



タイムシリーズ関数の活用

(ギャップ値補完、パターンマッチング、R言語による予測分析)

Maurizio Felici
APJ Bootcamp 05 June 2018

アジェンダ

- Vertica イベントベース関数
- Vertica タイムシリーズ関数 (ギャップ値補完)
- Vertica イベントシリーズ結合
- Time Series Similarity
- タイムシリーズ パターンマッチング
- タイムシリーズ 予測分析
- 並列処理に関する補足情報

Vertica イベントベース関数

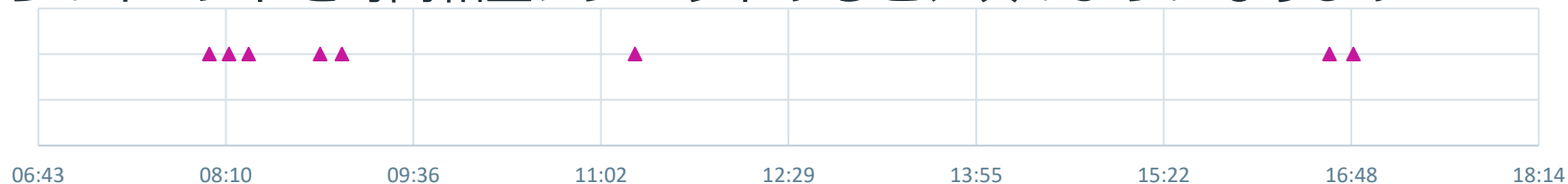
Vertica イベントベース関数

- 重要なイベントを使用してウィンドウの「境界線」を定義できます
- タイムスタンプで表される一連のイベントがあるとしたします

```
SQL> select * from ts1 order by ts ;
      ts
```

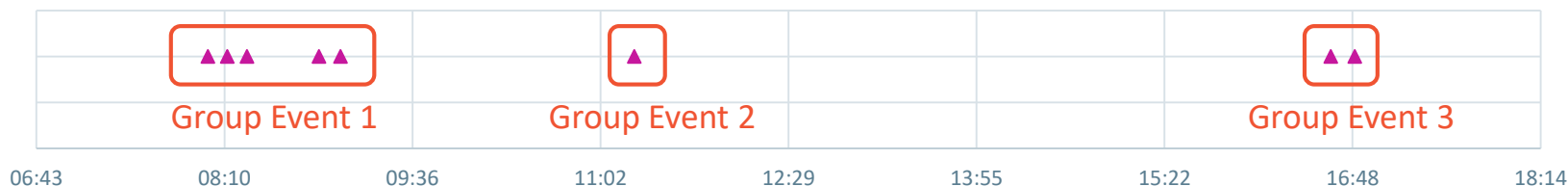
```
-----
2015-05-21 23:08:02
2015-05-21 23:08:11
2015-05-21 23:08:20
2015-05-21 23:08:53
2015-05-21 23:09:03
2015-05-21 23:11:18
2015-05-21 23:16:38
2015-05-21 23:16:49
```

- これらのイベントを時間軸上にプロットすると、次のようになります



Vertica イベントベース関数

- 1つまたは複数の単一イベントからなるグループ・イベントを識別します。「単一イベント」は、60秒のウィンドウ内で発生する場合、グループイベントは以下のように可視化できます



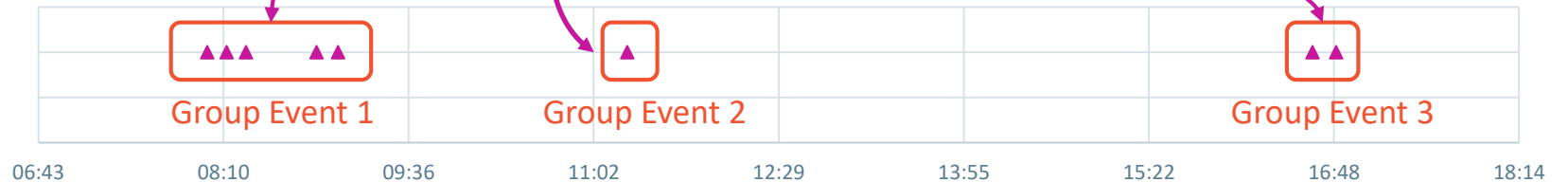
- 各グループ・イベントは、次の属性によって記述されます
 - スタート タイムスタンプ
 - エンド タイムスタンプ
 - グループイベント サイズ (例、イベントグループ内のイベント数)
- 応用例
 - ウェブサイトのセッション定義：特定のIPアドレスからのすべてのアクセスと定義された時間間隔=セッション
 - ミニ株式市場取引の疑わしい「バースト」を分析
 - カスタマーインタラクション分析（予測モデルの変更、購入意欲...） etc...

Vertica イベントベース関数

- グループ谷でイベントを分析します

```
SQL> select min(ts) as event_start,  
           max(ts) as event_end,  
           count(*) as event_size  
from ( select ts,  
           conditional_true_event ( ts - lag(ts) > '60 seconds' ) over (order by ts) as cce from ts1 ) x  
group by cce order by 1 ;
```

event_start	event_end	event_size
2015-05-21 23:08:02	2015-05-21 23:09:03	5
2015-05-21 23:11:18	2015-05-21 23:11:18	1
2015-05-21 23:16:38	2015-05-21 23:16:49	2



Vertica イベントベース関数

- 基本的な動作を確認します

```
SQL> select ts,  
           conditional_true_event ( ts - lag(ts) > '60 seconds' ) over (order by ts) as cce  
from   ts1 ;
```

ts	cce
2015-05-21 23:08:02	0
2015-05-21 23:08:11	0
2015-05-21 23:08:20	0
2015-05-21 23:08:53	0
2015-05-21 23:09:03	0
2015-05-21 23:11:18	1
2015-05-21 23:16:38	2
2015-05-21 23:16:49	2

Group-event 1

Group-event 2

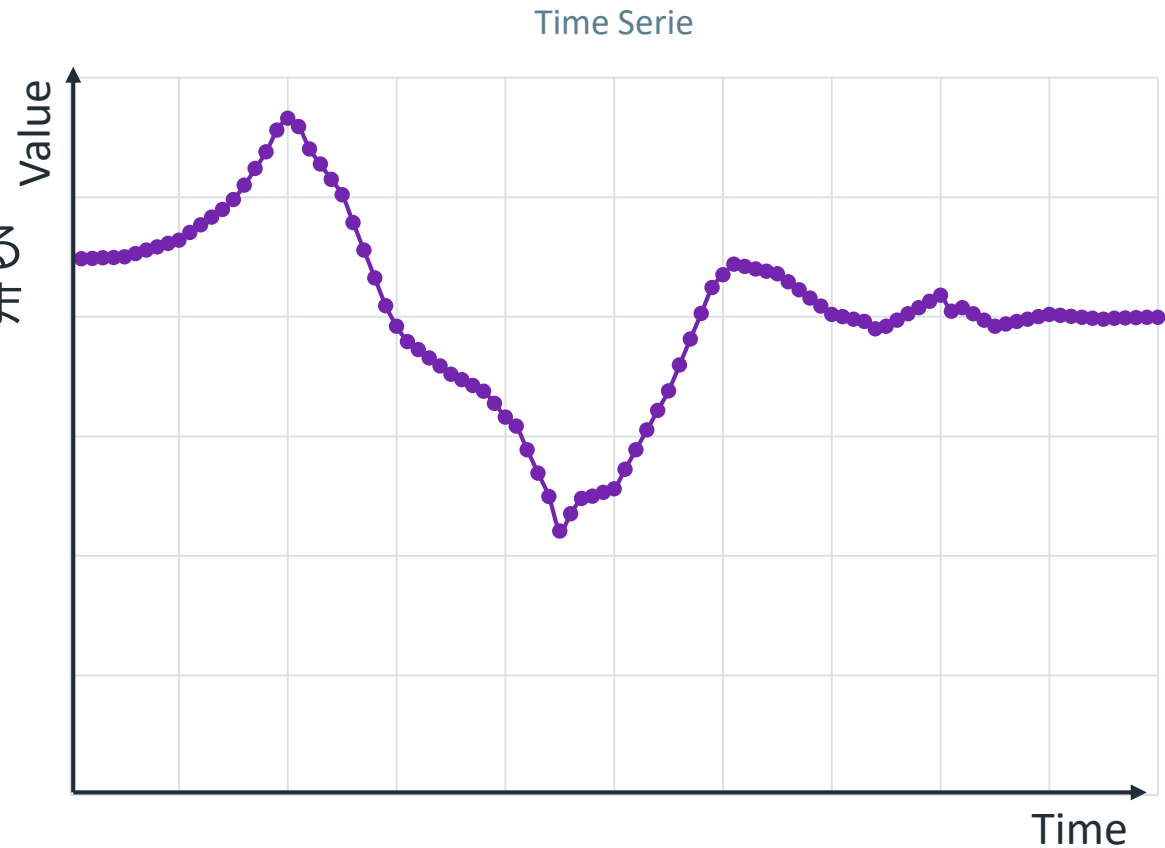
Group-event 3

現在の行のTS値とその前のTSの値を比較し、60秒以上の差がある場合、Boolean expressionはTRUEとなります。Boolean expressionがTRUEの場合、conditional_true_event() 関数は、1 つ数値を増加します。

Vertica タイムシリーズ関数 (ギャップ値補完)

タイムシリーズとは?

