12-27T00:05:00.000+08:00|                0.3528915091942786|
|2019-12-27T00:10:00.000+08:00|                0.1761346122516304|
|2019-12-27T00:15:00.000+08:00|                0.1492391973294682|
|2019-12-27T00:20:00.000+08:00|               0.03560059645868398|
|2019-12-27T00:25:00.000+08:00|                0.0366222998995286|
+-----------------------------+-----------------------------------+
```

2.10 Percentile

2.10.1 函数简介

本函数用于计算单列数值型数据的精确或近似分位数。

函数名： PERCENTILE

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

参数：

- `rank`：所求分位数在所有数据中的排名百分比，取值范围为(0,1]，默认值为0.5。如当设为0.5时则计算中位数。
- `error`：近似分位数的基于排名的误差百分比，取值范围为[0,1)，默认值为0。如`rank`=0.5且`error`=0.0则计算出的分位数的真实排名百分比在0.49~0.51之间。当`error`=0时，计算结果为精确分位数。

输出序列： 输出单个序列，类型为DOUBLE，序列仅包含一个时间戳为0、值为分位数的数据点。

提示： 数据中的空值、缺失值和`NaN`将会被忽略。

2.10.2 使用示例

输入序列：

```
+-----------------------------+-----------+
|                         Time|root.test.s0|
+-----------------------------+-----------+
|2021-03-17T10:32:17.054+08:00|  0.5319929|
|2021-03-17T10:32:18.054+08:00|  0.9304316|
|2021-03-17T10:32:19.054+08:00| -1.4800133|
|2021-03-17T10:32:20.054+08:00|  0.6114087|
|2021-03-17T10:32:21.054+08:00|  2.5163336|
|2021-03-17T10:32:22.054+08:00| -1.0845392|
|2021-03-17T10:32:23.054+08:00|  1.0562582|
|2021-03-17T10:32:24.054+08:00|  1.3867859|
|2021-03-17T10:32:25.054+08:00| -0.45429882|
|2021-03-17T10:32:26.054+08:00|  1.0353678|
|2021-03-17T10:32:27.054+08:00|  0.7307929|
|2021-03-17T10:32:28.054+08:00|  2.3167255|
|2021-03-17T10:32:29.054+08:00|   2.342443|
|2021-03-17T10:32:30.054+08:00|  1.5809103|
|2021-03-17T10:32:31.054+08:00|  1.4829416|
|2021-03-17T10:32:32.054+08:00|  1.5800357|
|2021-03-17T10:32:33.054+08:00|  0.7124368|
|2021-03-17T10:32:34.054+08:00| -0.78597564|
|2021-03-17T10:32:35.054+08:00|  1.2058644|
|2021-03-17T10:32:36.054+08:00|  1.4215064|
|2021-03-17T10:32:37.054+08:00|  1.2808295|
|2021-03-17T10:32:38.054+08:00| -0.6173715|
|2021-03-17T10:32:39.054+08:00| 0.06644377|
|2021-03-17T10:32:40.054+08:00|   2.349338|
|2021-03-17T10:32:41.054+08:00|  1.7335888|
|2021-03-17T10:32:42.054+08:00|  1.5872132|
.............
Total line number = 10000
```

用于查询的SQL语句：

```
select percentile(s0, "rank"="0.2", "error"="0.01") from root.test
```

输出序列：

```
+-----------------------------+-----------------------------------------------------+
|                         Time|percentile(root.test.s0, "rank"="0.2", "error"="0.01")|
+-----------------------------+-----------------------------------------------------+
|1970-01-01T08:00:00.000+08:00|                                   0.1801469624042511|
+-----------------------------+-----------------------------------------------------+
```

2.11 Period

2.11.1 函数简介

本函数用于计算单列数值型数据的周期。

函数名： PERIOD

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

输出序列： 输出单个序列，类型为INT32，序列仅包含一个时间戳为0、值为周期的数据点。

2.11.2 使用示例

输入序列：

```
+-----------------------------+---------------+
|                         Time|root.test.d3.s1|
+-----------------------------+---------------+
|1970-01-01T08:00:00.001+08:00|            1.0|
|1970-01-01T08:00:00.002+08:00|            2.0|
|1970-01-01T08:00:00.003+08:00|            3.0|
|1970-01-01T08:00:00.004+08:00|            1.0|
|1970-01-01T08:00:00.005+08:00|            2.0|
|1970-01-01T08:00:00.006+08:00|            3.0|
|1970-01-01T08:00:00.007+08:00|            1.0|
|1970-01-01T08:00:00.008+08:00|            2.0|
|1970-01-01T08:00:00.009+08:00|            3.0|
+-----------------------------+---------------+
```

用于查询的SQL语句：

```
select period(s1) from root.test.d3
```

输出序列：

```
+-----------------------------+-----------------------+
|                         Time|period(root.test.d3.s1)|
```

```
+-----------------------------+---------------------+
|1970-01-01T08:00:00.000+08:00|                    3|
+-----------------------------+---------------------+
```

2.11.2.1 Zeppelin示例

链接: <http://101.6.15.213:18181/#/notebook/2GEJBUSZ9>

2.12 QLB

2.12.1 函数简介

本函数对输入序列计算 Q_{LB} 统计量，并计算对应的p值。p值越小表明序列越有可能为非平稳序列。

函数名： QLB

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

参数：

- shift：计算时用到的最大延迟阶数，取值应为1至n-2之间的整数，n为序列采样总数。默认取n-2。

输出序列： 输出单个序列，类型为DOUBLE。该序列是 Q_{LB} 统计量对应的p值，时间标签代表偏移阶数。

提示： Q_{LB} 统计量由自相关系数求得，如需得到统计量而非p值，可以使用AutoCorrelation函数。

2.12.2 使用示例

2.12.2.1 使用默认参数

输入序列：

```
+-----------------------------+--------------+
|                         Time|root.test.d1.s1|
+-----------------------------+--------------+
|1970-01-01T00:00:00.100+08:00|          1.22|
|1970-01-01T00:00:00.200+08:00|         -2.78|
|1970-01-01T00:00:00.300+08:00|          1.53|
|1970-01-01T00:00:00.400+08:00|          0.70|
|1970-01-01T00:00:00.500+08:00|          0.75|
|1970-01-01T00:00:00.600+08:00|         -0.72|
|1970-01-01T00:00:00.700+08:00|         -0.22|
|1970-01-01T00:00:00.800+08:00|          0.28|
|1970-01-01T00:00:00.900+08:00|          0.57|
|1970-01-01T00:00:01.000+08:00|         -0.22|
|1970-01-01T00:00:01.100+08:00|         -0.72|
```

```
|1970-01-01T00:00:01.200+08:00|                1.34|
|1970-01-01T00:00:01.300+08:00|               -0.25|
|1970-01-01T00:00:01.400+08:00|                0.17|
|1970-01-01T00:00:01.500+08:00|                2.51|
|1970-01-01T00:00:01.600+08:00|                1.42|
|1970-01-01T00:00:01.700+08:00|               -1.34|
|1970-01-01T00:00:01.800+08:00|               -0.01|
|1970-01-01T00:00:01.900+08:00|               -0.49|
|1970-01-01T00:00:02.000+08:00|                1.63|
+-----------------------------+--------------------+
```

用于查询的SQL语句：

```
select QLB(s1) from root.test.d1
```

输出序列：

```
+-----------------------------+--------------------+
|                         Time|QLB(root.test.d1.s1)|
+-----------------------------+--------------------+
|1970-01-01T00:00:00.001+08:00|  0.2168702295315677|
|1970-01-01T00:00:00.002+08:00|  0.3068948509261751|
|1970-01-01T00:00:00.003+08:00|  0.4217859150918444|
|1970-01-01T00:00:00.004+08:00|  0.5114539874276656|
|1970-01-01T00:00:00.005+08:00|  0.6560619525616759|
|1970-01-01T00:00:00.006+08:00|  0.7722398654053280|
|1970-01-01T00:00:00.007+08:00|  0.8532491661465290|
|1970-01-01T00:00:00.008+08:00|  0.9028575017542528|
|1970-01-01T00:00:00.009+08:00|  0.9434989988192729|
|1970-01-01T00:00:00.010+08:00|  0.8950280161464689|
|1970-01-01T00:00:00.011+08:00|  0.7701048398839656|
|1970-01-01T00:00:00.012+08:00|  0.7845536060001281|
|1970-01-01T00:00:00.013+08:00|  0.5943030981705825|
|1970-01-01T00:00:00.014+08:00|  0.4618413512531093|
|1970-01-01T00:00:00.015+08:00|  0.2645948244673964|
|1970-01-01T00:00:00.016+08:00|  0.3167530476666645|
|1970-01-01T00:00:00.017+08:00|  0.2330010780351453|
|1970-01-01T00:00:00.018+08:00|  0.0666611237622325|
+-----------------------------+--------------------+
```

2.13 Re_sample

2.13.1 函数简介

本函数对输入序列按照指定的频率进行重采样，包括上采样和下采样。目前，本函数支持的上采样方法包括`NaN`填充法(NaN)、前值填充法(FFill)、后值填充法(BFill)以及

线性插值法(Linear)；本函数支持的下采样方法为分组聚合，聚合方法包括最大值(Max)、最小值(Min)、首值(First)、末值(Last)、平均值(Mean)和中位数(Median)。

函数名： RE_SAMPLE

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

参数：

- `every`：重采样频率，是一个有单位的正数。目前支持五种单位，分别是'ms'（毫秒）、's'（秒）、'm'（分钟）、'h'（小时）和'd'（天）。该参数不允许缺省。
- `interp`：上采样的插值方法，取值为'NaN'、'FFill'、'BFill'或'Linear'。在缺省情况下，使用`NaN`填充法。
- `aggr`：下采样的聚合方法，取值为'Max'、'Min'、'First'、'Last'、'Mean'或'Median'。在缺省情况下，使用平均数聚合。
- `start`：重采样的起始时间（包含），是一个格式为'yyyy-MM-dd HH:mm:ss'的时间字符串。在缺省情况下，使用第一个有效数据点的时间戳。
- `end`：重采样的结束时间（不包含），是一个格式为'yyyy-MM-dd HH:mm:ss'的时间字符串。在缺省情况下，使用最后一个有效数据点的时间戳。

输出序列： 输出单个序列，类型为DOUBLE。该序列按照重采样频率严格等间隔分布。

提示： 数据中的`NaN`将会被忽略。

2.13.2 使用示例

2.13.2.1 上采样

当重采样频率高于数据原始频率时，将会进行上采样。

输入序列：

```
+-----------------------------+-------------+
|                         Time|root.test.d1.s1|
+-----------------------------+-------------+
|2021-03-06T16:00:00.000+08:00|         3.09|
|2021-03-06T16:15:00.000+08:00|         3.53|
|2021-03-06T16:30:00.000+08:00|          3.5|
|2021-03-06T16:45:00.000+08:00|         3.51|
|2021-03-06T17:00:00.000+08:00|         3.41|
+-----------------------------+-------------+
```

用于查询的SQL语句：

```
select re_sample(s1,'every'='5m','interp'='linear') from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------------------+
|                         Time|re_sample(root.test.d1.s1, "every"="5m", "interp"="linear")|
+-----------------------------+-----------------------------------------------------------+
```

```
|2021-03-06T16:00:00.000+08:00|                                3.0899999141693115|
|2021-03-06T16:05:00.000+08:00|                                3.2366665999094644|
|2021-03-06T16:10:00.000+08:00|                                3.3833332856496177|
|2021-03-06T16:15:00.000+08:00|                                3.5299999713897705|
|2021-03-06T16:20:00.000+08:00|                                3.5199999809265137|
|2021-03-06T16:25:00.000+08:00|                                 3.509999990463257|
|2021-03-06T16:30:00.000+08:00|                                               3.5|
|2021-03-06T16:35:00.000+08:00|                                3.503333330154419|
|2021-03-06T16:40:00.000+08:00|                                3.506666660308838|
|2021-03-06T16:45:00.000+08:00|                                 3.509999990463257|
|2021-03-06T16:50:00.000+08:00|                                3.4766666889190674|
|2021-03-06T16:55:00.000+08:00|                                 3.443333387374878|
|2021-03-06T17:00:00.000+08:00|                                3.4100000858306885|
+-----------------------------+-----------------------------------------------+
```

2.13.2.2 下采样

当重采样频率低于数据原始频率时，将会进行下采样。

输入序列同上，用于查询的SQL语句如下：

```
select re_sample(s1,'every'='30m','aggr'='first') from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------+
|                         Time|re_sample(root.test.d1.s1, "every"="30m", "aggr"="first")|
+-----------------------------+-----------------------------------------------+
|2021-03-06T16:00:00.000+08:00|                                3.0899999141693115|
|2021-03-06T16:30:00.000+08:00|                                               3.5|
|2021-03-06T17:00:00.000+08:00|                                3.4100000858306885|
+-----------------------------+-----------------------------------------------+
```

2.13.2.3 指定重采样时间段

可以使用 start 和 end 两个参数指定重采样的时间段，超出实际时间范围的部分会被插值填补。

输入序列同上，用于查询的SQL语句如下：

```
select re_sample(s1,'every'='30m','start'='2021-03-06 15:00:00') from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------------------------------+
|                         Time|re_sample(root.test.d1.s1, "every"="30m", "start"="2021-03-06 15:00:00")|
+-----------------------------+-----------------------------------------------------------------------+
```

```
|2021-03-06T15:00:00.000+08:00|                                                   NaN
        |
|2021-03-06T15:30:00.000+08:00|                                                   NaN
        |
|2021-03-06T16:00:00.000+08:00|
        3.309999942779541|
|2021-03-06T16:30:00.000+08:00|
        3.5049999952316284|
|2021-03-06T17:00:00.000+08:00|
        3.4100000858306885|
+-----------------------------+----------------------------------------------------------+
```

2.14 Sample

2.14.1 函数简介

本函数对输入序列进行采样，即从输入序列中选取指定数量的数据点并输出。目前，本函数支持两种采样方法：**蓄水池采样法（reservoir sampling）** 对数据进行随机采样，所有数据点被采样的概率相同；**等距采样法（isometric sampling）** 按照相等的索引间隔对数据进行采样。

函数名： SAMPLE

输入序列： 仅支持单个输入序列，类型可以是任意的。

参数：

- `method`：采样方法，取值为'reservoir'或'isometric'。在缺省情况下，采用蓄水池采样法。
- `k`：采样数，它是一个正整数，在缺省情况下为1。

输出序列： 输出单个序列，类型与输入序列相同。该序列的长度为采样数，序列中的每一个数据点都来自于输入序列。

提示： 如果采样数大于序列长度，那么输入序列中所有的数据点都会被输出。

2.14.2 使用示例

2.14.2.1 蓄水池采样

当 `method` 参数为'reservoir'或缺省时，采用蓄水池采样法对输入序列进行采样。由于该采样方法具有随机性，下面展示的输出序列只是一种可能的结果。

输入序列：

```
+-----------------------------+-------------+
|                         Time|root.test.d1.s1|
+-----------------------------+-------------+
|2020-01-01T00:00:01.000+08:00|          1.0|
```

```
|2020-01-01T00:00:02.000+08:00|                2.0|
|2020-01-01T00:00:03.000+08:00|                3.0|
|2020-01-01T00:00:04.000+08:00|                4.0|
|2020-01-01T00:00:05.000+08:00|                5.0|
|2020-01-01T00:00:06.000+08:00|                6.0|
|2020-01-01T00:00:07.000+08:00|                7.0|
|2020-01-01T00:00:08.000+08:00|                8.0|
|2020-01-01T00:00:09.000+08:00|                9.0|
|2020-01-01T00:00:10.000+08:00|               10.0|
+-----------------------------+--------------------+
```

用于查询的SQL语句：

```
select sample(s1,'method'='reservoir','k'='5') from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------------------+
|                         Time|sample(root.test.d1.s1, "method"="reservoir", "k"="5")|
+-----------------------------+-----------------------------------------------------------+
|2020-01-01T00:00:02.000+08:00|                                                        2.0|
|2020-01-01T00:00:03.000+08:00|                                                        3.0|
|2020-01-01T00:00:05.000+08:00|                                                        5.0|
|2020-01-01T00:00:08.000+08:00|                                                        8.0|
|2020-01-01T00:00:10.000+08:00|                                                       10.0|
+-----------------------------+-----------------------------------------------------------+
```

2.14.2.2 等距采样

当 method 参数为'isometric'时，采用等距采样法对输入序列进行采样。

输入序列同上，用于查询的SQL语句如下：

```
select sample(s1,'method'='isometric','k'='5') from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------------------+
|                         Time|sample(root.test.d1.s1, "method"="isometric", "k"="5")|
+-----------------------------+-----------------------------------------------------------+
|2020-01-01T00:00:01.000+08:00|                                                        1.0|
|2020-01-01T00:00:03.000+08:00|                                                        3.0|
|2020-01-01T00:00:05.000+08:00|                                                        5.0|
|2020-01-01T00:00:07.000+08:00|                                                        7.0|
|2020-01-01T00:00:09.000+08:00|                                                        9.0|
+-----------------------------+-----------------------------------------------------------+
```

2.15 Segment

2.15.1 函数简介

本函数按照数据的线性变化趋势将数据划分为多个子序列，返回分段直线拟合后的子序列首值或所有拟合值。

函数名： SEGMENT

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- `output`：“all”输出所有拟合值；“first”输出子序列起点拟合值。默认为“first”。
- `error`：判定存在线性趋势的误差允许阈值。误差的定义为子序列进行线性拟合的误差的绝对值的均值。默认为0.1.

输出序列： 输出单个序列，类型为DOUBLE。

提示： 函数默认所有数据等时间间隔分布。函数读取所有数据，若原始数据过多，请先进行降采样处理。拟合采用自底向上方法，子序列的尾值可能会被认作子序列首值输出。

2.15.2 使用示例

输入序列：

```
+-----------------------------+------------+
|                         Time|root.test.s1|
+-----------------------------+------------+
|1970-01-01T08:00:00.000+08:00|         5.0|
|1970-01-01T08:00:00.100+08:00|         0.0|
|1970-01-01T08:00:00.200+08:00|         1.0|
|1970-01-01T08:00:00.300+08:00|         2.0|
|1970-01-01T08:00:00.400+08:00|         3.0|
|1970-01-01T08:00:00.500+08:00|         4.0|
|1970-01-01T08:00:00.600+08:00|         5.0|
|1970-01-01T08:00:00.700+08:00|         6.0|
|1970-01-01T08:00:00.800+08:00|         7.0|
|1970-01-01T08:00:00.900+08:00|         8.0|
|1970-01-01T08:00:01.000+08:00|         9.0|
|1970-01-01T08:00:01.100+08:00|         9.1|
|1970-01-01T08:00:01.200+08:00|         9.2|
|1970-01-01T08:00:01.300+08:00|         9.3|
|1970-01-01T08:00:01.400+08:00|         9.4|
|1970-01-01T08:00:01.500+08:00|         9.5|
|1970-01-01T08:00:01.600+08:00|         9.6|
|1970-01-01T08:00:01.700+08:00|         9.7|
|1970-01-01T08:00:01.800+08:00|         9.8|
|1970-01-01T08:00:01.900+08:00|         9.9|
|1970-01-01T08:00:02.000+08:00|        10.0|
```

```
|1970-01-01T08:00:02.100+08:00|           8.0|
|1970-01-01T08:00:02.200+08:00|           6.0|
|1970-01-01T08:00:02.300+08:00|           4.0|
|1970-01-01T08:00:02.400+08:00|           2.0|
|1970-01-01T08:00:02.500+08:00|           0.0|
|1970-01-01T08:00:02.600+08:00|          -2.0|
|1970-01-01T08:00:02.700+08:00|          -4.0|
|1970-01-01T08:00:02.800+08:00|          -6.0|
|1970-01-01T08:00:02.900+08:00|          -8.0|
|1970-01-01T08:00:03.000+08:00|         -10.0|
|1970-01-01T08:00:03.100+08:00|          10.0|
|1970-01-01T08:00:03.200+08:00|          10.0|
|1970-01-01T08:00:03.300+08:00|          10.0|
|1970-01-01T08:00:03.400+08:00|          10.0|
|1970-01-01T08:00:03.500+08:00|          10.0|
|1970-01-01T08:00:03.600+08:00|          10.0|
|1970-01-01T08:00:03.700+08:00|          10.0|
|1970-01-01T08:00:03.800+08:00|          10.0|
|1970-01-01T08:00:03.900+08:00|          10.0|
+-----------------------------+---------------+
```

用于查询的SQL语句：