- タイムシリーズは、以下の2つのデータで表現されます
 - 変数の値
 - 時系列データ
- タイムシリーズは以下の代表的される様々なユースケースで活用されています
 - 株価
 - 温度
 - 国内総生産
 - ウェブページへのユニークな訪問者



Vertica タイムシリーズ関数

- 通常、変数値が特定のサンプリング間隔で測定される離散的な時系列データを扱います
- タイムシリーズは、通常、2つのディメンショングラフで表されます
 - 時間のX軸
 - 変数値のY軸
- 時系列解析は、以下の場合には困難です
 - ギャップ
 - 不完全なタイムシリーズ
 - 不均一な時間間隔
- タイムシリーズ関数は、時間の経過とともに変数を分析し、グループ化する非常に便利な Vertica SQL 拡張関数です
 - SELECT ... TIMESERIES句
 - TS_FIRST_VALUEおよびTS_LAST_VALUE集約関数

SELECT ... TIMESERIES句

- 利用方法は、他の分析関数と同じ方法で利用できます

```
TIMESERIES slice_time AS time_interval  
  OVER (  
    [ window_partition_clause ]  
    window_order_clause  
  )
```

- スライス時間は、TIMESERIES句によって指定する時間列です
- 時間間隔は、タイムスライスの長さです
- ウィンドウ パーティション句 (PARTITION BY):
 - (オプション) パーティションを他の関数と同様に利用
- Window Order句 (ORDER BY):
 - (必須) データソートに利用

タイムシリーズ ギャップ補完関数

- 前のセクションで使用したのと同じタイムスタンプのリスト（EVENT BASEDウィンドウ関数）を利用します

```
SQL> select * from ts1 ;  
      ts
```

```
-----  
2015-05-21 23:08:02  
2015-05-21 23:08:11  
2015-05-21 23:08:20  
2015-05-21 23:08:53  
2015-05-21 23:09:03  
2015-05-21 23:11:18  
2015-05-21 23:16:38  
2015-05-21 23:16:49
```

ギャップ補完



```
SQL> select tm from ts1  
      timeseries tm as '1 minute' over ( order by ts ) ;  
      tm
```

```
-----  
2015-05-21 23:08:00  
2015-05-21 23:09:00  
2015-05-21 23:10:00  
2015-05-21 23:11:00  
2015-05-21 23:12:00  
2015-05-21 23:13:00  
2015-05-21 23:14:00  
2015-05-21 23:15:00  
2015-05-21 23:16:00
```

- SSELECT ... TIMESERIESは毎分1つの「タイムポイント」を作成しました。
- 注意点：元のテーブルには存在しない「tm」を選択しました

タイムシリーズ データ補完関数

- 新しい列を追加して、「実際の」時系列を作成します

```
SQL> select * from tseries  
      order by ts ;
```

ts	value
2015-05-21 23:08:02	27
2015-05-21 23:08:11	26
2015-05-21 23:08:20	30
2015-05-21 23:08:52	28
2015-05-21 23:09:03	27
2015-05-21 23:11:18	20
2015-05-21 23:16:38	13
2015-05-21 23:16:49	12

ギャップ補完（直前の値を補完）

```
SQL> select tm, ts_first_value(value) as int_value from tseries  
      timeseries tm as '1 minute' over ( order by ts ) ;
```

tm	int_value
2015-05-21 23:08:00	
2015-05-21 23:09:00	28
2015-05-21 23:10:00	27
2015-05-21 23:11:00	27
2015-05-21 23:12:00	20
2015-05-21 23:13:00	20
2015-05-21 23:14:00	20
2015-05-21 23:15:00	20
2015-05-21 23:16:00	20

- TS_FIRST_VALUEがタイムスライスの初期値をとり、23:08:00に値がないため、最初の値が欠落しています

タイムシリーズ データ補完関数

- タイムスライスの最後の値を取得する異なる関数を利用します

```
SQL> select * from tseries  
      order by ts ;
```

ts	value
2015-05-21 23:08:02	27
2015-05-21 23:08:11	26
2015-05-21 23:08:20	30
2015-05-21 23:08:52	28
2015-05-21 23:09:03	27
2015-05-21 23:11:18	20
2015-05-21 23:16:38	13
2015-05-21 23:16:49	12

Gap Filling AND Interpolation

```
SQL> select tm, ts_last_value(value) as int_value from tseries  
      timeseries tm as '1 minute' over ( order by ts ) ;
```

tm	int_value
2015-05-21 23:08:00	28
2015-05-21 23:09:00	27
2015-05-21 23:10:00	27
2015-05-21 23:11:00	20
2015-05-21 23:12:00	20
2015-05-21 23:13:00	20
2015-05-21 23:14:00	20
2015-05-21 23:15:00	20
2015-05-21 23:16:00	12

TS_LAST_VALUEはタイムスライスの最終値を取るため、最初の値を採用しません。

タイムシリーズ データ補完関数

- Verticaは2つの異なる補間方法を使用できます
 - コンスタント補間
 - 線形補間
- Verticaが使用する補間方法を指定しない場合、デフォルトでコンスタント補間が採用されます。
- 補間方法は、TS_FIRST_VALUE () / TS_LAST_VALUE () 関数で指定します:

TS_FIRST_VALUE (expression [IGNORE NULLS] ... [, { 'CONST' | 'LINEAR' }])

TS_LAST_VALUE (expression [IGNORE NULLS] ... [, { 'CONST' | 'LINEAR' }])

- IGNORE NULLオプションの動作は、特定の関数によって異なります（後述）
- 適切な補完方法はデータタイプに応じて選択してください（例）
 - CONST: 株式相場を分析
 - LINEAR: 温度

タイムシリーズ データ補完関数

- オプションLINEAR の動作を確認します

```
SQL> select * from tseries  
      order by ts ;
```

ts	value
2015-05-21 23:08:02	27
2015-05-21 23:08:11	26
2015-05-21 23:08:20	30
2015-05-21 23:08:52	28
2015-05-21 23:09:03	27
2015-05-21 23:11:18	20
2015-05-21 23:16:38	13
2015-05-21 23:16:49	12

線形でデータ補完

```
SQL> select tm, ts_first_value(value, 'LINEAR') as int_value from tseries  
      timeseries tm as '1 minute' over ( order by ts ) ;
```

tm	int_value
2015-05-21 23:08:00	
2015-05-21 23:09:00	27.2727272727273
2015-05-21 23:10:00	24.0444444444444
2015-05-21 23:11:00	20.9333333333333
2015-05-21 23:12:00	19.08125
2015-05-21 23:13:00	17.76875
2015-05-21 23:14:00	16.45625
2015-05-21 23:15:00	15.14375
2015-05-21 23:16:00	13.83125

タイムシリーズ タイムスタンプ列情報のNULL値補完

- 時系列は、「時間軸」または「値軸」のいずれかにNULLを処理することも可能です。時間軸にNULLの場合、以下の方法で補完することが可能です

```
SQL> select * from tseries_null
      ts          | value
-----+-----
      (null)      | 27
2015-05-21 23:08:11 | 26
2015-05-21 23:08:20 | 30
2015-05-21 23:08:52 | 28
      (null)      | 27
2015-05-21 23:11:18 | 20
2015-05-21 23:16:38 | 13
2015-05-21 23:16:49 | 12
```

Gap Filling and Interpolation with NULLs on the time axis

```
SQL> select tm, ts_last_value(value) as int_value from tseries_null
      timeseries tm as '1 minute' over ( order by ts ) ;
      tm          | int_value
-----+-----
2015-05-21 23:08:00 | 28
2015-05-21 23:09:00 | 28
2015-05-21 23:10:00 | 28
2015-05-21 23:11:00 | 20
2015-05-21 23:12:00 | 20
2015-05-21 23:13:00 | 20
2015-05-21 23:14:00 | 20
2015-05-21 23:15:00 | 20
2015-05-21 23:16:00 | 12
```

タイムシリーズ NULL値の補完（定数補間）

- NULLが値の列にある場合を考えてみましょう。time_sliceの最初/最後の値がNULLの場合、デフォルトでVerticaはNULL補間値を生成します

```
SQL> select * from tseries_null
      ts          | value
-----+-----
2015-05-21 23:08:02 | (null)
2015-05-21 23:08:11 | 26
2015-05-21 23:08:20 | 30
2015-05-21 23:08:52 | 28
2015-05-21 23:09:03 | (null)
2015-05-21 23:11:18 | 20
2015-05-21 23:16:38 | 13
2015-05-21 23:16:49 | 12
```

Gap Filling and Interpolation with NULLs on the value axis

```
SQL> select tm, ts_last_value(value) as int_value from tseries_null
      timeseries tm as '1 minute' over ( order by ts ) ;
      tm          | int_value
-----+-----
2015-05-21 23:08:00 | 28
2015-05-21 23:09:00 | (null)
2015-05-21 23:10:00 | (null)
2015-05-21 23:11:00 | 20
2015-05-21 23:12:00 | 20
2015-05-21 23:13:00 | 20
2015-05-21 23:14:00 | 20
2015-05-21 23:15:00 | 20
2015-05-21 23:16:00 | 12
```

- 注意：デフォルトの補間方法はCONSTANTです。

タイムシリーズ NULL値の補完（定数補間）

- VerticaにIGNORE NULLSを指示すると、タイムスライスの前/次のLAST / FIRST値が使用されます

```
SQL> select * from tseries_null
      ts          | value
-----+-----
2015-05-21 23:08:02 | (null)
2015-05-21 23:08:11 | 26
2015-05-21 23:08:20 | 30
2015-05-21 23:08:52 | 28
2015-05-21 23:09:03 | (null)
2015-05-21 23:11:18 | 20
2015-05-21 23:16:38 | 13
2015-05-21 23:16:49 | 12
```

Gap Filling and Interpolation with NULLs on the value axis

```
SQL> select tm, ts_last_value(value ignore nulls) as int_value
      from tseries_null timeseries tm as '1 minute' over(order by ts);
      tm          | int_value
-----+-----
2015-05-21 23:08:00 | 28
2015-05-21 23:09:00 | 28 -- 28 was the last NON NULL value
2015-05-21 23:10:00 | 28
2015-05-21 23:11:00 | 20
2015-05-21 23:12:00 | 20
2015-05-21 23:13:00 | 20
2015-05-21 23:14:00 | 20
2015-05-21 23:15:00 | 20
2015-05-21 23:16:00 | 12
```

タイムシリーズ NULL値の補完（線形補間）

- LINEAR補間では、タイムスライス内に少なくとも2つの値が必要です。1がNULLの場合、Verticaの補間値もNULLになります

```
SQL> select * from tseries_null
      ts          | value
-----+-----
2015-05-21 23:08:02 | (null)
2015-05-21 23:08:11 | 26
2015-05-21 23:08:20 | 30
2015-05-21 23:08:52 | 28
2015-05-21 23:09:03 | (null)
2015-05-21 23:11:18 | 20
2015-05-21 23:16:38 | 13
2015-05-21 23:16:49 | 12
```

Gap Filling and Interpolation with NULLs on the value axis

```
SQL> select tm, ts_last_value(value, 'linear') as int_value
      tm          | int_value
-----+-----
2015-05-21 23:08:00 |
2015-05-21 23:09:00 |
2015-05-21 23:10:00 |
2015-05-21 23:11:00 | 19.08125
2015-05-21 23:12:00 | 17.76875
2015-05-21 23:13:00 | 16.45625
2015-05-21 23:14:00 | 15.14375
2015-05-21 23:15:00 | 13.83125
2015-05-21 23:16:00 |
```

タイムシリーズ NULL値の補完（線形補間）

- IGNORE NULLSにLINEAR補間を指定すると、VerticaはNULL値をスキップし、欠損値の代わりに最も近い値を使用します

```
SQL> select * from tseries_null
      ts          | value
-----+-----
2015-05-21 23:08:02 | (null)
2015-05-21 23:08:11 | 26
2015-05-21 23:08:20 | 30
2015-05-21 23:08:52 | 28
2015-05-21 23:09:03 | (null)
2015-05-21 23:11:18 | 20
2015-05-21 23:16:38 | 13
2015-05-21 23:16:49 | 12
```

Gap Filling and Interpolation with NULLs on the value axis

```
SQL> select tm, ts_last_value(value ignore nulls, 'linear') as int_value
      from tseries_null timeseries tm as '1 minute' over(order by ts);
      tm          | int_value
-----+-----
2015-05-21 23:08:00 | 27.5616438356164
2015-05-21 23:09:00 | 24.2739726027397
2015-05-21 23:10:00 | 20.986301369863
2015-05-21 23:11:00 | 19.08125
2015-05-21 23:12:00 | 17.76875
2015-05-21 23:13:00 | 16.45625
2015-05-21 23:14:00 | 15.14375
2015-05-21 23:15:00 | 13.83125
2015-05-21 23:16:00 |
```

時間間隔ベースの集計


時間間隔ベースの集計

- 時間間隔に基づく集計は一般的に利用されます
- 時間の経過とともに不規則に分布するデータ（購入、ウェブサイト訪問など）を定期的に集計することができます
 - 例えば：
 - 毎分MIN / MAXの温度
 - 1時間あたりの平均訪問者数
- TIME_SLICE () 関数で簡単に処理することが可能です

TIME_SLICE関数

- 時系列のテーブルがあり、イベント数とイベント値の合計を1分間隔でカウントします

```
SQL> select * from tseries;
      ts          | value
-----+-----
2015-05-21 23:08:02 |    27
2015-05-21 23:08:11 |    26
2015-05-21 23:08:20 |    30
2015-05-21 23:08:52 |    28
2015-05-21 23:09:03 |    27
2015-05-21 23:11:18 |    20
2015-05-21 23:16:38 |    13
2015-05-21 23:16:49 |    12
```



```
SQL> select time_slice(ts, 1, 'minute'),
           count(*) as num, sum(value)
           from tseries group by 1 order by 1 ;
```

time_slice	num	sum
2015-05-21 23:08:00	4	111
2015-05-21 23:09:00	1	27
2015-05-21 23:11:00	1	20
2015-05-21 23:16:00	2	25

TIME_SLICE関数

- TIME_SLICE () はデータの無い時間間隔を「スキップ」します。すべての時間間隔（データなしのものを除いて、以下の処理を実施します

```
SQL> select * from tseries;
      ts          | value
-----+-----
2015-05-21 23:08:02 |    27
2015-05-21 23:08:11 |    26
2015-05-21 23:08:20 |    30
2015-05-21 23:08:52 |    28
2015-05-21 23:09:03 |    27
2015-05-21 23:11:18 |    20
2015-05-21 23:16:38 |    13
2015-05-21 23:16:49 |    12
```

```
select s.tm, t.num, t.sum from
  ( select time_slice(ts, 1, 'minute') as tm,
    count(*) as num, sum(value) as sum
    from tseries group by 1 ) t
right outer join
  ( select tm from tseries timeseries tm
    as '1 minute' over ( order by ts ) ) s
using ( tm );
```

tm	num	sum
2015-05-21 23:08:00	4	111
2015-05-21 23:09:00	1	27
2015-05-21 23:10:00	0	0
2015-05-21 23:11:00	1	20
2015-05-21 23:12:00	0	0
2015-05-21 23:13:00	0	0
2015-05-21 23:14:00	0	0
2015-05-21 23:15:00	0	0
2015-05-21 23:16:00	2	25

(9 rows)

Vertica イベントシリーズ結合

イベントシリーズ結合

- イベントシリーズ結合は、OUTER JOINのVertica SQL拡張です
- この機能により、正確にアライメントされない時系列の表を結合することが可能になりました
- OUTER JOINで次の2つのテーブルが一致しない場合（赤で表示）、NULLで「パディング」します

```
SQL> select * from ts1;
```

ts	val
2015-05-21 23:08:02	27
2015-05-21 23:08:11	26
2015-05-21 23:08:20	30
2015-05-21 23:08:52	28
2015-05-21 23:09:03	27
2015-05-21 23:11:18	20
2015-05-21 23:16:38	13
2015-05-21 23:16:49	12

```
SQL> select * from ts2;
```

ts	val
2015-05-21 23:08:02	37
2015-05-21 23:08:13	36
2015-05-21 23:08:20	40
2015-05-21 23:08:53	38
2015-05-21 23:09:14	37
2015-05-21 23:11:18	30
2015-05-21 23:16:38	23
2015-05-21 23:16:49	22

```
SQL> select * from ts1 full outer join ts2
      on ( ts1.ts = ts2.ts );
```

ts	val	ts	val
2015-05-21 23:08:02	27	2015-05-21 23:08:02	37
2015-05-21 23:09:03	27		
2015-05-21 23:11:18	20	2015-05-21 23:11:18	30
		2015-05-21 23:08:13	36
		2015-05-21 23:09:14	37
2015-05-21 23:08:11	26		
2015-05-21 23:08:20	30	2015-05-21 23:08:20	40
2015-05-21 23:16:49	12	2015-05-21 23:16:49	22
2015-05-21 23:08:52	28		
2015-05-21 23:16:38	13	2015-05-21 23:16:38	23
		2015-05-21 23:08:53	38

FULL OUTER JOIN

イベントシリーズ結合

- イベントシリーズ結合は、「ギャップ」を直前の値で置き換えます

```
SQL> select * from ts1;
```

ts	val
2015-05-21 23:08:02	27
2015-05-21 23:08:11	26
2015-05-21 23:08:20	30
2015-05-21 23:08:52	28
2015-05-21 23:09:03	27
2015-05-21 23:11:18	20
2015-05-21 23:16:38	13
2015-05-21 23:16:49	12

```
SQL> select * from ts2;
```

ts	val
2015-05-21 23:08:02	37
2015-05-21 23:08:13	36
2015-05-21 23:08:20	40
2015-05-21 23:08:53	38
2015-05-21 23:09:14	37
2015-05-21 23:11:18	30
2015-05-21 23:16:38	23
2015-05-21 23:16:49	22

```
SQL select * from ts1 a full outer join ts2 b  
on ( a.ts interpolate previous value b.ts );
```

ts	val	ts	val
2015-05-21 23:08:02	27	2015-05-21 23:08:02	37
2015-05-21 23:08:11	26	2015-05-21 23:08:02	37
2015-05-21 23:08:11	26	2015-05-21 23:08:13	36
2015-05-21 23:08:20	30	2015-05-21 23:08:20	40
2015-05-21 23:08:52	28	2015-05-21 23:08:20	40
2015-05-21 23:08:52	28	2015-05-21 23:08:53	38
2015-05-21 23:09:03	27	2015-05-21 23:08:53	38
2015-05-21 23:09:03	27	2015-05-21 23:09:14	37
2015-05-21 23:11:18	20	2015-05-21 23:11:18	30
2015-05-21 23:16:38	13	2015-05-21 23:16:38	23
2015-05-21 23:16:49	12	2015-05-21 23:16:49	22

(Interpolated) FULL OUTER JOIN

イベントシリーズ結合

- 同様に「通常の」LEFT OUTER JOINの場合、次のようになります

```
SQL> select * from ts1;
```

ts	val
2015-05-21 23:08:02	27
2015-05-21 23:08:11	26
2015-05-21 23:08:20	30
2015-05-21 23:08:52	28
2015-05-21 23:09:03	27
2015-05-21 23:11:18	20
2015-05-21 23:16:38	13
2015-05-21 23:16:49	12

```
SQL> select * from ts2;
```

ts	val
2015-05-21 23:08:02	37
2015-05-21 23:08:13	36
2015-05-21 23:08:20	40
2015-05-21 23:08:53	38
2015-05-21 23:09:14	37
2015-05-21 23:11:18	30
2015-05-21 23:16:38	23
2015-05-21 23:16:49	22

```
SQL> select * from ts1 a left outer join ts2 b  
on ( a.ts = b.ts );
```

ts	val	ts	val
2015-05-21 23:08:02	27	2015-05-21 23:08:02	37
2015-05-21 23:09:03	27		
2015-05-21 23:11:18	20	2015-05-21 23:11:18	30
2015-05-21 23:08:11	26		
2015-05-21 23:08:20	30	2015-05-21 23:08:20	40
2015-05-21 23:16:49	12	2015-05-21 23:16:49	22
2015-05-21 23:08:52	28		
2015-05-21 23:16:38	13	2015-05-21 23:16:38	23