```
select segment(s1,"error"="0.1") from root.test
```

输出序列：

```
+-----------------------------+-----------------------------------------+
|                         Time|segment(root.test.s1, "error"="0.1")|
+-----------------------------+-----------------------------------------+
|1970-01-01T08:00:00.000+08:00|                                      5.0|
|1970-01-01T08:00:00.200+08:00|                                      1.0|
|1970-01-01T08:00:01.000+08:00|                                      9.0|
|1970-01-01T08:00:02.000+08:00|                                     10.0|
|1970-01-01T08:00:03.000+08:00|                                    -10.0|
|1970-01-01T08:00:03.200+08:00|                                     10.0|
+-----------------------------+-----------------------------------------+
```

2.16 Skew

2.16.1 函数简介

本函数用于计算单列数值型数据的总体偏度

函数名： SKEW

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

输出序列： 输出单个序列，类型为DOUBLE，序列仅包含一个时间戳为0、值为总体偏度的数据点。

提示： 数据中的空值、缺失值和 NaN 将会被忽略。

2.16.2 使用示例

输入序列：

```
+-----------------------------+-------------+
|                         Time|root.test.d1.s1|
+-----------------------------+-------------+
|2020-01-01T00:00:00.000+08:00|          1.0|
|2020-01-01T00:00:01.000+08:00|          2.0|
|2020-01-01T00:00:02.000+08:00|          3.0|
|2020-01-01T00:00:03.000+08:00|          4.0|
|2020-01-01T00:00:04.000+08:00|          5.0|
|2020-01-01T00:00:05.000+08:00|          6.0|
|2020-01-01T00:00:06.000+08:00|          7.0|
|2020-01-01T00:00:07.000+08:00|          8.0|
|2020-01-01T00:00:08.000+08:00|          9.0|
|2020-01-01T00:00:09.000+08:00|         10.0|
|2020-01-01T00:00:10.000+08:00|         10.0|
|2020-01-01T00:00:11.000+08:00|         10.0|
|2020-01-01T00:00:12.000+08:00|         10.0|
|2020-01-01T00:00:13.000+08:00|         10.0|
|2020-01-01T00:00:14.000+08:00|         10.0|
|2020-01-01T00:00:15.000+08:00|         10.0|
|2020-01-01T00:00:16.000+08:00|         10.0|
|2020-01-01T00:00:17.000+08:00|         10.0|
|2020-01-01T00:00:18.000+08:00|         10.0|
|2020-01-01T00:00:19.000+08:00|         10.0|
+-----------------------------+-------------+
```

用于查询的SQL语句：

```
select skew(s1) from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------+
|                         Time|  skew(root.test.d1.s1)|
+-----------------------------+-----------------------+
|1970-01-01T08:00:00.000+08:00|    -0.9998427402292644|
+-----------------------------+-----------------------+
```

2.17 Spline

2.17.1 函数简介

本函数提供对原始序列进行三次样条曲线拟合后的插值重采样。

函数名： SPLINE

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

参数：

- points：重采样个数。

输出序列：输出单个序列，类型为DOUBLE。

提示：输出序列保留输入序列的首尾值，等时间间隔采样。仅当输入点个数不少于4个时才计算插值。

2.17.2 使用示例

2.17.2.1 指定插值个数

输入序列：

```
+-----------------------------+-----------+
|                         Time|root.test.s1|
+-----------------------------+-----------+
|1970-01-01T08:00:00.000+08:00|        0.0|
|1970-01-01T08:00:00.300+08:00|        1.2|
|1970-01-01T08:00:00.500+08:00|        1.7|
|1970-01-01T08:00:00.700+08:00|        2.0|
|1970-01-01T08:00:00.900+08:00|        2.1|
|1970-01-01T08:00:01.100+08:00|        2.0|
|1970-01-01T08:00:01.200+08:00|        1.8|
|1970-01-01T08:00:01.300+08:00|        1.2|
|1970-01-01T08:00:01.400+08:00|        1.0|
|1970-01-01T08:00:01.500+08:00|        1.6|
+-----------------------------+-----------+
```

用于查询的SQL语句：

```
select spline(s1, "points"="151") from root.test
```

输出序列：

```
+-----------------------------+-----------------------------------+
|                         Time|spline(root.test.s1, "points"="151")|
+-----------------------------+-----------------------------------+
|1970-01-01T08:00:00.000+08:00|                                0.0|
|1970-01-01T08:00:00.010+08:00|                0.04870000251134237|
|1970-01-01T08:00:00.020+08:00|                0.09680000495910646|
|1970-01-01T08:00:00.030+08:00|                0.14430000734329226|
```

|1970–01–01T08:00:00.040+08:00| 0.19120000966389972|
|1970–01–01T08:00:00.050+08:00| 0.23750001192092896|
|1970–01–01T08:00:00.060+08:00| 0.2832000141143799|
|1970–01–01T08:00:00.070+08:00| 0.32830001624425253|
|1970–01–01T08:00:00.080+08:00| 0.3728000183105469|
|1970–01–01T08:00:00.090+08:00| 0.416700020313263|
|1970–01–01T08:00:00.100+08:00| 0.4600000222524008|
|1970–01–01T08:00:00.110+08:00| 0.5027000241279602|
|1970–01–01T08:00:00.120+08:00| 0.5448000259399414|
|1970–01–01T08:00:00.130+08:00| 0.5863000276883443|
|1970–01–01T08:00:00.140+08:00| 0.627200029373169|
|1970–01–01T08:00:00.150+08:00| 0.6675000309944153|
|1970–01–01T08:00:00.160+08:00| 0.7072000325520833|
|1970–01–01T08:00:00.170+08:00| 0.7463000340461731|
|1970–01–01T08:00:00.180+08:00| 0.7848000354766846|
|1970–01–01T08:00:00.190+08:00| 0.8227000368436178|
|1970–01–01T08:00:00.200+08:00| 0.8600000381469728|
|1970–01–01T08:00:00.210+08:00| 0.8967000393867494|
|1970–01–01T08:00:00.220+08:00| 0.9328000405629477|
|1970–01–01T08:00:00.230+08:00| 0.9683000416755676|
|1970–01–01T08:00:00.240+08:00| 1.0032000427246095|
|1970–01–01T08:00:00.250+08:00| 1.037500043710073|
|1970–01–01T08:00:00.260+08:00| 1.071200044631958|
|1970–01–01T08:00:00.270+08:00| 1.1043000454902647|
|1970–01–01T08:00:00.280+08:00| 1.1368000462849934|
|1970–01–01T08:00:00.290+08:00| 1.1687000470161437|
|1970–01–01T08:00:00.300+08:00| 1.2000000476837158|
|1970–01–01T08:00:00.310+08:00| 1.2307000483103594|
|1970–01–01T08:00:00.320+08:00| 1.2608000489139557|
|1970–01–01T08:00:00.330+08:00| 1.2903000494873524|
|1970–01–01T08:00:00.340+08:00| 1.3192000500233967|
|1970–01–01T08:00:00.350+08:00| 1.3475000505149364|
|1970–01–01T08:00:00.360+08:00| 1.3752000509548186|
|1970–01–01T08:00:00.370+08:00| 1.402300051335891|
|1970–01–01T08:00:00.380+08:00| 1.4288000516510009|
|1970–01–01T08:00:00.390+08:00| 1.4547000518929958|
|1970–01–01T08:00:00.400+08:00| 1.480000052054723|
|1970–01–01T08:00:00.410+08:00| 1.5047000521290301|
|1970–01–01T08:00:00.420+08:00| 1.5288000521087646|
|1970–01–01T08:00:00.430+08:00| 1.5523000519867738|
|1970–01–01T08:00:00.440+08:00| 1.575200051755905|
|1970–01–01T08:00:00.450+08:00| 1.597500051409006|
|1970–01–01T08:00:00.460+08:00| 1.619200050938924|
|1970–01–01T08:00:00.470+08:00| 1.6403000503385066|
|1970–01–01T08:00:00.480+08:00| 1.660800049600601|
|1970–01–01T08:00:00.490+08:00| 1.680700048718055|
|1970–01–01T08:00:00.500+08:00| 1.7000000476837158|

|1970–01–01T08:00:00.510+08:00| 1.7188475466453037|
|1970–01–01T08:00:00.520+08:00| 1.7373800457262996|
|1970–01–01T08:00:00.530+08:00| 1.7555825448831923|
|1970–01–01T08:00:00.540+08:00| 1.7734400440724702|
|1970–01–01T08:00:00.550+08:00| 1.790937543250622|
|1970–01–01T08:00:00.560+08:00| 1.8080600423741364|
|1970–01–01T08:00:00.570+08:00| 1.8247925413995016|
|1970–01–01T08:00:00.580+08:00| 1.8411200402832066|
|1970–01–01T08:00:00.590+08:00| 1.8570275389817397|
|1970–01–01T08:00:00.600+08:00| 1.8725000374515897|
|1970–01–01T08:00:00.610+08:00| 1.8875225356492449|
|1970–01–01T08:00:00.620+08:00| 1.902080033531194|
|1970–01–01T08:00:00.630+08:00| 1.9161575310539258|
|1970–01–01T08:00:00.640+08:00| 1.9297400281739288|
|1970–01–01T08:00:00.650+08:00| 1.9428125248476913|
|1970–01–01T08:00:00.660+08:00| 1.9553600210317021|
|1970–01–01T08:00:00.670+08:00| 1.96736751668245|
|1970–01–01T08:00:00.680+08:00| 1.9788200117564232|
|1970–01–01T08:00:00.690+08:00| 1.9897025062101101|
|1970–01–01T08:00:00.700+08:00| 2.0|
|1970–01–01T08:00:00.710+08:00| 2.0097024933913334|
|1970–01–01T08:00:00.720+08:00| 2.0188199867081615|
|1970–01–01T08:00:00.730+08:00| 2.027367479995188|
|1970–01–01T08:00:00.740+08:00| 2.0353599732971155|
|1970–01–01T08:00:00.750+08:00| 2.0428124666586482|
|1970–01–01T08:00:00.760+08:00| 2.049739960124489|
|1970–01–01T08:00:00.770+08:00| 2.056157453739342|
|1970–01–01T08:00:00.780+08:00| 2.06207994754791|
|1970–01–01T08:00:00.790+08:00| 2.067522441594897|
|1970–01–01T08:00:00.800+08:00| 2.072499935925006|
|1970–01–01T08:00:00.810+08:00| 2.07702743058294|
|1970–01–01T08:00:00.820+08:00| 2.081119925613404|
|1970–01–01T08:00:00.830+08:00| 2.0847924210611|
|1970–01–01T08:00:00.840+08:00| 2.0880599169707317|
|1970–01–01T08:00:00.850+08:00| 2.0909374133870027|
|1970–01–01T08:00:00.860+08:00| 2.0934399103546166|
|1970–01–01T08:00:00.870+08:00| 2.0955824079182768|
|1970–01–01T08:00:00.880+08:00| 2.0973799061226863|
|1970–01–01T08:00:00.890+08:00| 2.098847405012549|
|1970–01–01T08:00:00.900+08:00| 2.0999999046325684|
|1970–01–01T08:00:00.910+08:00| 2.1005574051201332|
|1970–01–01T08:00:00.920+08:00| 2.1002599065303778|
|1970–01–01T08:00:00.930+08:00| 2.0991524087846245|
|1970–01–01T08:00:00.940+08:00| 2.0972799118041947|
|1970–01–01T08:00:00.950+08:00| 2.0946874155104105|
|1970–01–01T08:00:00.960+08:00| 2.0914199198245944|
|1970–01–01T08:00:00.970+08:00| 2.0875224246680673|

|1970–01–01T08:00:00.980+08:00| 2.083039929962151|
|1970–01–01T08:00:00.990+08:00| 2.0780174356281687|
|1970–01–01T08:00:01.000+08:00| 2.0724999415874406|
|1970–01–01T08:00:01.010+08:00| 2.06653244776129|
|1970–01–01T08:00:01.020+08:00| 2.060159954071038|
|1970–01–01T08:00:01.030+08:00| 2.053427460438006|
|1970–01–01T08:00:01.040+08:00| 2.046379966783517|
|1970–01–01T08:00:01.050+08:00| 2.0390624730288924|
|1970–01–01T08:00:01.060+08:00| 2.031519979095454|
|1970–01–01T08:00:01.070+08:00| 2.0237974849045237|
|1970–01–01T08:00:01.080+08:00| 2.015939990377423|
|1970–01–01T08:00:01.090+08:00| 2.0079924954354746|
|1970–01–01T08:00:01.100+08:00| 2.0|
|1970–01–01T08:00:01.110+08:00| 1.9907018211101906|
|1970–01–01T08:00:01.120+08:00| 1.9788509124245144|
|1970–01–01T08:00:01.130+08:00| 1.9645127287932083|
|1970–01–01T08:00:01.140+08:00| 1.9477527250665083|
|1970–01–01T08:00:01.150+08:00| 1.9286363560946513|
|1970–01–01T08:00:01.160+08:00| 1.9072290767278735|
|1970–01–01T08:00:01.170+08:00| 1.8835963418164114|
|1970–01–01T08:00:01.180+08:00| 1.8578036062105014|
|1970–01–01T08:00:01.190+08:00| 1.8299163247603802|
|1970–01–01T08:00:01.200+08:00| 1.7999999523162842|
|1970–01–01T08:00:01.210+08:00| 1.7623635841923329|
|1970–01–01T08:00:01.220+08:00| 1.7129696477516976|
|1970–01–01T08:00:01.230+08:00| 1.6543635959181928|
|1970–01–01T08:00:01.240+08:00| 1.5890908816156328|
|1970–01–01T08:00:01.250+08:00| 1.5196969577678319|
|1970–01–01T08:00:01.260+08:00| 1.4487272772986044|
|1970–01–01T08:00:01.270+08:00| 1.3787272931317647|
|1970–01–01T08:00:01.280+08:00| 1.3122424581911272|
|1970–01–01T08:00:01.290+08:00| 1.251818225400506|
|1970–01–01T08:00:01.300+08:00| 1.2000000476837158|
|1970–01–01T08:00:01.310+08:00| 1.1548000470995912|
|1970–01–01T08:00:01.320+08:00| 1.1130667107899999|
|1970–01–01T08:00:01.330+08:00| 1.0756000393033045|
|1970–01–01T08:00:01.340+08:00| 1.043200033187868|
|1970–01–01T08:00:01.350+08:00| 1.016666692992053|
|1970–01–01T08:00:01.360+08:00| 0.9968000192642223|
|1970–01–01T08:00:01.370+08:00| 0.9844000125527389|
|1970–01–01T08:00:01.380+08:00| 0.9802666734059655|
|1970–01–01T08:00:01.390+08:00| 0.9852000023722649|
|1970–01–01T08:00:01.400+08:00| 1.0|
|1970–01–01T08:00:01.410+08:00| 1.023999999165535|
|1970–01–01T08:00:01.420+08:00| 1.0559999990463256|
|1970–01–01T08:00:01.430+08:00| 1.0959999996423722|
|1970–01–01T08:00:01.440+08:00| 1.1440000009536744|

```
|1970-01-01T08:00:01.450+08:00|                1.2000000029802322|
|1970-01-01T08:00:01.460+08:00|                 1.264000005722046|
|1970-01-01T08:00:01.470+08:00|                1.3360000091791153|
|1970-01-01T08:00:01.480+08:00|                1.4160000133514405|
|1970-01-01T08:00:01.490+08:00|                1.5040000182390214|
|1970-01-01T08:00:01.500+08:00|                 1.600000023841858|
+-----------------------------+----------------------------------+
```

2.18 Spread

2.18.1 函数简介

本函数用于计算时间序列的极差，即最大值减去最小值的结果。

函数名： SPREAD

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

输出序列： 输出单个序列，类型与输入相同，序列仅包含一个时间戳为0、值为极差的数据点。

提示： 数据中的空值、缺失值和 NaN 将会被忽略。

2.18.2 使用示例

输入序列：

```
+-----------------------------+-------------+
|                         Time|root.test.d1.s1|
+-----------------------------+-------------+
|2020-01-01T00:00:02.000+08:00|        100.0|
|2020-01-01T00:00:03.000+08:00|        101.0|
|2020-01-01T00:00:04.000+08:00|        102.0|
|2020-01-01T00:00:06.000+08:00|        104.0|
|2020-01-01T00:00:08.000+08:00|        126.0|
|2020-01-01T00:00:10.000+08:00|        108.0|
|2020-01-01T00:00:14.000+08:00|        112.0|
|2020-01-01T00:00:15.000+08:00|        113.0|
|2020-01-01T00:00:16.000+08:00|        114.0|
|2020-01-01T00:00:18.000+08:00|        116.0|
|2020-01-01T00:00:20.000+08:00|        118.0|
|2020-01-01T00:00:22.000+08:00|        120.0|
|2020-01-01T00:00:26.000+08:00|        124.0|
|2020-01-01T00:00:28.000+08:00|        126.0|
|2020-01-01T00:00:30.000+08:00|          NaN|
+-----------------------------+-------------+
```

用于查询的SQL语句：

```
select spread(s1) from root.test.d1 where time <= 2020-01-01 00:00:30
```

输出序列：

```
+-----------------------------+-----------------------+
|                         Time|spread(root.test.d1.s1)|
+-----------------------------+-----------------------+
|1970-01-01T08:00:00.000+08:00|                   26.0|
+-----------------------------+-----------------------+
```

2.19 Stddev

2.19.1 函数简介

本函数用于计算单列数值型数据的总体标准差。

函数名： STDDEV

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

输出序列： 输出单个序列，类型为DOUBLE。序列仅包含一个时间戳为0、值为总体标准差的数据点。

提示： 数据中的空值、缺失值和 NaN 将会被忽略。

2.19.2 使用示例

输入序列：

```
+-----------------------------+--------------+
|                         Time|root.test.d1.s1|
+-----------------------------+--------------+
|2020-01-01T00:00:00.000+08:00|           1.0|
|2020-01-01T00:00:01.000+08:00|           2.0|
|2020-01-01T00:00:02.000+08:00|           3.0|
|2020-01-01T00:00:03.000+08:00|           4.0|
|2020-01-01T00:00:04.000+08:00|           5.0|
|2020-01-01T00:00:05.000+08:00|           6.0|
|2020-01-01T00:00:06.000+08:00|           7.0|
|2020-01-01T00:00:07.000+08:00|           8.0|
|2020-01-01T00:00:08.000+08:00|           9.0|
|2020-01-01T00:00:09.000+08:00|          10.0|
|2020-01-01T00:00:10.000+08:00|          11.0|
|2020-01-01T00:00:11.000+08:00|          12.0|
|2020-01-01T00:00:12.000+08:00|          13.0|
|2020-01-01T00:00:13.000+08:00|          14.0|
|2020-01-01T00:00:14.000+08:00|          15.0|
|2020-01-01T00:00:15.000+08:00|          16.0|
|2020-01-01T00:00:16.000+08:00|          17.0|
|2020-01-01T00:00:17.000+08:00|          18.0|
|2020-01-01T00:00:18.000+08:00|          19.0|
|2020-01-01T00:00:19.000+08:00|          20.0|
```