LEFT OUTER JOIN

イベントシリーズ結合

- 「補間したLEFT OUTER JOIN」は、（RIGHT OUTER JOINと類似）結果になります

```
SQL> select * from ts1;
```

ts	val
2015-05-21 23:08:02	27
2015-05-21 23:08:11	26
2015-05-21 23:08:20	30
2015-05-21 23:08:52	28
2015-05-21 23:09:03	27
2015-05-21 23:11:18	20
2015-05-21 23:16:38	13
2015-05-21 23:16:49	12

```
SQL> select * from ts2;
```

ts	val
2015-05-21 23:08:02	37
2015-05-21 23:08:13	36
2015-05-21 23:08:20	40
2015-05-21 23:08:53	38
2015-05-21 23:09:14	37
2015-05-21 23:11:18	30
2015-05-21 23:16:38	23
2015-05-21 23:16:49	22

```
SQL> select * from ts1 a left outer join ts2 b
```

```
on ( a.ts interpolate previous value b.ts );
```

ts	val	ts	val
2015-05-21 23:08:02	27	2015-05-21 23:08:02	37
2015-05-21 23:08:11	26	2015-05-21 23:08:02	37
2015-05-21 23:08:20	30	2015-05-21 23:08:20	40
2015-05-21 23:08:52	28	2015-05-21 23:08:20	40
2015-05-21 23:09:03	27	2015-05-21 23:08:53	38
2015-05-21 23:11:18	20	2015-05-21 23:11:18	30
2015-05-21 23:16:38	13	2015-05-21 23:16:38	23
2015-05-21 23:16:49	12	2015-05-21 23:16:49	22

(Interpolated) LEFT OUTER JOIN

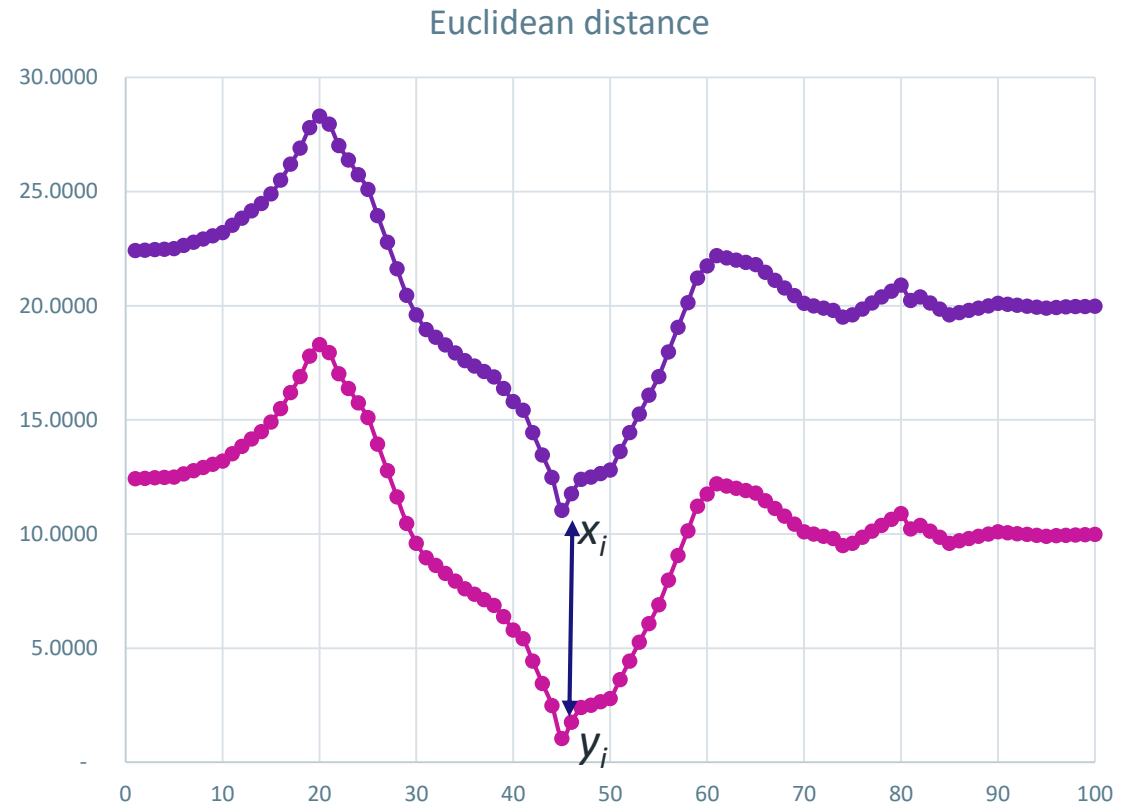
タイムシリーズの応用 ～類似パターン検知～

Euclidean distance to measure (dis)similarity

- You want to measure similarity between two Time Series having the **same length and frequency**
- You can use Euclidean Distance defined as:

$$L_2(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

- As you can see we need values from both time-series with homogeneous “sampling rate”
- If the sampling rate is different...we can pre-process both time series using Vertica’s **Gap Filling** to “add” our missing points with either constant or linear **Interpolation**



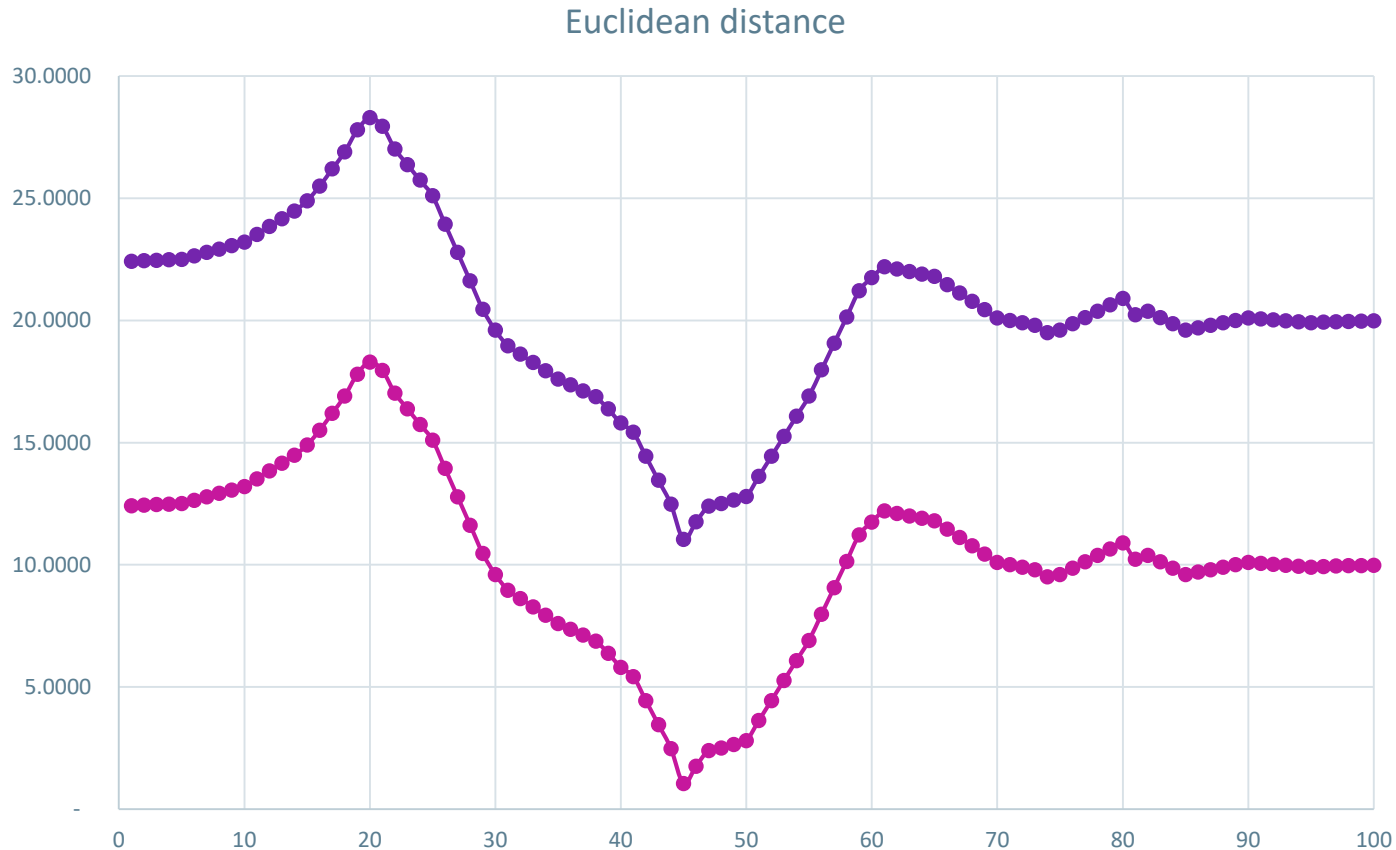
Test environment

```
$ cat tsample.csv
ts,val1,val2,val3,val4
2016-01-01 09:00:05,12.420,22.420,6.210,13.893
2016-01-01 09:00:06,12.440,22.440,6.220,11.797
2016-01-01 09:00:07,12.460,22.460,6.230,13.060
2016-01-01 09:00:08,12.480,22.480,6.240,11.951
...

-- Create tsample table
SQL> create table tsample (
    ts timestamp ,
    val float,
    val2 float,
    val3 float,
    val4 float);

-- load sample data
SQL> copy tsample from local '/Users/mauro/tsample.csv' delimiter ',' abort on error direct skip 1;
```

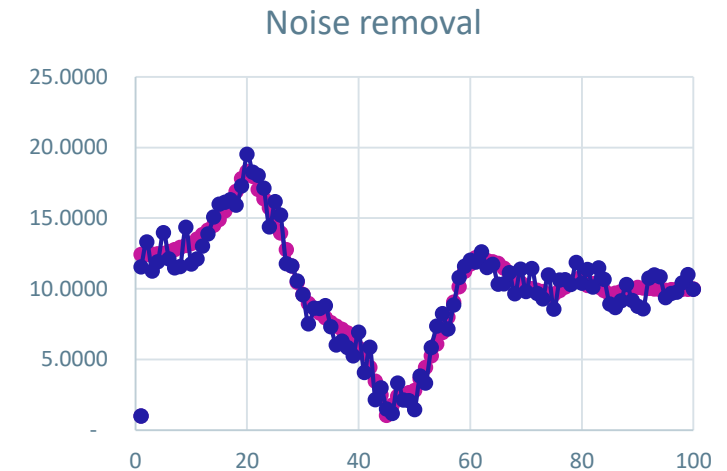
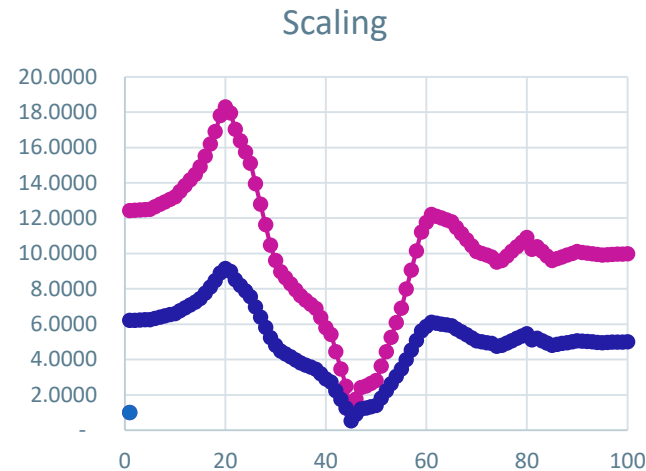
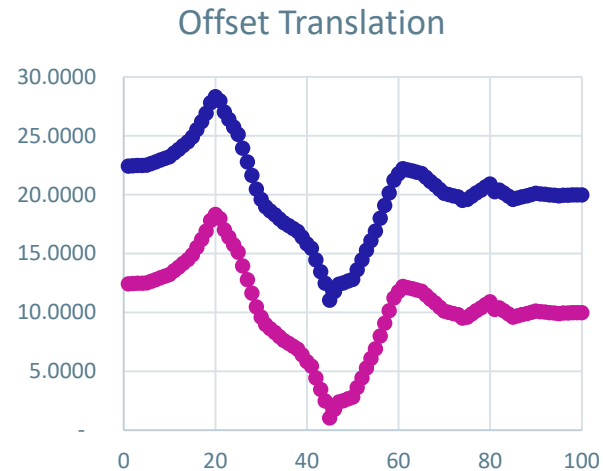
Euclidean distance in SQL



```
select  
    sqrt(sum(power(val-val2, 2)))  
from  
    tsample ;
```

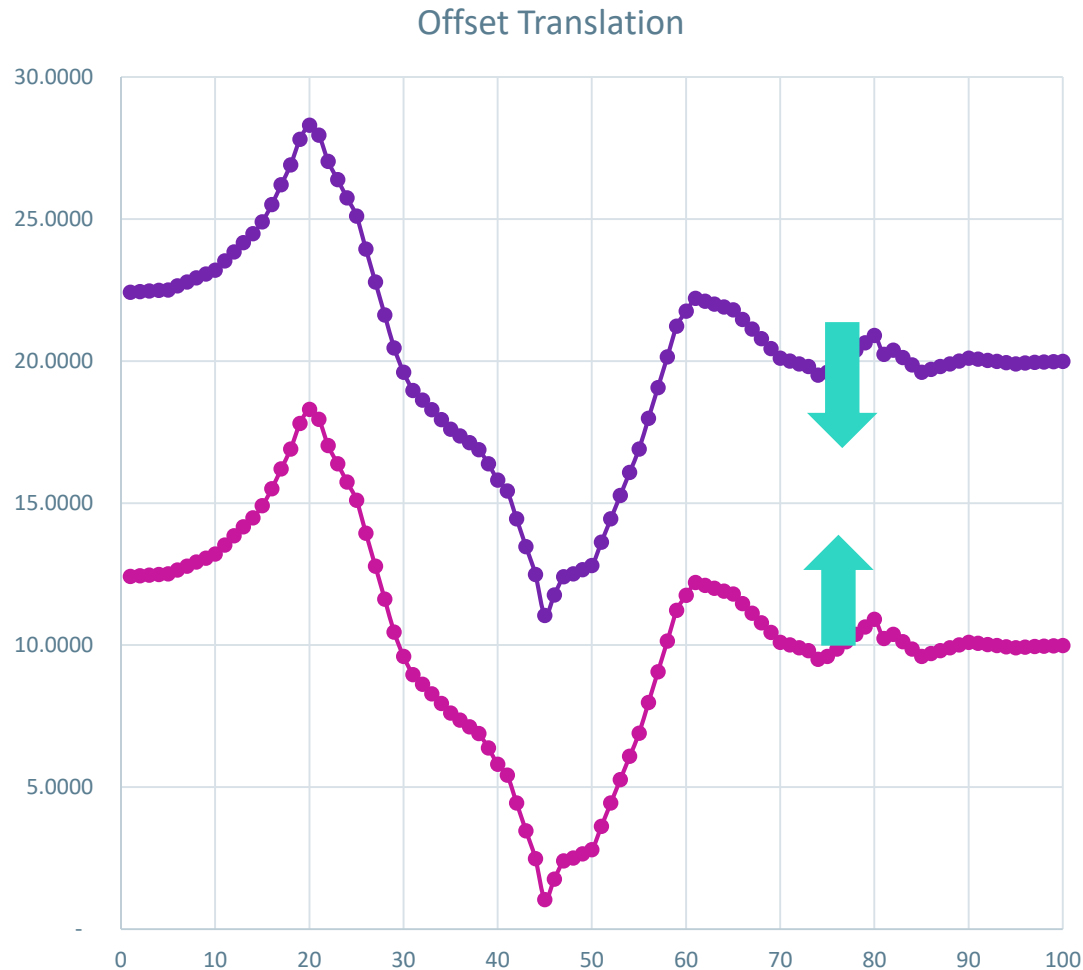
Time Series “pattern matching”

- Standard Euclidean Distance (dis)similarity measures can be influenced by several “disturbing” factors:



- In these cases you might need to pre-process them before calculating the Euclidean Distance

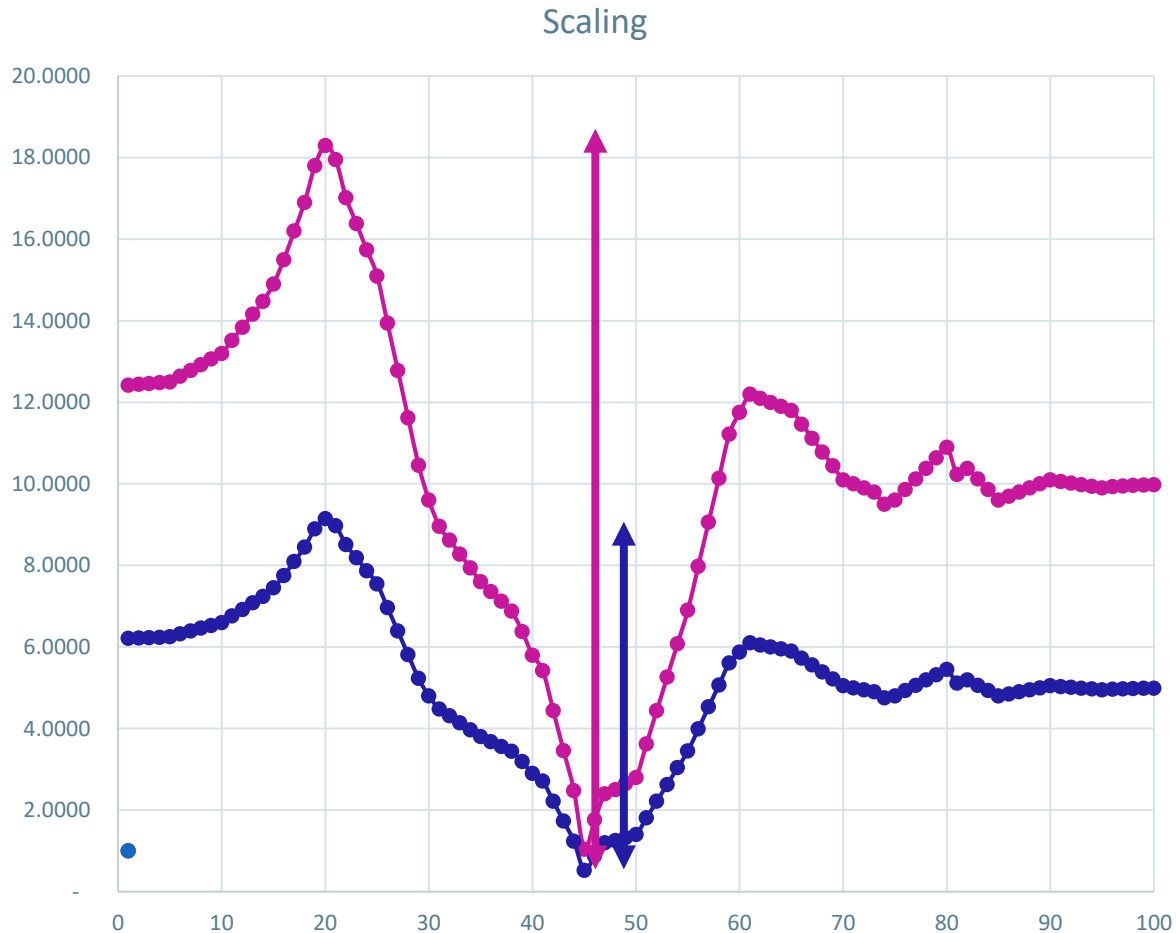
Euclidean distance with *offset translation* pre-processing



```
select
  sqrt(sum(power(val-val2,2)))
from (
  select
    val-avg(val) over() as val,
    val2-avg(val2) over() as val2
  from
    tsample
) a;
```

We can manage *offset translation* by removing the averages from each of the two time-series