```
+-----------------------------+-------------+
```

用于查询的SQL语句：

```
select stddev(s1) from root.test.d1
```

输出序列：

```
+-----------------------------+-------------------------+
|                         Time|stddev(root.test.d1.s1)|
+-----------------------------+-------------------------+
|1970-01-01T08:00:00.000+08:00|       5.7662812973353965|
+-----------------------------+-------------------------+
```

2.20 TimeWeightedAvg

2.20.1 函数简介

本函数用于计算时间序列的时间加权平均值，即在相同时间单位下的数值积分除以序列总的时间跨度。更多关于数值积分计算的信息请参考 Integral 函数。

函数名： TIMEWEIGHTEDAVG

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

输出序列： 输出单个序列，类型为DOUBLE，序列仅包含一个时间戳为0、值为时间加权平均结果的数据点。

提示：

- 时间加权的平均值等于在任意时间单位 unit 下计算的数值积分（即折线图中每相邻两个数据点和时间轴形成的直角梯形的面积之和），

除以相同时间单位下输入序列的时间跨度，其值与具体采用的时间单位无关，默认与IoTDB时间单位一致。

- 数据中的 NaN 将会被忽略。折线将以临近两个有值数据点为准。
- 输入序列为空时，函数输出结果为0；仅有一个数据点时，输出结果为该点数值。

2.20.2 使用示例

输入序列：

```
+-----------------------------+--------------+
|                         Time|root.test.d1.s1|
+-----------------------------+--------------+
|2020-01-01T00:00:01.000+08:00|             1|
|2020-01-01T00:00:02.000+08:00|             2|
|2020-01-01T00:00:03.000+08:00|             5|
|2020-01-01T00:00:04.000+08:00|             6|
|2020-01-01T00:00:05.000+08:00|             7|
|2020-01-01T00:00:08.000+08:00|             8|
```

```
|2020-01-01T00:00:09.000+08:00|                  NaN|
|2020-01-01T00:00:10.000+08:00|                   10|
+-----------------------------+---------------------+
```

用于查询的SQL语句：

```
select timeweightedavg(s1) from root.test.d1 where time <= 2020-01-01 00:00:10
```

输出序列：

```
+-----------------------------+-----------------------------------------+
|                         Time|timeweightedavg(root.test.d1.s1)|
+-----------------------------+-----------------------------------------+
|1970-01-01T08:00:00.000+08:00|                                     5.75|
+-----------------------------+-----------------------------------------+
```

其计算公式为：

$$\frac{1}{2}[(1+2)\times 1+(2+5)\times 1+(5+6)\times 1+(6+7)\times 1+(7+8)\times 3+(8+10)\times 2]/10=5.75$$

2.21 ZScore

2.21.1 函数简介

本函数将输入序列使用z-score方法进行归一化。

函数名： ZSCORE

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

参数：

- `method`：若设置为"batch"，则将数据全部读入后转换；若设置为"stream"，则需用户提供均值及方差进行流式计算转换。默认为"batch"。
- `avg`：使用流式计算时的均值。
- `sd`：使用流式计算时的标准差。

输出序列： 输出单个序列，类型为DOUBLE。

2.21.2 使用示例

2.21.2.1 全数据计算

输入序列：

```
+-----------------------------+------------+
|                         Time|root.test.s1|
+-----------------------------+------------+
|1970-01-01T08:00:00.100+08:00|         0.0|
|1970-01-01T08:00:00.200+08:00|         0.0|
|1970-01-01T08:00:00.300+08:00|         1.0|
```

```
|1970-01-01T08:00:00.400+08:00|          -1.0|
|1970-01-01T08:00:00.500+08:00|           0.0|
|1970-01-01T08:00:00.600+08:00|           0.0|
|1970-01-01T08:00:00.700+08:00|          -2.0|
|1970-01-01T08:00:00.800+08:00|           2.0|
|1970-01-01T08:00:00.900+08:00|           0.0|
|1970-01-01T08:00:01.000+08:00|           0.0|
|1970-01-01T08:00:01.100+08:00|           1.0|
|1970-01-01T08:00:01.200+08:00|          -1.0|
|1970-01-01T08:00:01.300+08:00|          -1.0|
|1970-01-01T08:00:01.400+08:00|           1.0|
|1970-01-01T08:00:01.500+08:00|           0.0|
|1970-01-01T08:00:01.600+08:00|           0.0|
|1970-01-01T08:00:01.700+08:00|          10.0|
|1970-01-01T08:00:01.800+08:00|           2.0|
|1970-01-01T08:00:01.900+08:00|          -2.0|
|1970-01-01T08:00:02.000+08:00|           0.0|
+-----------------------------+---------------+
```

用于查询的SQL语句：

```
select zscore(s1) from root.test
```

输出序列：

```
+-----------------------------+--------------------+
|                         Time|zscore(root.test.s1)|
+-----------------------------+--------------------+
|1970-01-01T08:00:00.100+08:00|-0.20672455764868078|
|1970-01-01T08:00:00.200+08:00|-0.20672455764868078|
|1970-01-01T08:00:00.300+08:00| 0.20672455764868078|
|1970-01-01T08:00:00.400+08:00| -0.6201736729460423|
|1970-01-01T08:00:00.500+08:00|-0.20672455764868078|
|1970-01-01T08:00:00.600+08:00|-0.20672455764868078|
|1970-01-01T08:00:00.700+08:00|  -1.033622788243404|
|1970-01-01T08:00:00.800+08:00|  0.6201736729460423|
|1970-01-01T08:00:00.900+08:00|-0.20672455764868078|
|1970-01-01T08:00:01.000+08:00|-0.20672455764868078|
|1970-01-01T08:00:01.100+08:00| 0.20672455764868078|
|1970-01-01T08:00:01.200+08:00| -0.6201736729460423|
|1970-01-01T08:00:01.300+08:00| -0.6201736729460423|
|1970-01-01T08:00:01.400+08:00| 0.20672455764868078|
|1970-01-01T08:00:01.500+08:00|-0.20672455764868078|
|1970-01-01T08:00:01.600+08:00|-0.20672455764868078|
|1970-01-01T08:00:01.700+08:00|  3.9277665953249348|
|1970-01-01T08:00:01.800+08:00|  0.6201736729460423|
|1970-01-01T08:00:01.900+08:00|  -1.033622788243404|
|1970-01-01T08:00:02.000+08:00|-0.20672455764868078|
+-----------------------------+--------------------+
```

第 3 章 数据质量

3.1 Completeness

3.1.1 函数简介

本函数用于计算时间序列的完整性。将输入序列划分为若干个连续且不重叠的窗口，分别计算每一个窗口的完整性，并输出窗口第一个数据点的时间戳和窗口的完整性。

函数名： COMPLETENESS

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

参数：

- window：窗口大小，它是一个大于0的整数或者一个有单位的正数。前者代表每一个窗口包含的数据点数目，最后一个窗口的数据点数目可能会不足；后者代表窗口的时间跨度，目前支持五种单位，分别是'ms'（毫秒）、's'（秒）、'm'（分钟）、'h'（小时）和'd'（天）。缺省情况下，全部输入数据都属于同一个窗口。
- downtime：完整性计算是否考虑停机异常。它的取值为'true'或'false'，默认值为'true'。在考虑停机异常时，长时间的数据缺失将被视作停机，不对完整性产生影响。

输出序列： 输出单个序列，类型为DOUBLE，其中每一个数据点的值的范围都是[0,1]。

提示： 只有当窗口内的数据点数目超过10时，才会进行完整性计算。否则，该窗口将被忽略，不做任何输出。

3.1.2 使用示例

3.1.2.1 参数缺省

在参数缺省的情况下，本函数将会把全部输入数据都作为同一个窗口计算完整性。

输入序列：

```
+----------------------------+--------------+
|                        Time|root.test.d1.s1|
+----------------------------+--------------+
|2020-01-01T00:00:02.000+08:00|         100.0|
|2020-01-01T00:00:03.000+08:00|         101.0|
|2020-01-01T00:00:04.000+08:00|         102.0|
|2020-01-01T00:00:06.000+08:00|         104.0|
|2020-01-01T00:00:08.000+08:00|         126.0|
|2020-01-01T00:00:10.000+08:00|         108.0|
|2020-01-01T00:00:14.000+08:00|         112.0|
|2020-01-01T00:00:15.000+08:00|         113.0|
|2020-01-01T00:00:16.000+08:00|         114.0|
```

```
|2020-01-01T00:00:18.000+08:00|                116.0|
|2020-01-01T00:00:20.000+08:00|                118.0|
|2020-01-01T00:00:22.000+08:00|                120.0|
|2020-01-01T00:00:26.000+08:00|                124.0|
|2020-01-01T00:00:28.000+08:00|                126.0|
|2020-01-01T00:00:30.000+08:00|                 NaN|
+-----------------------------+--------------------+
```

用于查询的SQL语句：

```
select completeness(s1) from root.test.d1 where time <= 2020-01-01 00:00:30
```

输出序列：

```
+-----------------------------+-----------------------------------+
|                         Time|completeness(root.test.d1.s1)|
+-----------------------------+-----------------------------------+
|2020-01-01T00:00:02.000+08:00|                              0.875|
+-----------------------------+-----------------------------------+
```

3.1.2.2 指定窗口大小

在指定窗口大小的情况下，本函数会把输入数据划分为若干个窗口计算完整性。

输入序列:

```
+-----------------------------+--------------------+
|                         Time|root.test.d1.s1|
+-----------------------------+--------------------+
|2020-01-01T00:00:02.000+08:00|                100.0|
|2020-01-01T00:00:03.000+08:00|                101.0|
|2020-01-01T00:00:04.000+08:00|                102.0|
|2020-01-01T00:00:06.000+08:00|                104.0|
|2020-01-01T00:00:08.000+08:00|                126.0|
|2020-01-01T00:00:10.000+08:00|                108.0|
|2020-01-01T00:00:14.000+08:00|                112.0|
|2020-01-01T00:00:15.000+08:00|                113.0|
|2020-01-01T00:00:16.000+08:00|                114.0|
|2020-01-01T00:00:18.000+08:00|                116.0|
|2020-01-01T00:00:20.000+08:00|                118.0|
|2020-01-01T00:00:22.000+08:00|                120.0|
|2020-01-01T00:00:26.000+08:00|                124.0|
|2020-01-01T00:00:28.000+08:00|                126.0|
|2020-01-01T00:00:30.000+08:00|                 NaN|
|2020-01-01T00:00:32.000+08:00|                130.0|
|2020-01-01T00:00:34.000+08:00|                132.0|
|2020-01-01T00:00:36.000+08:00|                134.0|
|2020-01-01T00:00:38.000+08:00|                136.0|
|2020-01-01T00:00:40.000+08:00|                138.0|
```

```
|2020-01-01T00:00:42.000+08:00|                140.0|
|2020-01-01T00:00:44.000+08:00|                142.0|
|2020-01-01T00:00:46.000+08:00|                144.0|
|2020-01-01T00:00:48.000+08:00|                146.0|
|2020-01-01T00:00:50.000+08:00|                148.0|
|2020-01-01T00:00:52.000+08:00|                150.0|
|2020-01-01T00:00:54.000+08:00|                152.0|
|2020-01-01T00:00:56.000+08:00|                154.0|
|2020-01-01T00:00:58.000+08:00|                156.0|
|2020-01-01T00:01:00.000+08:00|                158.0|
+-----------------------------+--------------------+
```

用于查询的SQL语句：

```
select completeness(s1,"window"="15") from root.test.d1 where time <= 2020-01-01 00:01:00
```

输出序列：

```
+-----------------------------+-----------------------------------------------+
|                         Time|completeness(root.test.d1.s1, "window"="15")|
+-----------------------------+-----------------------------------------------+
|2020-01-01T00:00:02.000+08:00|                                          0.875|
|2020-01-01T00:00:32.000+08:00|                                            1.0|
+-----------------------------+-----------------------------------------------+
```

3.2 Consistency

3.2.1 函数简介

本函数用于计算时间序列的一致性。将输入序列划分为若干个连续且不重叠的窗口，分别计算每一个窗口的一致性，并输出窗口第一个数据点的时间戳和窗口的时效性。

函数名： CONSISTENCY

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- window：窗口大小，它是一个大于0的整数或者一个有单位的正数。前者代表每一个窗口包含的数据点数目，最后一个窗口的数据点数目可能会不足；后者代表窗口的时间跨度，目前支持五种单位，分别是’ms’（毫秒）、’s’（秒）、’m’（分钟）、’h’（小时）和’d’（天）。缺省情况下，全部输入数据都属于同一个窗口。

输出序列： 输出单个序列，类型为DOUBLE，其中每一个数据点的值的范围都是[0,1]。

提示： 只有当窗口内的数据点数目超过10时，才会进行一致性计算。否则，该窗口将被忽略，不做任何输出。

3.2.2 使用示例

3.2.2.1 参数缺省

在参数缺省的情况下，本函数将会把全部输入数据都作为同一个窗口计算一致性。

输入序列：

```
+----------------------------+--------------+
|                        Time|root.test.d1.s1|
+----------------------------+--------------+
|2020-01-01T00:00:02.000+08:00|         100.0|
|2020-01-01T00:00:03.000+08:00|         101.0|
|2020-01-01T00:00:04.000+08:00|         102.0|
|2020-01-01T00:00:06.000+08:00|         104.0|
|2020-01-01T00:00:08.000+08:00|         126.0|
|2020-01-01T00:00:10.000+08:00|         108.0|
|2020-01-01T00:00:14.000+08:00|         112.0|
|2020-01-01T00:00:15.000+08:00|         113.0|
|2020-01-01T00:00:16.000+08:00|         114.0|
|2020-01-01T00:00:18.000+08:00|         116.0|
|2020-01-01T00:00:20.000+08:00|         118.0|
|2020-01-01T00:00:22.000+08:00|         120.0|
|2020-01-01T00:00:26.000+08:00|         124.0|
|2020-01-01T00:00:28.000+08:00|         126.0|
|2020-01-01T00:00:30.000+08:00|           NaN|
+----------------------------+--------------+
```

用于查询的SQL语句：

```
select consistency(s1) from root.test.d1 where time <= 2020-01-01 00:00:30
```

输出序列：

```
+----------------------------+-----------------------------+
|                        Time|consistency(root.test.d1.s1)|
+----------------------------+-----------------------------+
|2020-01-01T00:00:02.000+08:00|           0.9333333333333333|
+----------------------------+-----------------------------+
```

3.2.2.2 指定窗口大小

在指定窗口大小的情况下，本函数会把输入数据划分为若干个窗口计算一致性。

输入序列:

```
+----------------------------+--------------+
|                        Time|root.test.d1.s1|
+----------------------------+--------------+
|2020-01-01T00:00:02.000+08:00|         100.0|
|2020-01-01T00:00:03.000+08:00|         101.0|
```

```
|2020-01-01T00:00:04.000+08:00|                102.0|
|2020-01-01T00:00:06.000+08:00|                104.0|
|2020-01-01T00:00:08.000+08:00|                126.0|
|2020-01-01T00:00:10.000+08:00|                108.0|
|2020-01-01T00:00:14.000+08:00|                112.0|
|2020-01-01T00:00:15.000+08:00|                113.0|
|2020-01-01T00:00:16.000+08:00|                114.0|
|2020-01-01T00:00:18.000+08:00|                116.0|
|2020-01-01T00:00:20.000+08:00|                118.0|
|2020-01-01T00:00:22.000+08:00|                120.0|
|2020-01-01T00:00:26.000+08:00|                124.0|
|2020-01-01T00:00:28.000+08:00|                126.0|
|2020-01-01T00:00:30.000+08:00|                 NaN|
|2020-01-01T00:00:32.000+08:00|                130.0|
|2020-01-01T00:00:34.000+08:00|                132.0|
|2020-01-01T00:00:36.000+08:00|                134.0|
|2020-01-01T00:00:38.000+08:00|                136.0|
|2020-01-01T00:00:40.000+08:00|                138.0|
|2020-01-01T00:00:42.000+08:00|                140.0|
|2020-01-01T00:00:44.000+08:00|                142.0|
|2020-01-01T00:00:46.000+08:00|                144.0|
|2020-01-01T00:00:48.000+08:00|                146.0|
|2020-01-01T00:00:50.000+08:00|                148.0|
|2020-01-01T00:00:52.000+08:00|                150.0|
|2020-01-01T00:00:54.000+08:00|                152.0|
|2020-01-01T00:00:56.000+08:00|                154.0|
|2020-01-01T00:00:58.000+08:00|                156.0|
|2020-01-01T00:01:00.000+08:00|                158.0|
+-----------------------------+--------------------+
```

用于查询的SQL语句：

```
select consistency(s1,"window"="15") from root.test.d1 where time <= 2020-01-01 00:01:00
```

输出序列：

```
+-----------------------------+-----------------------------------------------+
|                         Time|consistency(root.test.d1.s1, "window"="15")|
+-----------------------------+-----------------------------------------------+
|2020-01-01T00:00:02.000+08:00|                              0.9333333333333333|
|2020-01-01T00:00:32.000+08:00|                                            1.0|
+-----------------------------+-----------------------------------------------+
```

3.3 Timeliness

3.3.1 函数简介

本函数用于计算时间序列的时效性。将输入序列划分为若干个连续且不重叠的窗口，分别计算每一个窗口的时效性，并输出窗口第一个数据点的时间戳和窗口的时效性。

函数名： TIMELINESS

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- window：窗口大小，它是一个大于0的整数或者一个有单位的正数。前者代表每一个窗口包含的数据点数目，最后一个窗口的数据点数目可能会不足；后者代表窗口的时间跨度，目前支持五种单位，分别是'ms'（毫秒）、's'（秒）、'm'（分钟）、'h'（小时）和'd'（天）。缺省情况下，全部输入数据都属于同一个窗口。

输出序列： 输出单个序列，类型为DOUBLE，其中每一个数据点的值的范围都是[0,1]。

提示： 只有当窗口内的数据点数目超过10时，才会进行时效性计算。否则，该窗口将被忽略，不做任何输出。

3.3.2 使用示例

3.3.2.1 参数缺省

在参数缺省的情况下，本函数将会把全部输入数据都作为同一个窗口计算时效性。

输入序列：

```
+----------------------------+--------------+
|                        Time|root.test.d1.s1|
+----------------------------+--------------+
|2020-01-01T00:00:02.000+08:00|         100.0|
|2020-01-01T00:00:03.000+08:00|         101.0|
|2020-01-01T00:00:04.000+08:00|         102.0|
|2020-01-01T00:00:06.000+08:00|         104.0|
|2020-01-01T00:00:08.000+08:00|         126.0|
|2020-01-01T00:00:10.000+08:00|         108.0|
|2020-01-01T00:00:14.000+08:00|         112.0|
|2020-01-01T00:00:15.000+08:00|         113.0|
|2020-01-01T00:00:16.000+08:00|         114.0|
|2020-01-01T00:00:18.000+08:00|         116.0|
|2020-01-01T00:00:20.000+08:00|         118.0|
|2020-01-01T00:00:22.000+08:00|         120.0|
|2020-01-01T00:00:26.000+08:00|         124.0|
|2020-01-01T00:00:28.000+08:00|         126.0|
|2020-01-01T00:00:30.000+08:00|           NaN|
```

```
+-----------------------------+-------------+
```

用于查询的SQL语句：

```
select timeliness(s1) from root.test.d1 where time <= 2020-01-01 00:00:30
```

输出序列：

```
+-----------------------------+-------------------------------+
|                         Time|timeliness(root.test.d1.s1)|
+-----------------------------+-------------------------------+
|2020-01-01T00:00:02.000+08:00|             0.9333333333333333|
+-----------------------------+-------------------------------+
```

3.3.2.2 指定窗口大小

在指定窗口大小的情况下，本函数会把输入数据划分为若干个窗口计算时效性。

输入序列:

```
+-----------------------------+-------------+
|                         Time|root.test.d1.s1|
+-----------------------------+-------------+
|2020-01-01T00:00:02.000+08:00|        100.0|
|2020-01-01T00:00:03.000+08:00|        101.0|
|2020-01-01T00:00:04.000+08:00|        102.0|
|2020-01-01T00:00:06.000+08:00|        104.0|
|2020-01-01T00:00:08.000+08:00|        126.0|
|2020-01-01T00:00:10.000+08:00|        108.0|
|2020-01-01T00:00:14.000+08:00|        112.0|
|2020-01-01T00:00:15.000+08:00|        113.0|
|2020-01-01T00:00:16.000+08:00|        114.0|
|2020-01-01T00:00:18.000+08:00|        116.0|
|2020-01-01T00:00:20.000+08:00|        118.0|
|2020-01-01T00:00:22.000+08:00|        120.0|
|2020-01-01T00:00:26.000+08:00|        124.0|
|2020-01-01T00:00:28.000+08:00|        126.0|
|2020-01-01T00:00:30.000+08:00|         NaN|
|2020-01-01T00:00:32.000+08:00|        130.0|
|2020-01-01T00:00:34.000+08:00|        132.0|
|2020-01-01T00:00:36.000+08:00|        134.0|
|2020-01-01T00:00:38.000+08:00|        136.0|
|2020-01-01T00:00:40.000+08:00|        138.0|
|2020-01-01T00:00:42.000+08:00|        140.0|
|2020-01-01T00:00:44.000+08:00|        142.0|
|2020-01-01T00:00:46.000+08:00|        144.0|
|2020-01-01T00:00:48.000+08:00|        146.0|
|2020-01-01T00:00:50.000+08:00|        148.0|
|2020-01-01T00:00:52.000+08:00|        150.0|
```

```
|2020-01-01T00:00:54.000+08:00|                152.0|
|2020-01-01T00:00:56.000+08:00|                154.0|
|2020-01-01T00:00:58.000+08:00|                156.0|
|2020-01-01T00:01:00.000+08:00|                158.0|
+-----------------------------+---------------------+
```

用于查询的SQL语句：

```
select timeliness(s1,"window"="15") from root.test.d1 where time <= 2020-01-01 00:01:00
```

输出序列：

```
+-----------------------------+-----------------------------------------------+
|                         Time|timeliness(root.test.d1.s1, "window"="15")|
+-----------------------------+-----------------------------------------------+
|2020-01-01T00:00:02.000+08:00|                              0.9333333333333333|
|2020-01-01T00:00:32.000+08:00|                                            1.0|
+-----------------------------+-----------------------------------------------+
```

3.4 Validity

3.4.1 函数简介

本函数用于计算时间序列的有效性。将输入序列划分为若干个连续且不重叠的窗口，分别计算每一个窗口的有效性，并输出窗口第一个数据点的时间戳和窗口的有效性。

函数名： VALIDITY

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- window：窗口大小，它是一个大于0的整数或者一个有单位的正数。前者代表每一个窗口包含的数据点数目，最后一个窗口的数据点数目可能会不足；后者代表窗口的时间跨度，目前支持五种单位，分别是’ms’（毫秒）、’s’（秒）、’m’（分钟）、’h’（小时）和’d’（天）。缺省情况下，全部输入数据都属于同一个窗口。

输出序列： 输出单个序列，类型为DOUBLE，其中每一个数据点的值的范围都是[0,1]。

提示： 只有当窗口内的数据点数目超过10时，才会进行有效性计算。否则，该窗口将被忽略，不做任何输出。

3.4.2 使用示例

3.4.2.1 参数缺省

在参数缺省的情况下，本函数将会把全部输入数据都作为同一个窗口计算有效性。

输入序列：

```
+-----------------------------+------------+
|                         Time|root.test.d1.s1|
+-----------------------------+------------+
|2020-01-01T00:00:02.000+08:00|       100.0|
|2020-01-01T00:00:03.000+08:00|       101.0|
|2020-01-01T00:00:04.000+08:00|       102.0|
|2020-01-01T00:00:06.000+08:00|       104.0|
|2020-01-01T00:00:08.000+08:00|       126.0|
|2020-01-01T00:00:10.000+08:00|       108.0|
|2020-01-01T00:00:14.000+08:00|       112.0|
|2020-01-01T00:00:15.000+08:00|       113.0|
|2020-01-01T00:00:16.000+08:00|       114.0|
|2020-01-01T00:00:18.000+08:00|       116.0|
|2020-01-01T00:00:20.000+08:00|       118.0|
|2020-01-01T00:00:22.000+08:00|       120.0|
|2020-01-01T00:00:26.000+08:00|       124.0|
|2020-01-01T00:00:28.000+08:00|       126.0|
|2020-01-01T00:00:30.000+08:00|         NaN|
+-----------------------------+------------+
```

用于查询的SQL语句：

```
select validity(s1) from root.test.d1 where time <= 2020-01-01 00:00:30
```

输出序列：

```
+-----------------------------+-----------------------+
|                         Time|validity(root.test.d1.s1)|
+-----------------------------+-----------------------+
|2020-01-01T00:00:02.000+08:00|     0.8833333333333333|
+-----------------------------+-----------------------+
```

3.4.2.2 指定窗口大小

在指定窗口大小的情况下，本函数会把输入数据划分为若干个窗口计算有效性。

输入序列:

```
+-----------------------------+------------+
|                         Time|root.test.d1.s1|
+-----------------------------+------------+
|2020-01-01T00:00:02.000+08:00|       100.0|
|2020-01-01T00:00:03.000+08:00|       101.0|
|2020-01-01T00:00:04.000+08:00|       102.0|
|2020-01-01T00:00:06.000+08:00|       104.0|
|2020-01-01T00:00:08.000+08:00|       126.0|
|2020-01-01T00:00:10.000+08:00|       108.0|
|2020-01-01T00:00:14.000+08:00|       112.0|
```

```
|2020-01-01T00:00:15.000+08:00|                113.0|
|2020-01-01T00:00:16.000+08:00|                114.0|
|2020-01-01T00:00:18.000+08:00|                116.0|
|2020-01-01T00:00:20.000+08:00|                118.0|
|2020-01-01T00:00:22.000+08:00|                120.0|
|2020-01-01T00:00:26.000+08:00|                124.0|
|2020-01-01T00:00:28.000+08:00|                126.0|
|2020-01-01T00:00:30.000+08:00|                 NaN|
|2020-01-01T00:00:32.000+08:00|                130.0|
|2020-01-01T00:00:34.000+08:00|                132.0|
|2020-01-01T00:00:36.000+08:00|                134.0|
|2020-01-01T00:00:38.000+08:00|                136.0|
|2020-01-01T00:00:40.000+08:00|                138.0|
|2020-01-01T00:00:42.000+08:00|                140.0|
|2020-01-01T00:00:44.000+08:00|                142.0|
|2020-01-01T00:00:46.000+08:00|                144.0|
|2020-01-01T00:00:48.000+08:00|                146.0|
|2020-01-01T00:00:50.000+08:00|                148.0|
|2020-01-01T00:00:52.000+08:00|                150.0|
|2020-01-01T00:00:54.000+08:00|                152.0|
|2020-01-01T00:00:56.000+08:00|                154.0|
|2020-01-01T00:00:58.000+08:00|                156.0|
|2020-01-01T00:01:00.000+08:00|                158.0|
+-----------------------------+--------------------+
```

用于查询的SQL语句：

```
select validity(s1,"window"="15") from root.test.d1 where time <= 2020-01-01 00:01:00
```

输出序列：

```
+-----------------------------+-----------------------------------------------+
|                         Time|validity(root.test.d1.s1, "window"="15")|
+-----------------------------+-----------------------------------------------+
|2020-01-01T00:00:02.000+08:00|                              0.8833333333333333|
|2020-01-01T00:00:32.000+08:00|                                            1.0|
+-----------------------------+-----------------------------------------------+
```

第 4 章 数据修复

4.1 ValueFill

4.1.1 函数简介

函数名： ValueFill

输入序列： 单列时序数据，类型为INT32 / INT64 / FLOAT / DOUBLE

参数：

- `method`: {"mean", "previous", "linear", "likelihood", "AR", "MA", "SCREEN"},默认为"linear"。其中，“mean”指使用均值填补的方法；“previous"指使用前值填补方法；“linear"指使用线性插值填补方法；“likelihood”:为基于速度的正态分布的极大似然估计方法；“AR”指自回归的填补方法；“MA”指滑动平均的填补方法；"SCREEN"指约束填补方法；缺省情况下使用“linear”。

输出序列： 填补后的单维序列。

4.1.2 使用示例

4.1.2.1 使用linear方法进行填补

当`method`缺省或取值为’linear’时，本函数将使用线性插值方法进行填补。

输入序列：

```
+----------------------------+--------------+
|                        Time|root.test.d2.s1|
+----------------------------+--------------+
|2020-01-01T00:00:02.000+08:00|           NaN|
|2020-01-01T00:00:03.000+08:00|         101.0|
|2020-01-01T00:00:04.000+08:00|         102.0|
|2020-01-01T00:00:06.000+08:00|         104.0|
|2020-01-01T00:00:08.000+08:00|         126.0|
|2020-01-01T00:00:10.000+08:00|         108.0|
|2020-01-01T00:00:14.000+08:00|           NaN|
|2020-01-01T00:00:15.000+08:00|         113.0|
|2020-01-01T00:00:16.000+08:00|         114.0|
|2020-01-01T00:00:18.000+08:00|         116.0|
|2020-01-01T00:00:20.000+08:00|           NaN|
|2020-01-01T00:00:22.000+08:00|           NaN|
|2020-01-01T00:00:26.000+08:00|         124.0|
|2020-01-01T00:00:28.000+08:00|         126.0|
|2020-01-01T00:00:30.000+08:00|         128.0|
+----------------------------+--------------+
```

用于查询的SQL语句：

```
select valuefill(s1) from root.test.d2
```

输出序列：

```
+----------------------------+-----------------------+
|                        Time|valuefill(root.test.d2)|
+----------------------------+-----------------------+
|2020-01-01T00:00:02.000+08:00|                    NaN|
|2020-01-01T00:00:03.000+08:00|                  101.0|
|2020-01-01T00:00:04.000+08:00|                  102.0|
|2020-01-01T00:00:06.000+08:00|                  104.0|
|2020-01-01T00:00:08.000+08:00|                  126.0|
|2020-01-01T00:00:10.000+08:00|                  108.0|
|2020-01-01T00:00:14.000+08:00|                  108.0|
|2020-01-01T00:00:15.000+08:00|                  113.0|
|2020-01-01T00:00:16.000+08:00|                  114.0|
|2020-01-01T00:00:18.000+08:00|                  116.0|
|2020-01-01T00:00:20.000+08:00|                  118.7|
|2020-01-01T00:00:22.000+08:00|                  121.3|
|2020-01-01T00:00:26.000+08:00|                  124.0|
|2020-01-01T00:00:28.000+08:00|                  126.0|
|2020-01-01T00:00:30.000+08:00|                  128.0|
+----------------------------+-----------------------+
```

4.1.2.2 使用previous方法进行填补

当 method 取值为'previous'时，本函数将使前值填补方法进行数值填补。

输入序列同上，用于查询的SQL语句如下：

```
select valuefill(s1,"method"="previous") from root.test.d2
```

输出序列：

```
+----------------------------+-----------------------------------------------+
|                        Time|valuefill(root.test.d2,"method"="previous")|
+----------------------------+-----------------------------------------------+
|2020-01-01T00:00:02.000+08:00|                                            NaN|
|2020-01-01T00:00:03.000+08:00|                                          101.0|
|2020-01-01T00:00:04.000+08:00|                                          102.0|
|2020-01-01T00:00:06.000+08:00|                                          104.0|
|2020-01-01T00:00:08.000+08:00|                                          126.0|
|2020-01-01T00:00:10.000+08:00|                                          108.0|
|2020-01-01T00:00:14.000+08:00|                                          110.5|
|2020-01-01T00:00:15.000+08:00|                                          113.0|
|2020-01-01T00:00:16.000+08:00|                                          114.0|
|2020-01-01T00:00:18.000+08:00|                                          116.0|
|2020-01-01T00:00:20.000+08:00|                                          116.0|
```

```
|2020-01-01T00:00:22.000+08:00|                                116.0|
|2020-01-01T00:00:26.000+08:00|                                124.0|
|2020-01-01T00:00:28.000+08:00|                                126.0|
|2020-01-01T00:00:30.000+08:00|                                128.0|
+-----------------------------+-------------------------------------+
```

4.2 TimestampRepair

4.2.1 函数简介

本函数用于时间戳修复。根据给定的标准时间间隔，采用最小化修复代价的方法，通过对数据时间戳的微调，将原本时间戳间隔不稳定的数据修复为严格等间隔的数据。在未给定标准时间间隔的情况下，本函数将使用时间间隔的中位数(median)、众数(mode)或聚类中心(cluster)来推算标准时间间隔。

函数名： TIMESTAMPREPAIR

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- `interval`: 标准时间间隔（单位是毫秒），是一个正整数。在缺省情况下，将根据指定的方法推算。
- `method`：推算标准时间间隔的方法，取值为'median'、'mode'或'cluster'，仅在`interval`缺省时有效。在缺省情况下，将使用中位数方法进行推算。

输出序列： 输出单个序列，类型与输入序列相同。该序列是修复后的输入序列。

4.2.2 使用示例

4.2.2.1 指定标准时间间隔

在给定`interval`参数的情况下，本函数将按照指定的标准时间间隔进行修复。

输入序列：

```
+-----------------------------+-------------+
|                         Time|root.test.d2.s1|
+-----------------------------+-------------+
|2021-07-01T12:00:00.000+08:00|          1.0|
|2021-07-01T12:00:10.000+08:00|          2.0|
|2021-07-01T12:00:19.000+08:00|          3.0|
|2021-07-01T12:00:30.000+08:00|          4.0|
|2021-07-01T12:00:40.000+08:00|          5.0|
|2021-07-01T12:00:50.000+08:00|          6.0|
|2021-07-01T12:01:01.000+08:00|          7.0|
|2021-07-01T12:01:11.000+08:00|          8.0|
|2021-07-01T12:01:21.000+08:00|          9.0|
|2021-07-01T12:01:31.000+08:00|         10.0|
+-----------------------------+-------------+
```

用于查询的SQL语句：

```
select timestamprepair(s1,'interval'='10000') from root.test.d2
```

输出序列：

```
+-----------------------------+-----------------------------------------------------+
|                         Time|timestamprepair(root.test.d2.s1, "interval"="10000")|
+-----------------------------+-----------------------------------------------------+
|2021-07-01T12:00:00.000+08:00|                                                  1.0|
|2021-07-01T12:00:10.000+08:00|                                                  2.0|
|2021-07-01T12:00:20.000+08:00|                                                  3.0|
|2021-07-01T12:00:30.000+08:00|                                                  4.0|
|2021-07-01T12:00:40.000+08:00|                                                  5.0|
|2021-07-01T12:00:50.000+08:00|                                                  6.0|
|2021-07-01T12:01:00.000+08:00|                                                  7.0|
|2021-07-01T12:01:10.000+08:00|                                                  8.0|
|2021-07-01T12:01:20.000+08:00|                                                  9.0|
|2021-07-01T12:01:30.000+08:00|                                                 10.0|
+-----------------------------+-----------------------------------------------------+
```

4.2.2.2 自动推算标准时间间隔

如果 interval 参数没有给定，本函数将按照推算的标准时间间隔进行修复。

输入序列同上，用于查询的SQL语句如下：

```
select timestamprepair(s1) from root.test.d2
```

输出序列：

```
+-----------------------------+-------------------------------+
|                         Time|timestamprepair(root.test.d2.s1)|
+-----------------------------+-------------------------------+
|2021-07-01T12:00:00.000+08:00|                            1.0|
|2021-07-01T12:00:10.000+08:00|                            2.0|
|2021-07-01T12:00:20.000+08:00|                            3.0|
|2021-07-01T12:00:30.000+08:00|                            4.0|
|2021-07-01T12:00:40.000+08:00|                            5.0|
|2021-07-01T12:00:50.000+08:00|                            6.0|
|2021-07-01T12:01:00.000+08:00|                            7.0|
|2021-07-01T12:01:10.000+08:00|                            8.0|
|2021-07-01T12:01:20.000+08:00|                            9.0|
|2021-07-01T12:01:30.000+08:00|                           10.0|
+-----------------------------+-------------------------------+
```

4.3 ValueRepair

4.3.1 函数简介

本函数用于对时间序列的数值进行修复。目前，本函数支持两种修复方法：**Screen**是一种基于速度阈值的方法，在最小改动的前提下使得所有的速度符合阈值要求；**LsGreedy**是一种基于速度变化似然的方法，将速度变化建模为高斯分布，并采用贪心算法极大化似然函数。

函数名： VALUEREPAIR

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

参数：

- `method`：修复时采用的方法，取值为'Screen'或'LsGreedy'。在缺省情况下，使用Screen方法进行修复。
- `minSpeed`：该参数仅在使用Screen方法时有效。当速度小于该值时会被视作数值异常点加以修复。在缺省情况下为中位数减去三倍绝对中位差。
- `maxSpeed`：该参数仅在使用Screen方法时有效。当速度大于该值时会被视作数值异常点加以修复。在缺省情况下为中位数加上三倍绝对中位差。
- `center`：该参数仅在使用LsGreedy方法时有效。对速度变化分布建立的高斯模型的中心。在缺省情况下为0。
- `sigma`：该参数仅在使用LsGreedy方法时有效。对速度变化分布建立的高斯模型的标准差。在缺省情况下为绝对中位差。

输出序列： 输出单个序列，类型与输入序列相同。该序列是修复后的输入序列。

提示： 输入序列中的`NaN`在修复之前会先进行线性插值填补。

4.3.2 使用示例

4.3.2.1 使用Screen方法进行修复

当`method`缺省或取值为'Screen'时，本函数将使用Screen方法进行数值修复。

输入序列：

```
+-----------------------------+--------------+
|                         Time|root.test.d2.s1|
+-----------------------------+--------------+
|2020-01-01T00:00:02.000+08:00|         100.0|
|2020-01-01T00:00:03.000+08:00|         101.0|
|2020-01-01T00:00:04.000+08:00|         102.0|
|2020-01-01T00:00:06.000+08:00|         104.0|
|2020-01-01T00:00:08.000+08:00|         126.0|
|2020-01-01T00:00:10.000+08:00|         108.0|
|2020-01-01T00:00:14.000+08:00|         112.0|
|2020-01-01T00:00:15.000+08:00|         113.0|
|2020-01-01T00:00:16.000+08:00|         114.0|
```

```
|2020-01-01T00:00:18.000+08:00|             116.0|
|2020-01-01T00:00:20.000+08:00|             118.0|
|2020-01-01T00:00:22.000+08:00|             100.0|
|2020-01-01T00:00:26.000+08:00|             124.0|
|2020-01-01T00:00:28.000+08:00|             126.0|
|2020-01-01T00:00:30.000+08:00|             NaN|
+-----------------------------+-------------------+
```

用于查询的SQL语句：

```
select valuerepair(s1) from root.test.d2
```

输出序列：

```
+-----------------------------+-----------------------------+
|                         Time|valuerepair(root.test.d2.s1)|
+-----------------------------+-----------------------------+
|2020-01-01T00:00:02.000+08:00|                        100.0|
|2020-01-01T00:00:03.000+08:00|                        101.0|
|2020-01-01T00:00:04.000+08:00|                        102.0|
|2020-01-01T00:00:06.000+08:00|                        104.0|
|2020-01-01T00:00:08.000+08:00|                        106.0|
|2020-01-01T00:00:10.000+08:00|                        108.0|
|2020-01-01T00:00:14.000+08:00|                        112.0|
|2020-01-01T00:00:15.000+08:00|                        113.0|
|2020-01-01T00:00:16.000+08:00|                        114.0|
|2020-01-01T00:00:18.000+08:00|                        116.0|
|2020-01-01T00:00:20.000+08:00|                        118.0|
|2020-01-01T00:00:22.000+08:00|                        120.0|
|2020-01-01T00:00:26.000+08:00|                        124.0|
|2020-01-01T00:00:28.000+08:00|                        126.0|
|2020-01-01T00:00:30.000+08:00|                        128.0|
+-----------------------------+-----------------------------+
```

4.3.2.2 使用LsGreedy方法进行修复

当 method 取值为'LsGreedy'时，本函数将使用LsGreedy方法进行数值修复。

输入序列同上，用于查询的SQL语句如下：