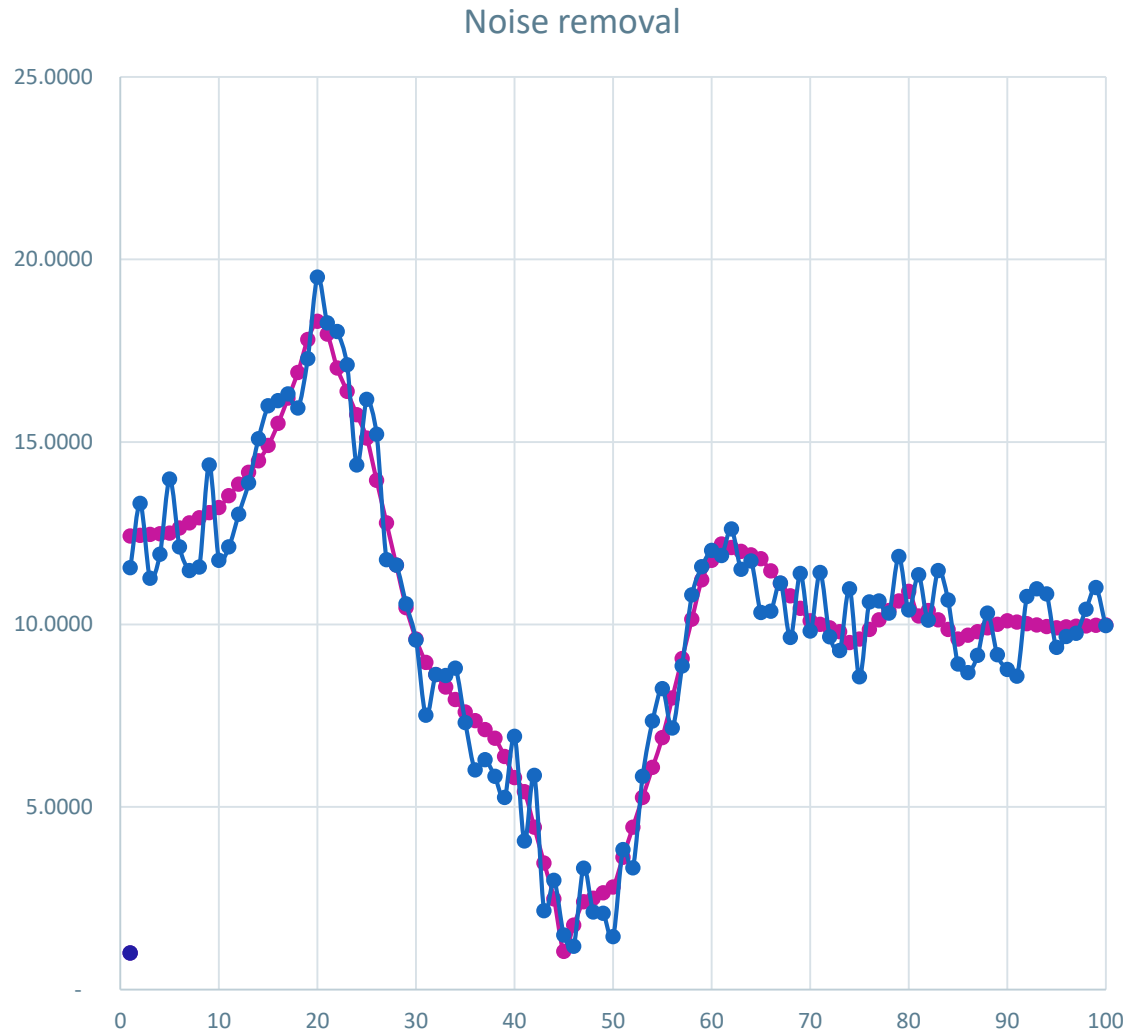
Euclidean distance with *scaling* pre-processing



```
select
    sqrt(sum(power(val-val3,2)))
from (
    select
        (val - avg(val) over()) /
        stddev(val) over() as val,
        (val3-avg(val3) over()) /
        stddev(val3) over() as val3
    from
        tsample
) a;
```

We can manage *different scales* by removing the averages from each of the two time-series and dividing by the respective standard deviations

Euclidean distance with *noise removal* pre-processing



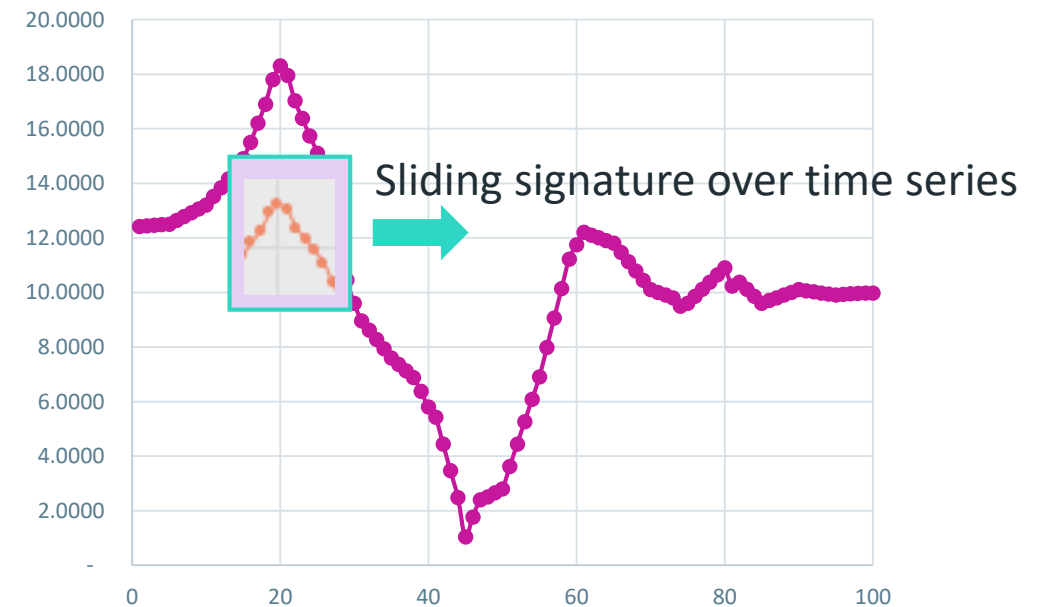
```
select
  sqrt(sum(power(val-val4,2)))
from (
  select
    val - avg(val) over(order by ts
      rows between 4 preceding
      and current row) as val,
    val4 - avg(val4) over(order by ts
      rows between 4 preceding
      and current row) as val4
  from
    tsample
) a;
```

We can “smooth” *noisy time-series* by removing moving averages over a given number of elements

Time Series Pattern Matching

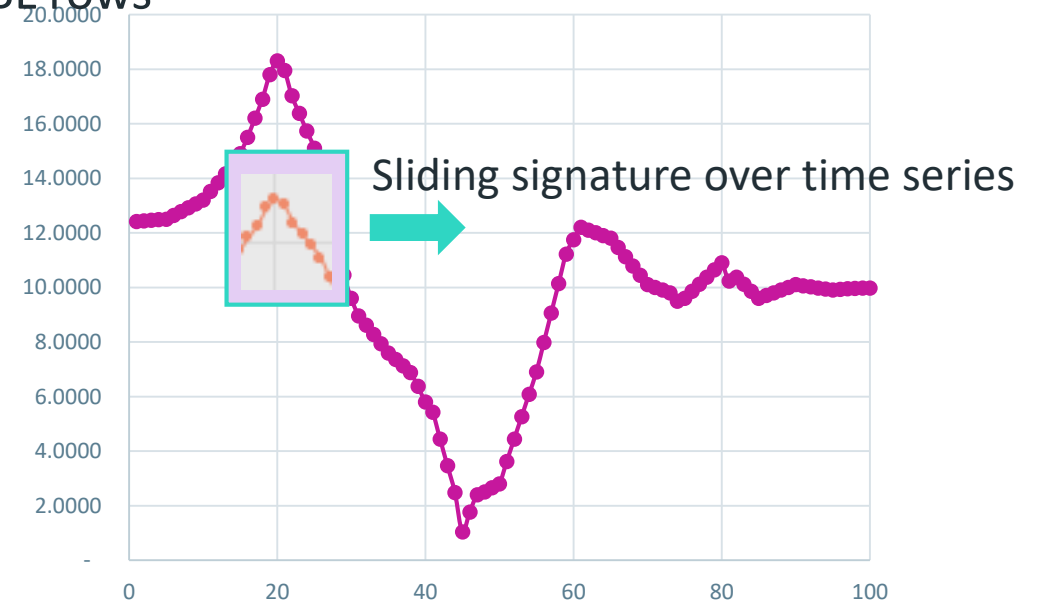
Time Series “pattern matching”

- The scope is to check if Time-Series A shows the same curve as Time-Series B
- Or you might want to check if a given layout we have in the last 30 seconds already happened in the past
- In this case you can measure the Normalized Euclidean Distance between Time Series by “sliding” the shorter Time Series (signature) over the long one:
- This way you will get different (dis)similarity measures at different offsets



Time Series “pattern matching” implementation

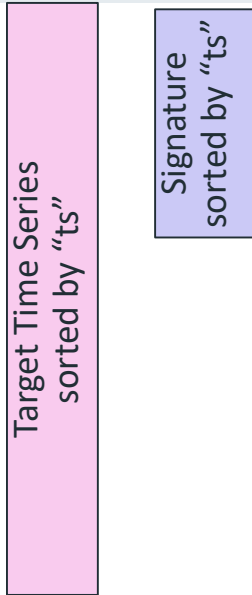
- Time Series Pattern Matching can be implemented in Vertica SQL without coding User Defined Functions
- But this requires a CROSS JOIN between the Target Time Series and the Signature
- If – for example – our target time series contains 1 BL rows and the signature 30 measures we will produce an (intermediate) result set consisting of 30 BL rows
- If we code Time Series pattern Matching as UDx:
 - We will avoid CROSS JOINS
 - We will avoid calling several (SQL) functions for each iteration
 - We can re-use several “partial” results
- In our implementation we will assume that both Signature and Target Time-Series have a very simple structure:
 - Timestamp (column name “**ts**”, data type “**timestamp**”)
 - Value (column name “**val**”, data type “**float**”)



Time Series “pattern matching” UDX arguments

- Our first approach could be to call our UDX (`reud()`) by passing values (sorted by timestamp) from both Signature and Target Time Series:

```
select reud(val1, val2) over() ...;
```



- But if the Target Time Series consists of millions/billions of values and the Signature just a few tens... most of the rows in input to our UDX will have empty Signature values

Time Series “pattern matching” UDX arguments

- So, in order to reduce the number of “round-trips” between Vertica and the UDX when passing data I did use a different approach:

```
select reud(val using parameter siglen=X) over(order by type, ts) ...;
```

Signature
sorted by “ts”

Target Time Series
sorted by “ts”

- We do pass to the UDX only one argument concatenating Signature (first) and Target Time Series
- An Extra Parameter (siglen) will tell the UDX that the first “siglen” values are for the Signature and the rest belongs to the target Time Series
- If you omit the SIGLEN Parameter the UDX will use a default value of 30. In other words will consider Signature the first 30 values of the input data.