```
select valuerepair(s1,'method'='LsGreedy') from root.test.d2
```

输出序列：

```
+-----------------------------+-----------------------------------------------------+
|                         Time|valuerepair(root.test.d2.s1, "method"="LsGreedy")|
+-----------------------------+-----------------------------------------------------+
|2020-01-01T00:00:02.000+08:00|                                                100.0|
|2020-01-01T00:00:03.000+08:00|                                                101.0|
|2020-01-01T00:00:04.000+08:00|                                                102.0|
```

```
|2020-01-01T00:00:06.000+08:00|                                104.0|
|2020-01-01T00:00:08.000+08:00|                                106.0|
|2020-01-01T00:00:10.000+08:00|                                108.0|
|2020-01-01T00:00:14.000+08:00|                                112.0|
|2020-01-01T00:00:15.000+08:00|                                113.0|
|2020-01-01T00:00:16.000+08:00|                                114.0|
|2020-01-01T00:00:18.000+08:00|                                116.0|
|2020-01-01T00:00:20.000+08:00|                                118.0|
|2020-01-01T00:00:22.000+08:00|                                120.0|
|2020-01-01T00:00:26.000+08:00|                                124.0|
|2020-01-01T00:00:28.000+08:00|                                126.0|
|2020-01-01T00:00:30.000+08:00|                                128.0|
+-----------------------------+-----------------------------------+
```

第 5 章 数据匹配

5.1 Cov

5.1.1 函数简介

本函数用于计算两列数值型数据的总体协方差。

函数名： COV

输入序列： 仅支持两个输入序列，类型均为 INT32 / INT64 / FLOAT / DOUBLE。

输出序列： 输出单个序列，类型为DOUBLE。序列仅包含一个时间戳为0、值为总体协方差的数据点。

提示：

- 如果某行数据中包含空值、缺失值或 NaN ，该行数据将会被忽略；
- 如果数据中所有的行都被忽略，函数将会输出 NaN 。

5.1.2 使用示例

输入序列：

```
+-----------------------------+---------------+---------------+
|                         Time|root.test.d2.s1|root.test.d2.s2|
+-----------------------------+---------------+---------------+
|2020-01-01T00:00:02.000+08:00|          100.0|          101.0|
|2020-01-01T00:00:03.000+08:00|          101.0|           null|
|2020-01-01T00:00:04.000+08:00|          102.0|          101.0|
|2020-01-01T00:00:06.000+08:00|          104.0|          102.0|
|2020-01-01T00:00:08.000+08:00|          126.0|          102.0|
|2020-01-01T00:00:10.000+08:00|          108.0|          103.0|
|2020-01-01T00:00:12.000+08:00|           null|          103.0|
|2020-01-01T00:00:14.000+08:00|          112.0|          104.0|
|2020-01-01T00:00:15.000+08:00|          113.0|           null|
|2020-01-01T00:00:16.000+08:00|          114.0|          104.0|
|2020-01-01T00:00:18.000+08:00|          116.0|          105.0|
|2020-01-01T00:00:20.000+08:00|          118.0|          105.0|
|2020-01-01T00:00:22.000+08:00|          100.0|          106.0|
|2020-01-01T00:00:26.000+08:00|          124.0|          108.0|
|2020-01-01T00:00:28.000+08:00|          126.0|          108.0|
|2020-01-01T00:00:30.000+08:00|            NaN|          108.0|
+-----------------------------+---------------+---------------+
```

用于查询的SQL语句：

```
select cov(s1,s2) from root.test.d2
```

输出序列：

```
+-----------------------------+--------------------------------------+
|                         Time|cov(root.test.d2.s1, root.test.d2.s2)|
+-----------------------------+--------------------------------------+
|1970-01-01T08:00:00.000+08:00|                    12.291666666666666|
+-----------------------------+--------------------------------------+
```

5.2 CrossCorrelation

5.2.1 函数简介

本函数用于计算两条时间序列的互相关函数值，对离散序列而言，互相关函数可以表示为

$$CR(n) = \frac{1}{N}\sum_{m=1}^{N} S_1[m]S_2[m+n]$$

常用于表征两条序列在不同对齐条件下的相似度。

函数名： CROSSCORRELATION

输入序列： 仅支持两个输入序列，类型均为 INT32 / INT64 / FLOAT / DOUBLE。

输出序列： 输出单个序列，类型为DOUBLE。序列中共包含 $2N-1$ 个数据点，其中正中心的值为两条序列按照预先对齐的结果计算的互相关系数（即等于以上公式的 $CR(0)$ ），前半部分的值表示将后一条输入序列向前平移时计算的互相关系数，直至两条序列没有重合的数据点（不包含完全分离时的结果 $CR(-N) = 0.0$ ），后半部分类似。用公式可表示为（所有序列的索引从1开始计数）：

$$OS[i] = CR(-N+i) = \frac{1}{N}\sum_{m=1}^{i} S_1[m]S_2[N-i+m],\ if\ i <= N$$

$$OS[i] = CR(i-N) = \frac{1}{N}\sum_{m=1}^{2N-i} S_1[i-N+m]S_2[m],\ if\ i > N$$

提示：

- 两条序列中的 null 和 NaN 值会被忽略，在计算中表现为0。

5.2.2 使用示例

输入序列：

```
+-----------------------------+---------------+---------------+
|                         Time|root.test.d1.s1|root.test.d1.s2|
+-----------------------------+---------------+---------------+
|2020-01-01T00:00:01.000+08:00|           null|              6|
|2020-01-01T00:00:02.000+08:00|              2|              7|
|2020-01-01T00:00:03.000+08:00|              3|            NaN|
```

```
|2020-01-01T00:00:04.000+08:00|                4|                9|
|2020-01-01T00:00:05.000+08:00|                5|               10|
+-----------------------------+-----------------+-----------------+
```

用于查询的SQL语句：

```
select crosscorrelation(s1, s2) from root.test.d1 where time <= 2020-01-01 00:00:05
```

输出序列：

```
+-----------------------------+-----------------------------------------------------+
|                         Time|crosscorrelation(root.test.d1.s1, root.test.d1.s2)|
+-----------------------------+-----------------------------------------------------+
|1970-01-01T08:00:00.001+08:00|                                                  0.0|
|1970-01-01T08:00:00.002+08:00|                                                  4.0|
|1970-01-01T08:00:00.003+08:00|                                                  9.6|
|1970-01-01T08:00:00.004+08:00|                                                 13.4|
|1970-01-01T08:00:00.005+08:00|                                                 20.0|
|1970-01-01T08:00:00.006+08:00|                                                 15.6|
|1970-01-01T08:00:00.007+08:00|                                                  9.2|
|1970-01-01T08:00:00.008+08:00|                                                 11.8|
|1970-01-01T08:00:00.009+08:00|                                                  6.0|
+-----------------------------+-----------------------------------------------------+
```

5.2.2.1 Zeppelin示例

链接: <http://101.6.15.213:18181/#/notebook/2GETVW6AT>

5.3 Dtw

5.3.1 函数简介

本函数用于计算两列数值型数据的DTW距离。

函数名： DTW

输入序列： 仅支持两个输入序列，类型均为 INT32 / INT64 / FLOAT / DOUBLE。

输出序列： 输出单个序列，类型为DOUBLE。序列仅包含一个时间戳为0、值为两个时间序列的DTW距离值。

提示：

- 如果某行数据中包含空值、缺失值或 NaN ，该行数据将会被忽略；
- 如果数据中所有的行都被忽略，函数将会输出0。

5.3.2 使用示例

输入序列：

```
+-----------------------------+-------------+-------------+
|                         Time|root.test.d2.s1|root.test.d2.s2|
+-----------------------------+-------------+-------------+
|1970-01-01T08:00:00.001+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.002+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.003+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.004+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.005+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.006+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.007+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.008+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.009+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.010+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.011+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.012+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.013+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.014+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.015+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.016+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.017+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.018+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.019+08:00|          1.0|          2.0|
|1970-01-01T08:00:00.020+08:00|          1.0|          2.0|
+-----------------------------+-------------+-------------+
```

用于查询的SQL语句：

```
select dtw(s1,s2) from root.test.d2
```

输出序列：

```
+-----------------------------+-----------------------------------------+
|                         Time|dtw(root.test.d2.s1, root.test.d2.s2)|
+-----------------------------+-----------------------------------------+
|1970-01-01T08:00:00.000+08:00|                                     20.0|
+-----------------------------+-----------------------------------------+
```

5.4 PatternSymmetric

5.4.1 函数简介

本函数用于寻找序列中所有对称度小于阈值的对称子序列。对称度通过DTW计算，值越小代表序列对称性越高。

函数名： PATTERNSYMMETRIC

输入序列： 仅支持一个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

参数：

- window：对称子序列的长度，是一个正整数，默认值为10。
- threshold：对称度阈值，是一个非负数，只有对称度小于等于该值的对称子序列才会被输出。在缺省情况下，所有的子序列都会被输出。

输出序列： 输出单个序列，类型为DOUBLE。序列中的每一个数据点对应于一个对称子序列，时间戳为子序列的起始时刻，值为对称度。

5.4.2 使用示例

输入序列：

```
+-----------------------------+------------+
|                         Time|root.test.d1.s4|
+-----------------------------+------------+
|2021-01-01T12:00:00.000+08:00|         1.0|
|2021-01-01T12:00:01.000+08:00|         2.0|
|2021-01-01T12:00:02.000+08:00|         3.0|
|2021-01-01T12:00:03.000+08:00|         2.0|
|2021-01-01T12:00:04.000+08:00|         1.0|
|2021-01-01T12:00:05.000+08:00|         1.0|
|2021-01-01T12:00:06.000+08:00|         1.0|
|2021-01-01T12:00:07.000+08:00|         1.0|
|2021-01-01T12:00:08.000+08:00|         2.0|
|2021-01-01T12:00:09.000+08:00|         3.0|
|2021-01-01T12:00:10.000+08:00|         2.0|
|2021-01-01T12:00:11.000+08:00|         1.0|
+-----------------------------+------------+
```

用于查询的SQL语句：

```
select patternsymmetric(s4, 'window'='5', 'threshold'='0') from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------------------------------+
|                         Time|patternsymmetric(root.test.d1.s4, "window"="5", "threshold"="0")|
+-----------------------------+-----------------------------------------------------------------------+
|2021-01-01T12:00:00.000+08:00|                                                                    0.0|
|2021-01-01T12:00:07.000+08:00|                                                                    0.0|
+-----------------------------+-----------------------------------------------------------------------+
```

5.5 Pearson

5.5.1 函数简介

本函数用于计算两列数值型数据的皮尔森相关系数。

函数名： PEARSON

输入序列： 仅支持两个输入序列，类型均为 INT32 / INT64 / FLOAT / DOUBLE。

输出序列： 输出单个序列，类型为DOUBLE。序列仅包含一个时间戳为0、值为皮尔森相关系数的数据点。

提示：

- 如果某行数据中包含空值、缺失值或 NaN ，该行数据将会被忽略；
- 如果数据中所有的行都被忽略，函数将会输出 NaN 。

5.5.2 使用示例

输入序列：

```
+-----------------------------+---------------+---------------+
|                         Time|root.test.d2.s1|root.test.d2.s2|
+-----------------------------+---------------+---------------+
|2020-01-01T00:00:02.000+08:00|          100.0|          101.0|
|2020-01-01T00:00:03.000+08:00|          101.0|           null|
|2020-01-01T00:00:04.000+08:00|          102.0|          101.0|
|2020-01-01T00:00:06.000+08:00|          104.0|          102.0|
|2020-01-01T00:00:08.000+08:00|          126.0|          102.0|
|2020-01-01T00:00:10.000+08:00|          108.0|          103.0|
|2020-01-01T00:00:12.000+08:00|           null|          103.0|
|2020-01-01T00:00:14.000+08:00|          112.0|          104.0|
|2020-01-01T00:00:15.000+08:00|          113.0|           null|
|2020-01-01T00:00:16.000+08:00|          114.0|          104.0|
|2020-01-01T00:00:18.000+08:00|          116.0|          105.0|
|2020-01-01T00:00:20.000+08:00|          118.0|          105.0|
|2020-01-01T00:00:22.000+08:00|          100.0|          106.0|
|2020-01-01T00:00:26.000+08:00|          124.0|          108.0|
|2020-01-01T00:00:28.000+08:00|          126.0|          108.0|
|2020-01-01T00:00:30.000+08:00|            NaN|          108.0|
+-----------------------------+---------------+---------------+
```

用于查询的SQL语句：

```
select pearson(s1,s2) from root.test.d2
```

输出序列：

```
+-----------------------------+-----------------------------------------------+
|                         Time|pearson(root.test.d2.s1, root.test.d2.s2)|
+-----------------------------+-----------------------------------------------+
|1970-01-01T08:00:00.000+08:00|                              0.5630881927754872|
+-----------------------------+-----------------------------------------------+
```

5.6 SelfCorrelation

5.6.1 函数简介

本函数用于计算时间序列的自相关函数值，即序列与自身之间的互相关函数，详情参见 CrossCorrelation 函数文档。

函数名： SELFCORRELATION

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

输出序列： 输出单个序列，类型为DOUBLE。序列中共包含 $2N-1$ 个数据点，每个值的具体含义参见 CrossCorrelation 函数文档。

提示：

- 序列中的 NaN 值会被忽略，在计算中表现为0。

5.6.2 使用示例

输入序列：

```
+-----------------------------+---------------+
|                         Time|root.test.d1.s1|
+-----------------------------+---------------+
|2020-01-01T00:00:01.000+08:00|              1|
|2020-01-01T00:00:02.000+08:00|            NaN|
|2020-01-01T00:00:03.000+08:00|              3|
|2020-01-01T00:00:04.000+08:00|            NaN|
|2020-01-01T00:00:05.000+08:00|              5|
+-----------------------------+---------------+
```

用于查询的SQL语句：

```
select selfcorrelation(s1) from root.test.d1 where time <= 2020-01-01 00:00:05
```

输出序列：

```
+-----------------------------+-------------------------------------+
|                         Time|selfcorrelation(root.test.d1.s1)|
+-----------------------------+-------------------------------------+
|1970-01-01T08:00:00.001+08:00|                                  1.0|
|1970-01-01T08:00:00.002+08:00|                                  0.0|
|1970-01-01T08:00:00.003+08:00|                                  3.6|
|1970-01-01T08:00:00.004+08:00|                                  0.0|
|1970-01-01T08:00:00.005+08:00|                                  7.0|
|1970-01-01T08:00:00.006+08:00|                                  0.0|
|1970-01-01T08:00:00.007+08:00|                                  3.6|
|1970-01-01T08:00:00.008+08:00|                                  0.0|
|1970-01-01T08:00:00.009+08:00|                                  1.0|
+-----------------------------+-------------------------------------+
```

5.6.2.1 Zeppelin示例

链接: <http://101.6.15.213:18181/#/notebook/2GC91M5DY>

5.7 SeriesAlign(TODO)

5.8 SeriesSimilarity(TODO)

5.9 ValueAlign(TODO)

第 6 章 异常检测

6.1 ADWIN

6.1.1 函数简介

本函数用于查找时间序列可能的概念漂移。本方法根据提供的δ，使用ADWIN方法判断是否存在异常大于置信度δ的位置。返回所有认为发生概念漂移的时间戳。具体方法请参见

```
Learning from Time-Changing Data with Adaptive Windowing, A Bifetet al., 2005
```

函数名： ADWIN

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- `delta`:判断发生概念漂移的阈值。详见论文中定义的δ，默认为0.01.
- `windowsize`:进行检测的窗口大小，该值应大于20。

输出序列： 输出单个序列，类型为INT32，异常输出1，非异常输出0。

6.1.2 使用示例

6.1.2.1 提供参数

输入序列：

```
+-----------------------------+-----------+
|                         Time|root.test.s1|
+-----------------------------+-----------+
|1970-01-01T08:00:00.000+08:00|        5.0|
|1970-01-01T08:00:00.100+08:00|        5.0|
|1970-01-01T08:00:00.200+08:00|        5.0|
|1970-01-01T08:00:00.300+08:00|        5.0|
|1970-01-01T08:00:00.400+08:00|        5.0|
|1970-01-01T08:00:00.500+08:00|        5.0|
|1970-01-01T08:00:00.600+08:00|        5.0|
|1970-01-01T08:00:00.700+08:00|        5.0|
|1970-01-01T08:00:00.800+08:00|        5.0|
|1970-01-01T08:00:00.900+08:00|        5.0|
|1970-01-01T08:00:01.000+08:00|        5.0|
|1970-01-01T08:00:01.100+08:00|        5.0|
|1970-01-01T08:00:01.200+08:00|        5.0|
|1970-01-01T08:00:01.300+08:00|        5.0|
|1970-01-01T08:00:01.400+08:00|        5.0|
|1970-01-01T08:00:01.500+08:00|        5.0|
|1970-01-01T08:00:01.600+08:00|        5.0|
```

```
|1970-01-01T08:00:01.700+08:00|              5.0|
|1970-01-01T08:00:01.800+08:00|              5.0|
|1970-01-01T08:00:01.900+08:00|              5.0|
|1970-01-01T08:00:02.000+08:00|             10.0|
|1970-01-01T08:00:02.100+08:00|             10.0|
|1970-01-01T08:00:02.200+08:00|             10.0|
|1970-01-01T08:00:02.300+08:00|             10.0|
|1970-01-01T08:00:02.400+08:00|             10.0|
|1970-01-01T08:00:02.500+08:00|             10.0|
|1970-01-01T08:00:02.600+08:00|             10.0|
|1970-01-01T08:00:02.700+08:00|             10.0|
|1970-01-01T08:00:02.800+08:00|             10.0|
|1970-01-01T08:00:02.900+08:00|             10.0|
|1970-01-01T08:00:03.000+08:00|             10.0|
|1970-01-01T08:00:03.100+08:00|             10.0|
|1970-01-01T08:00:03.200+08:00|             10.0|
|1970-01-01T08:00:03.300+08:00|             10.0|
|1970-01-01T08:00:03.400+08:00|             10.0|
|1970-01-01T08:00:03.500+08:00|             10.0|
|1970-01-01T08:00:03.600+08:00|             10.0|
|1970-01-01T08:00:03.700+08:00|             10.0|
|1970-01-01T08:00:03.800+08:00|             10.0|
|1970-01-01T08:00:03.900+08:00|             10.0|
+-----------------------------+------------------+
```

用于查询的SQL语句：

```
select adwin(s1,"windowsize"="30","delta"="0.01") from root.test
```

输出序列：

```
+-----------------------------+-----------------------------------------------------------+
|                         Time|adwin(root.test.s1, "windowsize"="30", "delta"="0.01")|
+-----------------------------+-----------------------------------------------------------+
|1970-01-01T08:00:02.100+08:00|                                                          1|
+-----------------------------+-----------------------------------------------------------+
```

6.2 IQR

6.2.1 函数简介

本函数用于检验超出上下四分位数1.5倍IQR的数据分布异常。

函数名： IQR

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE。

参数：

- method：若设置为"batch"，则将数据全部读入后检测；若设置为"stream"，则需用户提供上下四分位数进行流式检测。默认为"batch"。
- q1：使用流式计算时的下四分位数。
- q3：使用流式计算时的上四分位数。

输出序列：输出单个序列，类型为DOUBLE。

说明： $IQR = Q_3 - Q_1$

6.2.2 使用示例

6.2.2.1 全数据计算

输入序列：

```
+-----------------------------+------------+
|                         Time|root.test.s1|
+-----------------------------+------------+
|1970-01-01T08:00:00.100+08:00|         0.0|
|1970-01-01T08:00:00.200+08:00|         0.0|
|1970-01-01T08:00:00.300+08:00|         1.0|
|1970-01-01T08:00:00.400+08:00|        -1.0|
|1970-01-01T08:00:00.500+08:00|         0.0|
|1970-01-01T08:00:00.600+08:00|         0.0|
|1970-01-01T08:00:00.700+08:00|        -2.0|
|1970-01-01T08:00:00.800+08:00|         2.0|
|1970-01-01T08:00:00.900+08:00|         0.0|
|1970-01-01T08:00:01.000+08:00|         0.0|
|1970-01-01T08:00:01.100+08:00|         1.0|
|1970-01-01T08:00:01.200+08:00|        -1.0|
|1970-01-01T08:00:01.300+08:00|        -1.0|
|1970-01-01T08:00:01.400+08:00|         1.0|
|1970-01-01T08:00:01.500+08:00|         0.0|
|1970-01-01T08:00:01.600+08:00|         0.0|
|1970-01-01T08:00:01.700+08:00|        10.0|
|1970-01-01T08:00:01.800+08:00|         2.0|
|1970-01-01T08:00:01.900+08:00|        -2.0|
|1970-01-01T08:00:02.000+08:00|         0.0|
+-----------------------------+------------+
```

用于查询的SQL语句：

```
select iqr(s1) from root.test
```

输出序列：

```
+-----------------------------+-----------------+
|                         Time|iqr(root.test.s1)|
+-----------------------------+-----------------+
|1970-01-01T08:00:01.700+08:00|             10.0|
```

```
+-----------------------------+-------------+
```

6.3 KSigma

6.3.1 函数简介

本函数利用动态K-Sigma算法进行异常检测。在一个窗口内，与平均值的差距超过k倍标准差的数据将被视作异常并输出。

函数名： KSIGMA

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- `k`：在动态K-Sigma算法中，分布异常的标准差倍数阈值，默认值为3。
- `window`：动态K-Sigma算法的滑动窗口大小，默认值为10000。

输出序列： 输出单个序列，类型与输入序列相同。

提示： k应大于0，否则将不做输出。

6.3.2 使用示例

6.3.2.1 指定k

输入序列：

```
+-----------------------------+-------------+
|                         Time|root.test.d1.s1|
+-----------------------------+-------------+
|2020-01-01T00:00:02.000+08:00|          0.0|
|2020-01-01T00:00:03.000+08:00|         50.0|
|2020-01-01T00:00:04.000+08:00|        100.0|
|2020-01-01T00:00:06.000+08:00|        150.0|
|2020-01-01T00:00:08.000+08:00|        200.0|
|2020-01-01T00:00:10.000+08:00|        200.0|
|2020-01-01T00:00:14.000+08:00|        200.0|
|2020-01-01T00:00:15.000+08:00|        200.0|
|2020-01-01T00:00:16.000+08:00|        200.0|
|2020-01-01T00:00:18.000+08:00|        200.0|
|2020-01-01T00:00:20.000+08:00|        150.0|
|2020-01-01T00:00:22.000+08:00|        100.0|
|2020-01-01T00:00:26.000+08:00|         50.0|
|2020-01-01T00:00:28.000+08:00|          0.0|
|2020-01-01T00:00:30.000+08:00|          NaN|
+-----------------------------+-------------+
```

用于查询的SQL语句：

```
select ksigma(s1,"k"="1.0") from root.test.d1 where time <= 2020-01-01 00:00:30
```

输出序列：

```
+-----------------------------+-----------------------------------+
|Time                         |ksigma(root.test.d1.s1,"k"="3.0")|
+-----------------------------+-----------------------------------+
|2020-01-01T00:00:02.000+08:00|                                0.0|
|2020-01-01T00:00:03.000+08:00|                               50.0|
|2020-01-01T00:00:26.000+08:00|                               50.0|
|2020-01-01T00:00:28.000+08:00|                                0.0|
+-----------------------------+-----------------------------------+
```

6.4 LOF

6.4.1 函数简介

本函数使用局部离群点检测方法用于查找序列的密度异常。将根据提供的第k距离数及局部离群点因子(lof)阈值，判断输入数据是否为离群点，即异常，并输出各点的LOF值。

函数名： LOF

输入序列： 多个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- `method`:使用的检测方法。默认为default，以高维数据计算。设置为series，将一维时间序列转换为高维数据计算。
- `k`:使用第k距离计算局部离群点因子.默认为3。
- `window`:每次读取数据的窗口长度。默认为10000.
- `windowsize`:使用series方法时，转化高维数据的维数，即单个窗口的大小。默认为5。

输出序列： 输出单时间序列，类型为DOUBLE。

提示： 不完整的数据行会被忽略，不参与计算，也不标记为离群点。

6.4.2 使用示例

6.4.2.1 默认参数

输入序列：

```
+-----------------------------+--------------+--------------+
|                         Time|root.test.d1.s1|root.test.d1.s2|
+-----------------------------+--------------+--------------+
|1970-01-01T08:00:00.100+08:00|           0.0|           0.0|
|1970-01-01T08:00:00.200+08:00|           0.0|           1.0|
|1970-01-01T08:00:00.300+08:00|           1.0|           1.0|
|1970-01-01T08:00:00.400+08:00|           1.0|           0.0|
|1970-01-01T08:00:00.500+08:00|           0.0|          -1.0|
```

```
|1970-01-01T08:00:00.600+08:00|          -1.0|          -1.0|
|1970-01-01T08:00:00.700+08:00|          -1.0|           0.0|
|1970-01-01T08:00:00.800+08:00|           2.0|           2.0|
|1970-01-01T08:00:00.900+08:00|           0.0|           null|
+-----------------------------+---------------+---------------+
```

查询语句：

```
select lof(s1,s2) from root.test.d1 where time<1000
```

输出序列：

```
+-----------------------------+-----------------------------------------+
|                         Time|lof(root.test.d1.s1, root.test.d1.s2)|
+-----------------------------+-----------------------------------------+
|1970-01-01T08:00:00.100+08:00|                3.8274824267668244|
|1970-01-01T08:00:00.200+08:00|                3.0117631741126156|
|1970-01-01T08:00:00.300+08:00|                2.838155437762879|
|1970-01-01T08:00:00.400+08:00|                3.0117631741126156|
|1970-01-01T08:00:00.500+08:00|                2.73518261244453|
|1970-01-01T08:00:00.600+08:00|                2.371440975708148|
|1970-01-01T08:00:00.700+08:00|                2.73518261244453|
|1970-01-01T08:00:00.800+08:00|                1.7561416374270742|
+-----------------------------+-----------------------------------------+
```

6.4.2.2 诊断一维时间序列

输入序列：

```
+-----------------------------+---------------+
|                         Time|root.test.d1.s1|
+-----------------------------+---------------+
|1970-01-01T08:00:00.100+08:00|           1.0|
|1970-01-01T08:00:00.200+08:00|           2.0|
|1970-01-01T08:00:00.300+08:00|           3.0|
|1970-01-01T08:00:00.400+08:00|           4.0|
|1970-01-01T08:00:00.500+08:00|           5.0|
|1970-01-01T08:00:00.600+08:00|           6.0|
|1970-01-01T08:00:00.700+08:00|           7.0|
|1970-01-01T08:00:00.800+08:00|           8.0|
|1970-01-01T08:00:00.900+08:00|           9.0|
|1970-01-01T08:00:01.000+08:00|          10.0|
|1970-01-01T08:00:01.100+08:00|          11.0|
|1970-01-01T08:00:01.200+08:00|          12.0|
|1970-01-01T08:00:01.300+08:00|          13.0|
|1970-01-01T08:00:01.400+08:00|          14.0|
|1970-01-01T08:00:01.500+08:00|          15.0|
|1970-01-01T08:00:01.600+08:00|          16.0|
```

```
|1970-01-01T08:00:01.700+08:00|                17.0|
|1970-01-01T08:00:01.800+08:00|                18.0|
|1970-01-01T08:00:01.900+08:00|                19.0|
|1970-01-01T08:00:02.000+08:00|                20.0|
+-----------------------------+--------------------+
```

查询语句：

```
select lof(s1, "method"="series") from root.test.d1 where time<1000
```

输出序列：

```
+-----------------------------+---------------------+
|                         Time|lof(root.test.d1.s1)|
+-----------------------------+---------------------+
|1970-01-01T08:00:00.100+08:00|    3.77777777777778|
|1970-01-01T08:00:00.200+08:00|    4.32727272727273|
|1970-01-01T08:00:00.300+08:00|    4.85714285714286|
|1970-01-01T08:00:00.400+08:00|    5.40909090909091|
|1970-01-01T08:00:00.500+08:00|    5.94999999999999|
|1970-01-01T08:00:00.600+08:00|    6.43243243243243|
|1970-01-01T08:00:00.700+08:00|    6.79999999999999|
|1970-01-01T08:00:00.800+08:00|                  7.0|
|1970-01-01T08:00:00.900+08:00|                  7.0|
|1970-01-01T08:00:01.000+08:00|    6.79999999999999|
|1970-01-01T08:00:01.100+08:00|    6.43243243243243|
|1970-01-01T08:00:01.200+08:00|    5.94999999999999|
|1970-01-01T08:00:01.300+08:00|    5.40909090909091|
|1970-01-01T08:00:01.400+08:00|    4.85714285714286|
|1970-01-01T08:00:01.500+08:00|    4.32727272727273|
|1970-01-01T08:00:01.600+08:00|    3.77777777777778|
+-----------------------------+---------------------+
```

6.4.2.3

6.5 Range

6.5.1 函数简介

本函数用于查找时间序列的范围异常。将根据提供的上界与下界，判断输入数据是否越界，即异常，并输出所有异常点为新的时间序列。

函数名： RANGE

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- `lower_bound`:范围异常检测的下界。
- `upper_bound`:范围异常检测的上界。

输出序列： 输出单个序列，类型与输入序列相同。

提示： 应满足`upper_bound`大于`lower_bound`，否则将不做输出。

6.5.2 使用示例

6.5.2.1 指定上界与下界

输入序列：

```
+-----------------------------+-------------+
|                         Time|root.test.d1.s1|
+-----------------------------+-------------+
|2020-01-01T00:00:02.000+08:00|        100.0|
|2020-01-01T00:00:03.000+08:00|        101.0|
|2020-01-01T00:00:04.000+08:00|        102.0|
|2020-01-01T00:00:06.000+08:00|        104.0|
|2020-01-01T00:00:08.000+08:00|        126.0|
|2020-01-01T00:00:10.000+08:00|        108.0|
|2020-01-01T00:00:14.000+08:00|        112.0|
|2020-01-01T00:00:15.000+08:00|        113.0|
|2020-01-01T00:00:16.000+08:00|        114.0|
|2020-01-01T00:00:18.000+08:00|        116.0|
|2020-01-01T00:00:20.000+08:00|        118.0|
|2020-01-01T00:00:22.000+08:00|        120.0|
|2020-01-01T00:00:26.000+08:00|        124.0|
|2020-01-01T00:00:28.000+08:00|        126.0|
|2020-01-01T00:00:30.000+08:00|          NaN|
+-----------------------------+-------------+
```

用于查询的SQL语句：

```
select range(s1,"lower_bound"="101.0","upper_bound"="125.0") from root.test.d1 where time <= 2020-01-01
    00:00:30
```

输出序列：

```
+-----------------------------+-----------------------------------------------------------------+
|Time                         |range(root.test.d1.s1,"lower_bound"="101.0","upper_bound"="125.0")|
+-----------------------------+-----------------------------------------------------------------+
|2020-01-01T00:00:02.000+08:00|                                                            100.0|
|2020-01-01T00:00:28.000+08:00|                                                            126.0|
+-----------------------------+-----------------------------------------------------------------+
```

6.6 TwoSidedFilter

6.6.1 函数简介

本函数基于双边窗口检测法对输入序列中的异常点进行过滤。

函数名： TWOSIDEDFILTER

输出序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

输出序列： 输出单个序列，类型与输入相同，是输入序列去除异常点后的结果。

参数：

- `len`：双边窗口检测法中的窗口大小，取值范围为正整数，默认值为5.如当`len`=3时，算法向前、向后各取长度为3的窗口，在窗口中计算异常度。
- `threshold`：异常度的阈值，取值范围为(0,1)，默认值为0.3。阈值越高，函数对于异常度的判定标准越严格。

6.6.2 使用示例

输入序列：

```
+-----------------------------+-----------+
|                         Time|root.test.s0|
+-----------------------------+-----------+
|1970-01-01T08:00:00.000+08:00|     2002.0|
|1970-01-01T08:00:01.000+08:00|     1946.0|
|1970-01-01T08:00:02.000+08:00|     1958.0|
|1970-01-01T08:00:03.000+08:00|     2012.0|
|1970-01-01T08:00:04.000+08:00|     2051.0|
|1970-01-01T08:00:05.000+08:00|     1898.0|
|1970-01-01T08:00:06.000+08:00|     2014.0|
|1970-01-01T08:00:07.000+08:00|     2052.0|
|1970-01-01T08:00:08.000+08:00|     1935.0|
|1970-01-01T08:00:09.000+08:00|     1901.0|
|1970-01-01T08:00:10.000+08:00|     1972.0|
|1970-01-01T08:00:11.000+08:00|     1969.0|
|1970-01-01T08:00:12.000+08:00|     1984.0|
|1970-01-01T08:00:13.000+08:00|     2018.0|
|1970-01-01T08:00:37.000+08:00|     1484.0|
|1970-01-01T08:00:38.000+08:00|     1055.0|
|1970-01-01T08:00:39.000+08:00|     1050.0|
|1970-01-01T08:01:05.000+08:00|     1023.0|
|1970-01-01T08:01:06.000+08:00|     1056.0|
|1970-01-01T08:01:07.000+08:00|      978.0|
|1970-01-01T08:01:08.000+08:00|     1050.0|
|1970-01-01T08:01:09.000+08:00|     1123.0|
|1970-01-01T08:01:10.000+08:00|     1150.0|
|1970-01-01T08:01:11.000+08:00|     1034.0|
|1970-01-01T08:01:12.000+08:00|      950.0|
|1970-01-01T08:01:13.000+08:00|     1059.0|
+-----------------------------+-----------+
```

用于查询的SQL语句：

```
select TwoSidedFilter(s0, 'len'='5', 'threshold'='0.3') from root.test
```

输出序列：

```
+-----------------------------+-----------+
|                         Time|root.test.s0|
+-----------------------------+-----------+
|1970-01-01T08:00:00.000+08:00|     2002.0|
|1970-01-01T08:00:01.000+08:00|     1946.0|
|1970-01-01T08:00:02.000+08:00|     1958.0|
|1970-01-01T08:00:03.000+08:00|     2012.0|
|1970-01-01T08:00:04.000+08:00|     2051.0|
|1970-01-01T08:00:05.000+08:00|     1898.0|
|1970-01-01T08:00:06.000+08:00|     2014.0|
|1970-01-01T08:00:07.000+08:00|     2052.0|
|1970-01-01T08:00:08.000+08:00|     1935.0|
|1970-01-01T08:00:09.000+08:00|     1901.0|
|1970-01-01T08:00:10.000+08:00|     1972.0|
|1970-01-01T08:00:11.000+08:00|     1969.0|
|1970-01-01T08:00:12.000+08:00|     1984.0|
|1970-01-01T08:00:13.000+08:00|     2018.0|
|1970-01-01T08:01:05.000+08:00|     1023.0|
|1970-01-01T08:01:06.000+08:00|     1056.0|
|1970-01-01T08:01:07.000+08:00|      978.0|
|1970-01-01T08:01:08.000+08:00|     1050.0|
|1970-01-01T08:01:09.000+08:00|     1123.0|
|1970-01-01T08:01:10.000+08:00|     1150.0|
|1970-01-01T08:01:11.000+08:00|     1034.0|
|1970-01-01T08:01:12.000+08:00|      950.0|
|1970-01-01T08:01:13.000+08:00|     1059.0|
+-----------------------------+-----------+
```

第 7 章 频域相关

7.1 Conv

7.1.1 函数简介

本函数对两个输入序列进行卷积，即多项式乘法。

函数名： CONV

输入序列： 仅支持两个输入序列，类型均为 INT32 / INT64 / FLOAT / DOUBLE

输出序列： 输出单个序列，类型为DOUBLE，它是两个序列卷积的结果。序列的时间戳从0开始，仅用于表示顺序。

提示： 输入序列中的 NaN 将被忽略。

7.1.2 使用示例

输入序列：

```
+-----------------------------+---------------+---------------+
|                         Time|root.test.d2.s1|root.test.d2.s2|
+-----------------------------+---------------+---------------+
|1970-01-01T08:00:00.000+08:00|            1.0|            7.0|
|1970-01-01T08:00:00.001+08:00|            0.0|            2.0|
|1970-01-01T08:00:00.002+08:00|            1.0|           null|
+-----------------------------+---------------+---------------+
```

用于查询的SQL语句：

```
select conv(s1,s2) from root.test.d2
```

输出序列：

```
+-----------------------------+-----------------------------------------+
|                         Time|conv(root.test.d2.s1, root.test.d2.s2)|
+-----------------------------+-----------------------------------------+
|1970-01-01T08:00:00.000+08:00|                                      7.0|
|1970-01-01T08:00:00.001+08:00|                                      2.0|
|1970-01-01T08:00:00.002+08:00|                                      7.0|
|1970-01-01T08:00:00.003+08:00|                                      2.0|
+-----------------------------+-----------------------------------------+
```

7.2 Deconv

7.2.1 函数简介

本函数对两个输入序列进行去卷积，即多项式除法运算。

函数名： DECONV

输入序列： 仅支持两个输入序列，类型均为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- `result`：去卷积的结果，取值为'quotient'或'remainder'，分别对应于去卷积的商和余数。在缺省情况下，输出去卷积的商。

输出序列： 输出单个序列，类型为DOUBLE。它是将第二个序列从第一个序列中去卷积（第一个序列除以第二个序列）的结果。序列的时间戳从0开始，仅用于表示顺序。

提示： 输入序列中的`NaN`将被忽略。

7.2.2 使用示例

7.2.2.1 计算去卷积的商

当`result`参数缺省或为'quotient'时，本函数计算去卷积的商。

输入序列：

```
+-----------------------------+---------------+---------------+
|                         Time|root.test.d2.s3|root.test.d2.s2|
+-----------------------------+---------------+---------------+
|1970-01-01T08:00:00.000+08:00|            8.0|            7.0|
|1970-01-01T08:00:00.001+08:00|            2.0|            2.0|
|1970-01-01T08:00:00.002+08:00|            7.0|           null|
|1970-01-01T08:00:00.003+08:00|            2.0|           null|
+-----------------------------+---------------+---------------+
```

用于查询的SQL语句：

```
select deconv(s3,s2) from root.test.d2
```

输出序列：

```
+-----------------------------+-----------------------------------------+
|                         Time|deconv(root.test.d2.s3, root.test.d2.s2)|
+-----------------------------+-----------------------------------------+
|1970-01-01T08:00:00.000+08:00|                                      1.0|
|1970-01-01T08:00:00.001+08:00|                                      0.0|
|1970-01-01T08:00:00.002+08:00|                                      1.0|
+-----------------------------+-----------------------------------------+
```

7.2.2.2 计算去卷积的余数

当`result`参数为'remainder'时，本函数计算去卷积的余数。输入序列同上，用于查询的SQL语句如下：