Time Series “pattern matching” UDX

Variable Initialization

```
// Variables initialization
```

```
vint ofs = 0 ;  
vint siglen = 30 ;  
vfloat *sig = 0 ;  
vfloat *t1m = 0 ;  
vfloat savg = 0.0 ;  
vfloat ssdev = 0.0 ;  
vfloat tsum = 0.0 ;  
vfloat tavg = 0.0 ;  
vfloat tsdev = 0.0 ;  
vfloat eud = 0.0 ;  
unsigned int i = 0;
```

```
// Offset  
// Default siglen parameter  
// Pointer to sig array  
// Pointer to t1m array  
// Signature elements avg  
// Signature elements stddev  
// Sub-array elements sum  
// Sub-array elements avg  
// Sub-array elements stddev  
// Euclidean Distance  
// Loop variable
```


Time Series “pattern matching” UDX

Parameter Evaluation and Memory Allocation

```
// Parameter evaluation
ParamReader params(srvInterface.getParamReader()); // Get ParameterReader
if (params.containsParameter("siglen")) {           // Check if "siglen" param was used
    siglen = params.getIntRef("siglen");             // Extract "siglen" value
}

// Allocate memory for sig array
if ( ( sig = (vfloat*)malloc((size_t)siglen * sizeof(vfloat)) ) == NULL )
    vt_report_error ( 100 , "Error allocating sig memory" ) ;;

// Allocate memory for tlm array
if ( ( tlm = (vfloat*)malloc((size_t)siglen * sizeof(vfloat)) ) == NULL )
    vt_report_error ( 100 , "Error allocating tlm memory" ) ;;
```

Time Series “pattern matching” UDX

Read Signature and Calculate AVG/STDDEV, Read first “siglen-1” elements from Target Time Serie

```
// First: read "siglen" elements (signature) and store them in sig array
for ( i = 0 ; i < siglen ; i++ , inputReader.next() ) {
    sig[i] = inputReader.getFloatRef(0) ;
    savg += sig[i] ; // Sum signature elements
}
savg /= siglen ; // Signature elements avg

// Calculate Signature standard deviation
for ( i = 0 ; i < siglen ; i++ )
    ssdev += ( sig[i] - savg ) * ( sig[i] - savg ) ;
ssdev = sqrt ( ssdev / siglen ) ;

// Second: read next "siglen-1" elements to initialize tlm sub-array
for ( i = 0 ; i < siglen - 1 ; i++ , inputReader.next() ) {
    tlm[i] = inputReader.getFloatRef(0) ;
    tsum += tlm[i] ; // sub-array sum
}
```

Time Series “pattern matching” UDX

Main Loop (part 1)

```
// Main loop...
do {
    // Read last tlm element from the input block
    tlm[siglen-1] = inputReader.getFloatRef(0) ;

    // Add newer element to sub-array sum
    tsum += tlm[siglen - 1];

    // Calculate Sub Array Avg and Standard Deviation
    tavg = tsum / siglen ;
    for ( i = 0, tsdev = 0 ; i < siglen ; i++)
        tsdev += ( tlm[i] - tavg ) * ( tlm[i] - tavg ) ;
    tsdev = sqrt ( tsdev / siglen ) ;

    // Calculate (normalized) Euclidean Distance Singnature>SubArray
    for ( i = 0, eud = 0 ; i < siglen ; i++)
        eud += ((( sig[i] - savg ) / ssdev) - (( tlm[i] -tavg ) / tsdev)) *
                ((( sig[i] - savg ) / ssdev) - (( tlm[i] -tavg ) / tsdev)) ;
    eud = sqrt(eud);
    ...
}
```

Time Series “pattern matching” UDX

Main Loop (part 2)

```
// Main loop...
do {
    ...
    // Write in output: offset and Euclidean Distance
    outputWriter.setInt(0, ofs) ;
    outputWriter.setFloat(1, eud) ;
    outputWriter.next() ;

    // Remove "older" elements from sub-array's sum
    tsum -= tlm[0] ;

    // Now "left-shift" sub-array to remove the older element
    for ( i = 1 ; i < siglen ; i++ )
        tlm[i-1] = tlm[i] ;

    // And increment the offset...
    ofs++ ;
} while ( inputReader.next() ); // Continue up to the end of the input block

// Free allocated memory
free( tlm ) ;
free( sig ) ;
```

タイムシリーズ ～予測～

タイムシリーズ分析ライブラリ

4つのR-UDx変換関数を含むライブラリ：

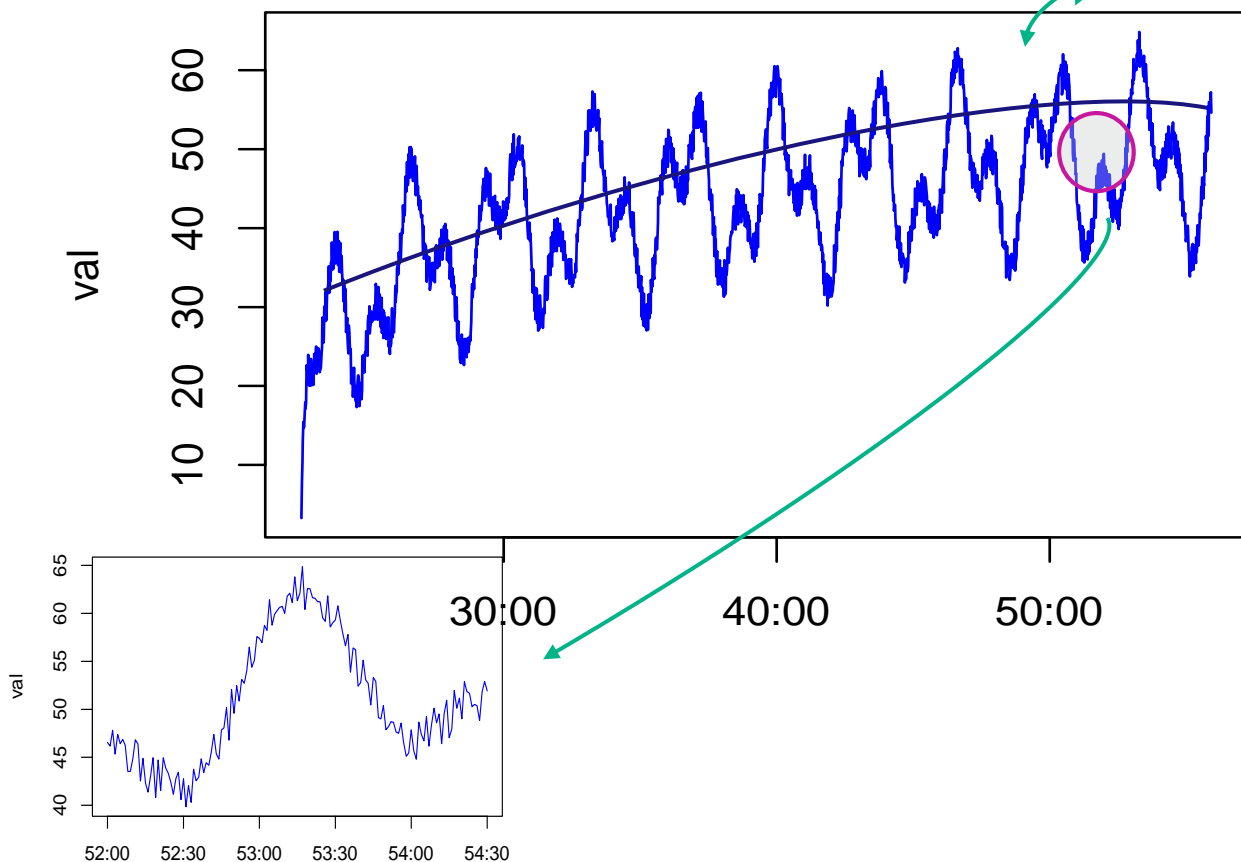
- RACF () : 自動相関を計算
- RCCF () : CrossCorrelationを計算
- RSPECTRUM () : 時系列スペクトル密度を分析
- RFCAST () : 時系列を予測

```
SQL> SELECT * FROM public.sensor ORDER BY ts  
LIMIT 10;
```

ts	val
2013-03-30 08:22:36	3.28284756406089
2013-03-30 08:22:37	7.80055884739127
2013-03-30 08:22:38	10.3598684797151
2013-03-30 08:22:39	12.9092655760947
2013-03-30 08:22:40	15.4543952569975
2013-03-30 08:22:41	14.7525059558571
2013-03-30 08:22:42	14.9996252407873
2013-03-30 08:22:43	16.5931857698105
2013-03-30 08:22:44	17.9132460024764
2013-03-30 08:22:45	17.1374123061181

(10 rows)