```
select deconv(s3,s2,'result'='remainder') from root.test.d2
```

输出序列：

```
+-----------------------------+-------------------------------------------------------------------+
|                         Time|deconv(root.test.d2.s3, root.test.d2.s2, "result"="remainder")|
+-----------------------------+-------------------------------------------------------------------+
|1970-01-01T08:00:00.000+08:00|                                                                1.0|
|1970-01-01T08:00:00.001+08:00|                                                                0.0|
|1970-01-01T08:00:00.002+08:00|                                                                0.0|
|1970-01-01T08:00:00.003+08:00|                                                                0.0|
+-----------------------------+-------------------------------------------------------------------+
```

7.3 DWT

7.3.1 函数简介

本函数对输入序列进行一维离散小波变换。

函数名： DWT

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- `method`：小波滤波的类型，提供'Haar', 'DB4', 'DB6', 'DB8'，其中DB指代Daubechies。若不设置该参数，则用户需提供小波滤波的系数。不区分大小写。
- `coef`：小波滤波的系数。若提供该参数，请使用英文逗号','分割各项，不添加空格或其它符号。
- `layer`：进行变换的次数，最终输出的向量个数等同于 $layer+1$.默认取1。

输出序列： 输出单个序列，类型为DOUBLE，长度与输入相等。

提示： 输入序列长度必须为2的整数次幂。

7.3.2 使用示例

7.3.2.1 Haar变换

输入序列：

```
+-----------------------------+---------------+
|                         Time|root.test.d1.s1|
+-----------------------------+---------------+
|1970-01-01T08:00:00.000+08:00|            0.0|
|1970-01-01T08:00:00.100+08:00|            0.2|
|1970-01-01T08:00:00.200+08:00|            1.5|
|1970-01-01T08:00:00.300+08:00|            1.2|
|1970-01-01T08:00:00.400+08:00|            0.6|
|1970-01-01T08:00:00.500+08:00|            1.7|
|1970-01-01T08:00:00.600+08:00|            0.8|
|1970-01-01T08:00:00.700+08:00|            2.0|
|1970-01-01T08:00:00.800+08:00|            2.5|
```

```
|1970-01-01T08:00:00.900+08:00|                2.1|
|1970-01-01T08:00:01.000+08:00|                0.0|
|1970-01-01T08:00:01.100+08:00|                2.0|
|1970-01-01T08:00:01.200+08:00|                1.8|
|1970-01-01T08:00:01.300+08:00|                1.2|
|1970-01-01T08:00:01.400+08:00|                1.0|
|1970-01-01T08:00:01.500+08:00|                1.6|
+-----------------------------+-------------------+
```

用于查询的SQL语句：

```
select dwt(s1,"method"="haar") from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------+
|                         Time|dwt(root.test.d1.s1, "method"="haar")|
+-----------------------------+-----------------------------------------+
|1970-01-01T08:00:00.000+08:00|                  0.14142135834465192|
|1970-01-01T08:00:00.100+08:00|                   1.909188342921157|
|1970-01-01T08:00:00.200+08:00|                   1.6263456473052773|
|1970-01-01T08:00:00.300+08:00|                   1.9798989957517026|
|1970-01-01T08:00:00.400+08:00|                   3.252691126023161|
|1970-01-01T08:00:00.500+08:00|                   1.414213562373095|
|1970-01-01T08:00:00.600+08:00|                   2.1213203435596424|
|1970-01-01T08:00:00.700+08:00|                   1.8384776479437628|
|1970-01-01T08:00:00.800+08:00|                  -0.14142135834465192|
|1970-01-01T08:00:00.900+08:00|                   0.21213200063848547|
|1970-01-01T08:00:01.000+08:00|                  -0.7778174761639416|
|1970-01-01T08:00:01.100+08:00|                  -0.8485281289944873|
|1970-01-01T08:00:01.200+08:00|                   0.2828427799095765|
|1970-01-01T08:00:01.300+08:00|                  -1.414213562373095|
|1970-01-01T08:00:01.400+08:00|                   0.42426400127697095|
|1970-01-01T08:00:01.500+08:00|                  -0.42426408557066786|
+-----------------------------+-----------------------------------------+
```

7.4 FFT

7.4.1 函数简介

本函数对输入序列进行快速傅里叶变换。

函数名： FFT

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- `type`：傅里叶变换的类型，取值为'uniform'或'nonuniform'，缺省情况下为'uniform'。当取值为'uniform'时，时间戳将被忽略，所有数据点都将被视作等距的，并应用等

距快速傅里叶算法；当取值为’nonuniform’时，将根据时间戳应用非等距快速傅里叶算法（未实现）。

- result：傅里叶变换的结果，取值为’real’、’imag’、’abs’或’angle’，分别对应于变换结果的实部、虚部、模和幅角。在缺省情况下，输出变换的模。
- compress：压缩参数，取值范围(0,1]，是有损压缩时保留的能量比例。在缺省情况下，不进行压缩。

输出序列：输出单个序列，类型为DOUBLE，长度与输入相等。序列的时间戳从0开始，仅用于表示顺序。

提示： 输入序列中的NaN将被忽略。

7.4.2 使用示例

7.4.2.1 等距傅里叶变换

当type参数缺省或为’uniform’时，本函数进行等距傅里叶变换。

输入序列：

```
+-----------------------------+---------------+
|                         Time|root.test.d1.s1|
+-----------------------------+---------------+
|1970-01-01T08:00:00.000+08:00|       2.902113|
|1970-01-01T08:00:01.000+08:00|      1.1755705|
|1970-01-01T08:00:02.000+08:00|     -2.1755705|
|1970-01-01T08:00:03.000+08:00|     -1.9021131|
|1970-01-01T08:00:04.000+08:00|            1.0|
|1970-01-01T08:00:05.000+08:00|      1.9021131|
|1970-01-01T08:00:06.000+08:00|      0.1755705|
|1970-01-01T08:00:07.000+08:00|     -1.1755705|
|1970-01-01T08:00:08.000+08:00|     -0.902113|
|1970-01-01T08:00:09.000+08:00|            0.0|
|1970-01-01T08:00:10.000+08:00|       0.902113|
|1970-01-01T08:00:11.000+08:00|      1.1755705|
|1970-01-01T08:00:12.000+08:00|     -0.1755705|
|1970-01-01T08:00:13.000+08:00|     -1.9021131|
|1970-01-01T08:00:14.000+08:00|           -1.0|
|1970-01-01T08:00:15.000+08:00|      1.9021131|
|1970-01-01T08:00:16.000+08:00|      2.1755705|
|1970-01-01T08:00:17.000+08:00|     -1.1755705|
|1970-01-01T08:00:18.000+08:00|     -2.902113|
|1970-01-01T08:00:19.000+08:00|            0.0|
+-----------------------------+---------------+
```

用于查询的SQL语句：

```
select fft(s1) from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------+
|                         Time|  fft(root.test.d1.s1)|
+-----------------------------+-----------------------+
|1970-01-01T08:00:00.000+08:00|                    0.0|
|1970-01-01T08:00:00.001+08:00| 1.2727111142703152E-8|
|1970-01-01T08:00:00.002+08:00|  2.385520799101839E-7|
|1970-01-01T08:00:00.003+08:00|  8.723291723972645E-8|
|1970-01-01T08:00:00.004+08:00|     19.999999960195904|
|1970-01-01T08:00:00.005+08:00|      9.999999850988388|
|1970-01-01T08:00:00.006+08:00| 3.2260694930700566E-7|
|1970-01-01T08:00:00.007+08:00|  8.723291605373329E-8|
|1970-01-01T08:00:00.008+08:00|  1.108657103979944E-7|
|1970-01-01T08:00:00.009+08:00| 1.2727110997246171E-8|
|1970-01-01T08:00:00.010+08:00|1.9852334701272664E-23|
|1970-01-01T08:00:00.011+08:00| 1.2727111194499847E-8|
|1970-01-01T08:00:00.012+08:00|  1.108657103979944E-7|
|1970-01-01T08:00:00.013+08:00|  8.723291785769131E-8|
|1970-01-01T08:00:00.014+08:00|  3.2260694930700057E-7|
|1970-01-01T08:00:00.015+08:00|      9.999999850988388|
|1970-01-01T08:00:00.016+08:00|     19.999999960195904|
|1970-01-01T08:00:00.017+08:00|  8.723291747109068E-8|
|1970-01-01T08:00:00.018+08:00| 2.3855207991018386E-7|
|1970-01-01T08:00:00.019+08:00| 1.2727112069910878E-8|
+-----------------------------+-----------------------+
```

注：输入序列服从 $y = sin(2\pi t/4) + 2sin(2\pi t/5)$ ，长度为20，因此在输出序列中 $k = 4$ 和 $k = 5$ 处有尖峰。

7.4.2.2 等距傅里叶变换并压缩

输入序列同上，用于查询的SQL语句如下：

```
select fft(s1, 'result'='real', 'compress'='0.99'), fft(s1, 'result'='imag','compress'='0.99') from
    root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------+-----------------------+
|                         Time|  fft(root.test.d1.s1,|  fft(root.test.d1.s1,|
|                             |      "result"="real",|      "result"="imag",|
|                             |    "compress"="0.99")|    "compress"="0.99")|
+-----------------------------+-----------------------+-----------------------+
|1970-01-01T08:00:00.000+08:00|                    0.0|                    0.0|
|1970-01-01T08:00:00.001+08:00| -3.932894010461041E-9| 1.2104201863039066E-8|
|1970-01-01T08:00:00.002+08:00|-1.4021739447490164E-7| 1.9299268669082926E-7|
|1970-01-01T08:00:00.003+08:00| -7.057291240286645E-8|  5.127422242345858E-8|
|1970-01-01T08:00:00.004+08:00|     19.021130288047125|     -6.180339875198807|
```

```
|1970-01-01T08:00:00.005+08:00|      9.999999850988388| 3.501852745067114E-16|
|1970-01-01T08:00:00.019+08:00| -3.932894898639461E-9|-1.21042025493762 64E-8|
+-----------------------------+-----------------------+-----------------------+
```

注：基于傅里叶变换结果的共轭性质，压缩结果只保留前一半；根据给定的压缩参数，从低频到高频保留数据点，直到保留的能量比例超过该值；保留最后一个数据点以表示序列长度。

7.5 HighPass

7.5.1 函数简介

本函数对输入序列进行高通滤波，提取高于截止频率的分量。输入序列的时间戳将被忽略，所有数据点都将被视作等距的。

函数名： HIGHPASS

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- `wpass`：归一化后的截止频率，取值为(0,1)，不可缺省。

输出序列： 输出单个序列，类型为DOUBLE，它是滤波后的序列，长度与时间戳均与输入一致。

提示： 输入序列中的`NaN`将被忽略。

7.5.2 使用示例

输入序列：

```
+-----------------------------+--------------------+
|                         Time|root.test.d1.s1|
+-----------------------------+--------------------+
|1970-01-01T08:00:00.000+08:00|           2.902113|
|1970-01-01T08:00:01.000+08:00|          1.1755705|
|1970-01-01T08:00:02.000+08:00|         -2.1755705|
|1970-01-01T08:00:03.000+08:00|         -1.9021131|
|1970-01-01T08:00:04.000+08:00|                1.0|
|1970-01-01T08:00:05.000+08:00|          1.9021131|
|1970-01-01T08:00:06.000+08:00|          0.1755705|
|1970-01-01T08:00:07.000+08:00|         -1.1755705|
|1970-01-01T08:00:08.000+08:00|          -0.902113|
|1970-01-01T08:00:09.000+08:00|                0.0|
|1970-01-01T08:00:10.000+08:00|           0.902113|
|1970-01-01T08:00:11.000+08:00|          1.1755705|
|1970-01-01T08:00:12.000+08:00|         -0.1755705|
|1970-01-01T08:00:13.000+08:00|         -1.9021131|
|1970-01-01T08:00:14.000+08:00|               -1.0|
|1970-01-01T08:00:15.000+08:00|          1.9021131|
```

```
|1970-01-01T08:00:16.000+08:00|         2.1755705|
|1970-01-01T08:00:17.000+08:00|        -1.1755705|
|1970-01-01T08:00:18.000+08:00|        -2.902113|
|1970-01-01T08:00:19.000+08:00|               0.0|
+-----------------------------+-------------------+
```

用于查询的SQL语句：

```
select highpass(s1,'wpass'='0.45') from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------+
|                         Time|highpass(root.test.d1.s1, "wpass"="0.45")|
+-----------------------------+-----------------------------------------------+
|1970-01-01T08:00:00.000+08:00|                      0.9999999534830373|
|1970-01-01T08:00:01.000+08:00|                    1.7462829277628608E-8|
|1970-01-01T08:00:02.000+08:00|                     -0.9999999593178128|
|1970-01-01T08:00:03.000+08:00|                   -4.1115269056426626E-8|
|1970-01-01T08:00:04.000+08:00|                      0.9999999925494194|
|1970-01-01T08:00:05.000+08:00|                    3.328126513330016E-8|
|1970-01-01T08:00:06.000+08:00|                     -1.0000000183304454|
|1970-01-01T08:00:07.000+08:00|                    6.260191433311374E-10|
|1970-01-01T08:00:08.000+08:00|                      1.0000000018134796|
|1970-01-01T08:00:09.000+08:00|                   -3.097210911744423E-17|
|1970-01-01T08:00:10.000+08:00|                     -1.0000000018134794|
|1970-01-01T08:00:11.000+08:00|                   -6.260191627862097E-10|
|1970-01-01T08:00:12.000+08:00|                      1.0000000183304454|
|1970-01-01T08:00:13.000+08:00|                    -3.328126501424346E-8|
|1970-01-01T08:00:14.000+08:00|                     -0.9999999925494196|
|1970-01-01T08:00:15.000+08:00|                    4.111526915498874E-8|
|1970-01-01T08:00:16.000+08:00|                      0.9999999593178128|
|1970-01-01T08:00:17.000+08:00|                    -1.7462829341296528E-8|
|1970-01-01T08:00:18.000+08:00|                     -0.9999999534830369|
|1970-01-01T08:00:19.000+08:00|                   -1.0352372227428736E-16|
+-----------------------------+-----------------------------------------------+
```

注：输入序列服从 $y = sin(2\pi t/4) + 2sin(2\pi t/5)$，长度为20，因此高通滤波之后的输出序列服从 $y = sin(2\pi t/4)$。

7.6 IFFT

7.6.1 函数简介

本函数将输入的两个序列作为实部和虚部视作一个复数，进行逆快速傅里叶变换，并输出结果的实部。输入数据的格式参见 FFT 函数的输出，并支持以 FFT 函数压缩后的输出作为本函数的输入。

函数名： IFFT

输入序列： 仅支持两个输入序列，类型均为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- `start`：输出序列的起始时刻，是一个格式为'yyyy-MM-dd HH:mm:ss'的时间字符串。在缺省情况下，为'1970-01-01 08:00:00'。
- `interval`：输出序列的时间间隔，是一个有单位的正数。目前支持五种单位，分别是'ms'（毫秒）、's'（秒）、'm'（分钟）、'h'（小时）和'd'（天）。在缺省情况下，为1s。

输出序列： 输出单个序列，类型为DOUBLE。该序列是一个等距时间序列，它的值是将两个输入序列依次作为实部和虚部进行逆快速傅里叶变换的结果。

提示： 如果某行数据中包含空值或`NaN`，该行数据将会被忽略。

7.6.2 使用示例

输入序列：

```
+-----------------------------+--------------------+--------------------+
|                         Time|       root.test.d1.re|       root.test.d1.im|
+-----------------------------+--------------------+--------------------+
|1970-01-01T08:00:00.000+08:00|                 0.0|                 0.0|
|1970-01-01T08:00:00.001+08:00| -3.932894010461041E-9| 1.2104201863039066E-8|
|1970-01-01T08:00:00.002+08:00|-1.4021739447490164E-7| 1.9299268669082926E-7|
|1970-01-01T08:00:00.003+08:00| -7.057291240286645E-8| 5.127422242345858E-8|
|1970-01-01T08:00:00.004+08:00|    19.021130288047125|    -6.180339875198807|
|1970-01-01T08:00:00.005+08:00|    9.999999850988388| 3.501852745067114E-16|
|1970-01-01T08:00:00.019+08:00| -3.932894898639461E-9|-1.2104202549376264E-8|
+-----------------------------+--------------------+--------------------+
```

用于查询的SQL语句：

```
select ifft(re, im, 'interval'='1m', 'start'='2021-01-01 00:00:00') from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------------+
|                         Time|ifft(root.test.d1.re, root.test.d1.im, "interval"="1m",|
|                             |                                "start"="2021-01-01 00:00:00")|
+-----------------------------+-----------------------------------------------------+
|2021-01-01T00:00:00.000+08:00|                                   2.902112992431231|
|2021-01-01T00:01:00.000+08:00|                                   1.1755704705132448|
|2021-01-01T00:02:00.000+08:00|                                  -2.175570513757101|
|2021-01-01T00:03:00.000+08:00|                                  -1.9021130389094498|
|2021-01-01T00:04:00.000+08:00|                                   0.9999999925494194|
|2021-01-01T00:05:00.000+08:00|                                   1.902113046743454|
|2021-01-01T00:06:00.000+08:00|                                  0.17557053610884188|
|2021-01-01T00:07:00.000+08:00|                                  -1.1755704886020932|
|2021-01-01T00:08:00.000+08:00|                                  -0.9021130371347148|
```

```
|2021-01-01T00:09:00.000+08:00|                    3.552713678800501E-16|
|2021-01-01T00:10:00.000+08:00|                      0.9021130371347154|
|2021-01-01T00:11:00.000+08:00|                       1.1755704886020932|
|2021-01-01T00:12:00.000+08:00|                     -0.17557053610884144|
|2021-01-01T00:13:00.000+08:00|                      -1.902113046743454|
|2021-01-01T00:14:00.000+08:00|                     -0.9999999925494196|
|2021-01-01T00:15:00.000+08:00|                      1.9021130389094498|
|2021-01-01T00:16:00.000+08:00|                      2.1755705137571004|
|2021-01-01T00:17:00.000+08:00|                     -1.1755704705132448|
|2021-01-01T00:18:00.000+08:00|                     -2.902112992431231|
|2021-01-01T00:19:00.000+08:00|                    -3.552713678800501E-16|
+-----------------------------+--------------------------------------+
```

7.7 LowPass

7.7.1 函数简介

本函数对输入序列进行低通滤波，提取低于截止频率的分量。输入序列的时间戳将被忽略，所有数据点都将被视作等距的。

函数名： LOWPASS

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- `wpass`：归一化后的截止频率，取值为(0,1)，不可缺省。

输出序列： 输出单个序列，类型为DOUBLE，它是滤波后的序列，长度与时间戳均与输入一致。

提示： 输入序列中的`NaN`将被忽略。

7.7.2 使用示例

输入序列：

```
+-----------------------------+-------------+
|                         Time|root.test.d1.s1|
+-----------------------------+-------------+
|1970-01-01T08:00:00.000+08:00|     2.902113|
|1970-01-01T08:00:01.000+08:00|    1.1755705|
|1970-01-01T08:00:02.000+08:00|   -2.1755705|
|1970-01-01T08:00:03.000+08:00|   -1.9021131|
|1970-01-01T08:00:04.000+08:00|          1.0|
|1970-01-01T08:00:05.000+08:00|    1.9021131|
|1970-01-01T08:00:06.000+08:00|    0.1755705|
|1970-01-01T08:00:07.000+08:00|   -1.1755705|
|1970-01-01T08:00:08.000+08:00|    -0.902113|
|1970-01-01T08:00:09.000+08:00|          0.0|
```

```
|1970-01-01T08:00:10.000+08:00|          0.902113|
|1970-01-01T08:00:11.000+08:00|           1.1755705|
|1970-01-01T08:00:12.000+08:00|          -0.1755705|
|1970-01-01T08:00:13.000+08:00|          -1.9021131|
|1970-01-01T08:00:14.000+08:00|                -1.0|
|1970-01-01T08:00:15.000+08:00|           1.9021131|
|1970-01-01T08:00:16.000+08:00|           2.1755705|
|1970-01-01T08:00:17.000+08:00|          -1.1755705|
|1970-01-01T08:00:18.000+08:00|          -2.902113|
|1970-01-01T08:00:19.000+08:00|                 0.0|
+-----------------------------+--------------------+
```

用于查询的SQL语句：

```
select lowpass(s1,'wpass'='0.45') from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------+
|                         Time|lowpass(root.test.d1.s1, "wpass"="0.45")|
+-----------------------------+-----------------------------------------------+
|1970-01-01T08:00:00.000+08:00|                     1.9021130073323922|
|1970-01-01T08:00:01.000+08:00|                     1.1755704705132448|
|1970-01-01T08:00:02.000+08:00|                    -1.1755705286582614|
|1970-01-01T08:00:03.000+08:00|                    -1.9021130389094498|
|1970-01-01T08:00:04.000+08:00|                   7.450580419288145E-9|
|1970-01-01T08:00:05.000+08:00|                     1.902113046743454|
|1970-01-01T08:00:06.000+08:00|                     1.1755705212076808|
|1970-01-01T08:00:07.000+08:00|                    -1.1755704886020932|
|1970-01-01T08:00:08.000+08:00|                    -1.9021130222335536|
|1970-01-01T08:00:09.000+08:00|                  3.552713678800501E-16|
|1970-01-01T08:00:10.000+08:00|                     1.9021130222335536|
|1970-01-01T08:00:11.000+08:00|                     1.1755704886020932|
|1970-01-01T08:00:12.000+08:00|                    -1.1755705212076801|
|1970-01-01T08:00:13.000+08:00|                    -1.902113046743454|
|1970-01-01T08:00:14.000+08:00|                   -7.45058112983088E-9|
|1970-01-01T08:00:15.000+08:00|                     1.9021130389094498|
|1970-01-01T08:00:16.000+08:00|                     1.1755705286582616|
|1970-01-01T08:00:17.000+08:00|                    -1.1755704705132448|
|1970-01-01T08:00:18.000+08:00|                    -1.9021130073323924|
|1970-01-01T08:00:19.000+08:00|                  -2.664535259100376E-16|
+-----------------------------+-----------------------------------------------+
```

注：输入序列服从 $y = sin(2\pi t/4) + 2sin(2\pi t/5)$ ，长度为20，因此低通滤波之后的输出序列服从 $y = 2sin(2\pi t/5)$ 。

第 8 章 字符串处理

8.1 RegexMatch

8.1.1 函数简介

本函数用于正则表达式匹配文本中的具体内容并返回。

函数名： REGEXMATCH

输入序列： 仅支持单个输入序列，类型为 TEXT。

参数：

- `regex`: 匹配的正则表达式，支持所有Java正则表达式语法，比如 `\d+\.\d+\.\d+\.\d+` 将会匹配任意IPv4地址.
- `group`: 输出的匹配组序号，根据java.util.regex规定，第0组为整个正则表达式，此后的组按照左括号出现的顺序依次编号。

如 `A(B(CD))` 中共有三个组，第0组 `A(B(CD))`，第1组 `B(CD)` 和第2组 `CD`。

输出序列： 输出单个序列，类型为TEXT。

提示： 空值或无法匹配给定的正则表达式的数据点没有输出结果。

8.1.2 使用示例

输入序列：

```
+-----------------------------+-------------------------------+
|                         Time|                root.test.d1.s1|
+-----------------------------+-------------------------------+
|2021-01-01T00:00:01.000+08:00|  [192.168.0.1] [SUCCESS]|
|2021-01-01T00:00:02.000+08:00| [192.168.0.24] [SUCCESS]|
|2021-01-01T00:00:03.000+08:00|    [192.168.0.2] [FAIL]|
|2021-01-01T00:00:04.000+08:00|  [192.168.0.5] [SUCCESS]|
|2021-01-01T00:00:05.000+08:00|[192.168.0.124] [SUCCESS]|
+-----------------------------+-------------------------------+
```

用于查询的SQL语句：

```
select regexmatch(s1, "regex"="\d+\.\d+\.\d+\.\d+", "group"="0") from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------------------------------------------+
|                         Time|regexmatch(root.test.d1.s1, "regex"="\d+\.\d+\.\d+\.\d+", "group"="0")|
+-----------------------------+-----------------------------------------------------------------------------------+
|2021-01-01T00:00:01.000+08:00|                                                                        192.168.0.1|
|2021-01-01T00:00:02.000+08:00|                                                                       192.168.0.24|
|2021-01-01T00:00:03.000+08:00|                                                                        192.168.0.2|
```

```
|2021-01-01T00:00:04.000+08:00|                                                   192.168.0.5|
|2021-01-01T00:00:05.000+08:00|                                                 192.168.0.124|
+-----------------------------+--------------------------------------------------------------+
```

8.2 RegexReplace

8.2.1 函数简介

本函数用于将文本中符合正则表达式的匹配结果替换为指定的字符串。

函数名： REGEXREPLACE

输入序列： 仅支持单个输入序列，类型为 TEXT。

参数：

- `regex`：需要替换的正则表达式，支持所有Java正则表达式语法。
- `replace`：替换后的字符串，支持Java正则表达式中的后向引用，

形如'$1'指代了正则表达式`regex`中的第一个分组，并会在替换时自动填充匹配到的子串。

- `limit`：替换次数，大于等于-1的整数，默认为-1表示所有匹配的子串都会被替换。
- `offset`：需要跳过的匹配次数，即前`offset`次匹配到的字符子串并不会被替换，默认为0。
- `reverse`：是否需要反向计数，默认为false即按照从左向右的次序。

输出序列： 输出单个序列，类型为TEXT。

8.2.2 使用示例

输入序列：

```
+-----------------------------+-------------------------------+
|                         Time|                root.test.d1.s1|
+-----------------------------+-------------------------------+
|2021-01-01T00:00:01.000+08:00|      [192.168.0.1] [SUCCESS]|
|2021-01-01T00:00:02.000+08:00|     [192.168.0.24] [SUCCESS]|
|2021-01-01T00:00:03.000+08:00|         [192.168.0.2] [FAIL]|
|2021-01-01T00:00:04.000+08:00|      [192.168.0.5] [SUCCESS]|
|2021-01-01T00:00:05.000+08:00|    [192.168.0.124] [SUCCESS]|
+-----------------------------+-------------------------------+
```

用于查询的SQL语句：

```sql
select regexreplace(s1, "regex"="192\.168\.0\.(\d+)", "replace"="cluster-$1", "limit"="1") from root.
    test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------------------+
|                         Time|regexreplace(root.test.d1.s1, "regex"="192\.168\.0\.(\d+)",|
```

```
|                             |                                                                  "replace"="cluster-$1", "limit"="1")|
+-----------------------------+----------------------------------------------------------------------------------------------------+
|2021-01-01T00:00:01.000+08:00|                                                                  [cluster-1] [SUCCESS]|
|2021-01-01T00:00:02.000+08:00|                                                                 [cluster-24] [SUCCESS]|
|2021-01-01T00:00:03.000+08:00|                                                                  [cluster-2] [FAIL]|
|2021-01-01T00:00:04.000+08:00|                                                                  [cluster-5] [SUCCESS]|
|2021-01-01T00:00:05.000+08:00|                                                                [cluster-124] [SUCCESS]|
+-----------------------------+----------------------------------------------------------------------------------------------------+
```

8.3 Replace

8.3.1 函数简介

本函数用于将文本中的子串替换为指定的字符串。

函数名： REPLACE

输入序列： 仅支持单个输入序列，类型为 TEXT。

参数：

- target : 需要替换的字符子串
- replace : 替换后的字符串。
- limit : 替换次数，大于等于-1的整数，默认为-1表示所有匹配的子串都会被替换。
- offset : 需要跳过的匹配次数，即前 offset 次匹配到的字符子串并不会被替换，默认为0。
- reverse : 是否需要反向计数，默认为false即按照从左向右的次序。

输出序列： 输出单个序列，类型为TEXT。

8.3.2 使用示例

输入序列：

```
+-----------------------------+-----------------+
|                         Time|root.test.d1.s1|
+-----------------------------+-----------------+
|2021-01-01T00:00:01.000+08:00|      A,B,A+,B-|
|2021-01-01T00:00:02.000+08:00|      A,A+,A,B+|
|2021-01-01T00:00:03.000+08:00|        B+,B,B|
|2021-01-01T00:00:04.000+08:00|      A+,A,A+,A|
|2021-01-01T00:00:05.000+08:00|       A,B-,B,B|
+-----------------------------+-----------------+
```

用于查询的SQL语句：

```
select replace(s1, "target"=",", "replace"="/", "limit"="2") from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------------------+
|                         Time|replace(root.test.d1.s1, "target"=",", "replace"="/", "limit"="2")|
+-----------------------------+-----------------------------------------------------------+
|2021-01-01T00:00:01.000+08:00|                                                  A/B/A+,B-|
|2021-01-01T00:00:02.000+08:00|                                                  A/A+/A,B+|
|2021-01-01T00:00:03.000+08:00|                                                    B+/B/B|
|2021-01-01T00:00:04.000+08:00|                                                  A+/A/A+,A|
|2021-01-01T00:00:05.000+08:00|                                                  A/B-/B,B|
+-----------------------------+-----------------------------------------------------------+
```

另一个用于查询的SQL语句：

```
select replace(s1, "target"=",", "replace"="/", "limit"="1", "offset"="1", "reverse"="true") from root.
    test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------------+
|                         Time|replace(root.test.d1.s1, "target"=",", "replace"="|", |
|                             |        "limit"="1", "offset"="1", "reverse"="true")  |
+-----------------------------+-----------------------------------------------------+
|2021-01-01T00:00:01.000+08:00|                                          A,B/A+,B-|
|2021-01-01T00:00:02.000+08:00|                                          A,A+/A,B+|
|2021-01-01T00:00:03.000+08:00|                                            B+/B,B|
|2021-01-01T00:00:04.000+08:00|                                          A+,A/A+,A|
|2021-01-01T00:00:05.000+08:00|                                          A,B-/B,B|
+-----------------------------+-----------------------------------------------------+
```

8.4 Split

8.4.1 函数简介

本函数用于使用给定的正则表达式切分文本，并返回指定的项。

函数名： SPLIT

输入序列： 仅支持单个输入序列，类型为 TEXT。

参数：

- `regex`：用于分割文本的正则表达式，支持所有Java正则表达式语法，比如`['"]`将会匹配任意的英文引号`'`和`"`。
- `index`：输出结果在切分后数组中的序号，需要是大于等于-1的整数，默认值为-1表示返回切分后数组的长度，其它非负整数即表示返回数组中对应位置的切分结果（数组的秩从0开始计数）。

输出序列： 输出单个序列，在`index`为-1时输出数据类型为INT32，否则为TEXT。

提示： 如果`index`超出了切分后结果数组的秩范围，例如使用`,`切分`0,1,2`时输入`index`为3，则该数据点没有输出结果。

8.4.2 使用示例

输入序列：

```
+-----------------------------+-------------+
|                         Time|root.test.d1.s1|
+-----------------------------+-------------+
|2021-01-01T00:00:01.000+08:00|      A,B,A+,B-|
|2021-01-01T00:00:02.000+08:00|      A,A+,A,B+|
|2021-01-01T00:00:03.000+08:00|        B+,B,B|
|2021-01-01T00:00:04.000+08:00|      A+,A,A+,A|
|2021-01-01T00:00:05.000+08:00|      A,B-,B,B|
+-----------------------------+-------------+
```

用于查询的SQL语句：

```
select split(s1, "regex",", "index"="-1") from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------------+
|                         Time|split(root.test.d1.s1, "regex",", "index"="-1")|
+-----------------------------+-----------------------------------------------------+
|2021-01-01T00:00:01.000+08:00|                                                    4|
|2021-01-01T00:00:02.000+08:00|                                                    4|
|2021-01-01T00:00:03.000+08:00|                                                    3|
|2021-01-01T00:00:04.000+08:00|                                                    4|
|2021-01-01T00:00:05.000+08:00|                                                    4|
+-----------------------------+-----------------------------------------------------+
```

另一个查询的SQL语句：

```
select split(s1, "regex",", "index"="3") from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------------+
|                         Time|split(root.test.d1.s1, "regex",", "index"="3")|
+-----------------------------+-----------------------------------------------------+
|2021-01-01T00:00:01.000+08:00|                                                   B-|
|2021-01-01T00:00:02.000+08:00|                                                   B+|
|2021-01-01T00:00:04.000+08:00|                                                    A|
|2021-01-01T00:00:05.000+08:00|                                                    B|
+-----------------------------+-----------------------------------------------------+
```

第 9 章 序列发现

9.1 ConsecutiveSequences

9.1.1 函数简介

本函数用于在多维严格等间隔数据中发现局部最长连续子序列。

严格等间隔数据是指数据的时间间隔是严格相等的，允许存在数据缺失（包括行缺失和值缺失），但不允许存在数据冗余和时间戳偏移。

连续子序列是指严格按照标准时间间隔等距排布，不存在任何数据缺失的子序列。如果某个连续子序列不是任何连续子序列的真子序列，那么它是局部最长的。

函数名： CONSECUTIVESEQUENCES

输入序列： 支持多个输入序列，类型可以是任意的，但要满足严格等间隔的要求。

参数：

- `gap`：标准时间间隔，是一个有单位的正数。目前支持五种单位，分别是'ms'（毫秒）、's'（秒）、'm'（分钟）、'h'（小时）和'd'（天）。在缺省情况下，函数会利用众数估计标准时间间隔。

输出序列： 输出单个序列，类型为INT32。输出序列中的每一个数据点对应一个局部最长连续子序列，时间戳为子序列的起始时刻，值为子序列包含的数据点个数。

提示： 对于不符合要求的输入，本函数不对输出做任何保证。

9.1.2 使用示例

9.1.2.1 手动指定标准时间间隔

本函数可以通过`gap`参数手动指定标准时间间隔。需要注意的是，错误的参数设置会导致输出产生严重错误。

输入序列：

```
+-----------------------------+-------------+-------------+
|                         Time|root.test.d1.s1|root.test.d1.s2|
+-----------------------------+-------------+-------------+
|2020-01-01T00:00:00.000+08:00|          1.0|          1.0|
|2020-01-01T00:05:00.000+08:00|          1.0|          1.0|
|2020-01-01T00:10:00.000+08:00|          1.0|          1.0|
|2020-01-01T00:20:00.000+08:00|          1.0|          1.0|
|2020-01-01T00:25:00.000+08:00|          1.0|          1.0|
|2020-01-01T00:30:00.000+08:00|          1.0|          1.0|
|2020-01-01T00:35:00.000+08:00|          1.0|          1.0|
|2020-01-01T00:40:00.000+08:00|          1.0|         null|
|2020-01-01T00:45:00.000+08:00|          1.0|          1.0|
|2020-01-01T00:50:00.000+08:00|          1.0|          1.0|
```

```
+-----------------------------+-----------+-----------+
```

用于查询的SQL语句：

```
select consecutivesequences(s1,s2,'gap'='5m') from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------------------------------+
|                         Time|consecutivesequences(root.test.d1.s1, root.test.d1.s2, "gap"="5m")|
+-----------------------------+-----------------------------------------------------------------------+
|2020-01-01T00:00:00.000+08:00|                                                                      3|
|2020-01-01T00:20:00.000+08:00|                                                                      4|
|2020-01-01T00:45:00.000+08:00|                                                                      2|
+-----------------------------+-----------------------------------------------------------------------+
```

9.1.2.2 自动估计标准时间间隔

当 gap 参数缺省时，本函数可以利用众数估计标准时间间隔，得到同样的结果。因此，这种用法更受推荐。

输入序列同上，用于查询的SQL语句如下：

```
select consecutivesequences(s1,s2) from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------------------+
|                         Time|consecutivesequences(root.test.d1.s1, root.test.d1.s2)|
+-----------------------------+-----------------------------------------------------------+
|2020-01-01T00:00:00.000+08:00|                                                          3|
|2020-01-01T00:20:00.000+08:00|                                                          4|
|2020-01-01T00:45:00.000+08:00|                                                          2|
+-----------------------------+-----------------------------------------------------------+
```

9.2 ConsecutiveWindows

9.2.1 函数简介

本函数用于在多维严格等间隔数据中发现指定长度的连续窗口。

严格等间隔数据是指数据的时间间隔是严格相等的，允许存在数据缺失（包括行缺失和值缺失），但不允许存在数据冗余和时间戳偏移。

连续窗口是指严格按照标准时间间隔等距排布，不存在任何数据缺失的子序列。

函数名： CONSECUTIVEWINDOWS

输入序列： 支持多个输入序列，类型可以是任意的，但要满足严格等间隔的要求。

参数：

- gap：标准时间间隔，是一个有单位的正数。目前支持五种单位，分别是’ms’（毫秒）、’s’（秒）、’m’（分钟）、’h’（小时）和’d’（天）。在缺省情况下，函数会利用众数估计标准时间间隔。
- length：序列长度，是一个有单位的正数。目前支持五种单位，分别是’ms’（毫秒）、’s’（秒）、’m’（分钟）、’h’（小时）和’d’（天）。该参数不允许缺省。

输出序列： 输出单个序列，类型为INT32。输出序列中的每一个数据点对应一个指定长度连续子序列，时间戳为子序列的起始时刻，值为子序列包含的数据点个数。

提示： 对于不符合要求的输入，本函数不对输出做任何保证。

9.2.2 使用示例

输入序列：

```
+-----------------------------+---------------+---------------+
|                         Time|root.test.d1.s1|root.test.d1.s2|
+-----------------------------+---------------+---------------+
|2020-01-01T00:00:00.000+08:00|            1.0|            1.0|
|2020-01-01T00:05:00.000+08:00|            1.0|            1.0|
|2020-01-01T00:10:00.000+08:00|            1.0|            1.0|
|2020-01-01T00:20:00.000+08:00|            1.0|            1.0|
|2020-01-01T00:25:00.000+08:00|            1.0|            1.0|
|2020-01-01T00:30:00.000+08:00|            1.0|            1.0|
|2020-01-01T00:35:00.000+08:00|            1.0|            1.0|
|2020-01-01T00:40:00.000+08:00|            1.0|           null|
|2020-01-01T00:45:00.000+08:00|            1.0|            1.0|
|2020-01-01T00:50:00.000+08:00|            1.0|            1.0|
+-----------------------------+---------------+---------------+
```

用于查询的SQL语句：

```
select consecutivewindows(s1,s2,’length’=’10m’) from root.test.d1
```

输出序列：

```
+-----------------------------+-----------------------------------------------------------------------------+
|                         Time|consecutivewindows(root.test.d1.s1, root.test.d1.s2, "length"="10m")|
+-----------------------------+-----------------------------------------------------------------------------+
|2020-01-01T00:00:00.000+08:00|                                                                            3|
|2020-01-01T00:20:00.000+08:00|                                                                            3|
|2020-01-01T00:25:00.000+08:00|                                                                            3|
+-----------------------------+-----------------------------------------------------------------------------+
```

第 10 章 复杂事件处理

10.1 AND(TODO)

10.1.1 函数简介

本函数用于查询时间序列中具有并行关系的模式匹配，并输出匹配的个数。

函数名： SEQ

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- `WITHIN`：匹配的时间序列的时间间隔的最大值。
- `ATLEAST`：匹配的时间序列的时间间隔的最小值。

输出序列： 输出匹配的个数，类型为INT32。

10.1.2 使用示例

TODO

10.2 EventMatching(TODO)

10.3 EventNameRepair(TODO)

10.4 EventTag(TODO)

10.5 EventTimeRepair(TODO)

10.6 MissingEventRecovery(TODO)

10.7 SEQ(TODO)

10.7.1 函数简介

本函数用于查询时间序列中具有顺序关系的模式匹配，并输出匹配的个数。

函数名： SEQ

输入序列： 仅支持单个输入序列，类型为 INT32 / INT64 / FLOAT / DOUBLE

参数：

- `WITHIN`：匹配的时间序列的时间间隔的最大值。
- `ATLEAST`：匹配的时间序列的时间间隔的最小值。

输出序列： 输出匹配的个数，类型为INT32。

10.7.2 使用示例

TODO