データ理解 – すべての対象データ



- 複数の周波数
- 非線形ドリフト
- いくつかの"ホワイトノイズ"

```
library(RODBC)
ch <- odbcConnect("vbox",uid="dbadmin",pwd="xxx",
  rows_at_time=1024)
df <- sqlQuery(ch,"select ts, valfrom
  public.sensor order by 1", as.is=TRUE)
odbcClose(ch)
df$ts <- as.POSIXct(df$ts, tz="UTC")
df$val <- as.numeric(df$val)
plot(df, type='l', col='blue', xlab='')
dev.off()
```

RSPECTRUM() R-UDx –スペクトル密度分析

```
# rspectrum UDx - Maurizio Felici - Version 0.1
rspectrum <- function(x)
{
  library(stats)
  s <- spectrum(x[,1], plot=FALSE)
  data.frame(s$freq, 2/s$freq, s$spec)
}

rspectrumFactory <- function()
{
  list ( name = rspectrum,
         udxtype = c("transform"),
         intype = c("float"),
         outtype = c("float", "float", "float"),
         outnames = c("freq", "period", "spdens")
  )
}
```

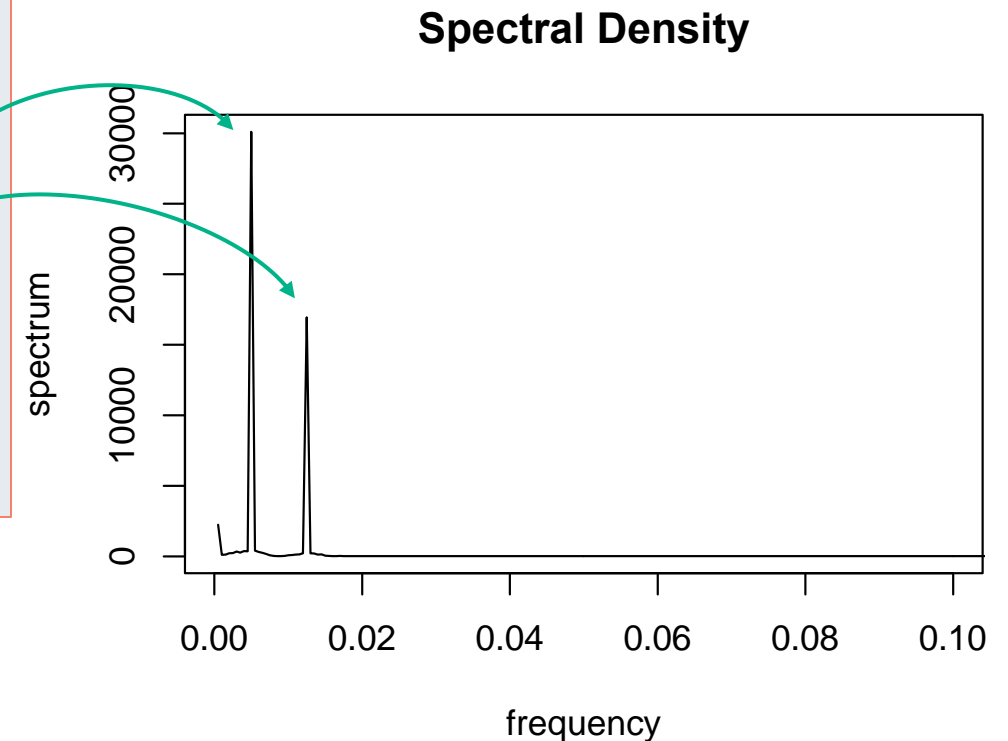

RSPECTRUM() R-UDx –スペクトル密度分析

```
SQL> SELECT rspectrum(val) OVER(ORDER BY ts)
      FROM public.sensor ORDER BY spdens DESC
      LIMIT 5;
```

freq	period	spdens
0.005	400	30110.5660663755
0.0125	160	16941.3377681408
0.0005	4000	2230.3979377785
0.0055	363.636363636364	403.4045581298
0.004	500	370.6086229655

(5 rows)

- "freq" is the frequency (FFT) in $-\text{freq}(x)/2 > \text{freq}(x)/2$
- "period" is the number of sampling intervals
- "spdens" is the spectral density



RACF() R-Udx-自動相關分析

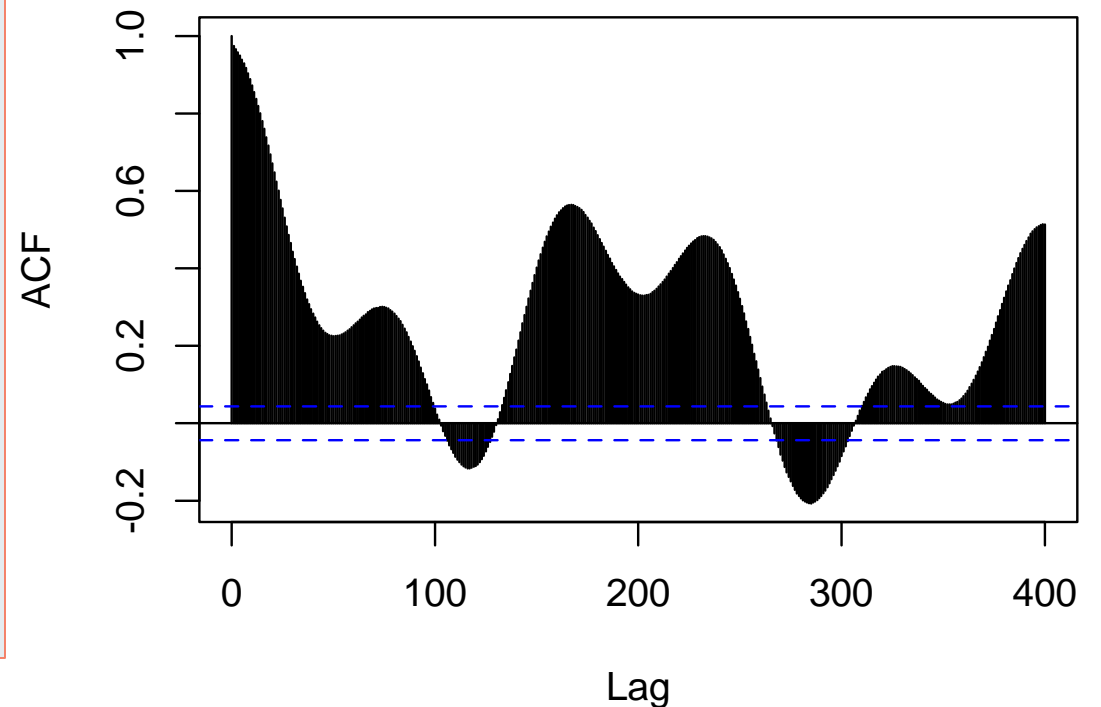
```
# racf Udx - Maurizio Felici - Version 0.1
racf <- function(x, y)
{
  lmax <- if ( !is.null( y[['lag_max']] ) )
    as.numeric(y[['lag_max']]) else 20
  a <- acf(x[,1], lag.max=lmax, plot=FALSE)
  data.frame(a$lag, a$acf)
}
racfParams <- function()
{
  data.frame(
    datatype = c("int"),
    length = c(NA),
    scale = c(NA),
    name = c("lag_max")
  )
}
```

```
racfFactory <- function()
{
  list (
    name = racf,
    udxtype = c("transform"),
    intype = c("float"),
    outtype = c("float", "float"),
    outnames = c("lag", "acf"),
    parametertypecallback = racfParams
  )
}
```

RACF() R-Udx-自動相関分析

```
SQL> SELECT racf(val USING PARAMETERS lag_max=400)
      OVER(ORDER BY ts)
FROM public.sensor ;
```

lag	acf
0	1
1	0.973650388117301
2	0.965459464988071
3	0.957821355197202
4	0.949604726311346
5	0.938949575176876
6	0.929290184908983
7	0.917222393121425
8	0.903501586299938
9	0.888128844918075
10	0.871943181680496
...	



RCCF() R-UDx – 相互相關分析

```
# rccf UDx - Maurizio Felici - Version 0.1
rccf <- function(x, y)
{
  lmax <- if ( !is.null( y[['lag_max']] ) )
    as.numeric(y[['lag_max']]) else 20
  c <- ccf(x[,1], x[,2], lag.max=lmax,
    plot=FALSE)
  data.frame(c$lag, c$acf)
}
rccfParams <- function()
{
  data.frame(
    datatype = c("int"),
    length = c(NA),
    scale = c(NA),
    name = c("lag_max")
  )
}
```

```
rccfFactory <- function()
{
  list (
    name = rccf,
    udxtype = c("transform"),
    intype = c("float", "float"),
    outtype = c("float", "float"),
    outnames = c("lag", "ccf"),
    parametertypecallback = rccfParams
  )
}
```

RFCAST() R-UDx – タイムシリーズ予測

```
# rfcast UDx - Maurizio Felici - Version 0.1
```

```
rfcast <- function(x, y)
{
  library(forecast)
  per <- c()
  nfp <- 20
  if(!is.null(y[['nprev']])) {
    nfp <- as.numeric(y[['nprev']])
  }
  if(!is.null(y[['period']])) {
    per[1] <- as.numeric(y[['period']])
  } else {
    stop("At least one period has to be specified")
  }
  if(!is.null(y[['period2']])) {
    per[2] <- as.numeric(y[['period2']])
  }
  if(!is.null(y[['period3']])) {
    if(is.null(y[['period2']])) {
      stop("You cannot specify period3 if period2 is empty")
    }
    per[3] <- as.numeric(y[['period3']])
  }
}
```

```
t <- msts ( as.numeric ( x[,1] ), seasonal.periods=per )
fcast <- forecast ( t , h=nfp )
df_fcast <- as.data.frame(fcast)
colnames(df_fcast) <-
  c("fcast", "lo_80", "hi_80", "lo_95", "hi_95")
df_fcast[, "no"] <- 1:nrow(df_fcast)
df_fcast[c("no", "fcast", "lo_80", "hi_80", "lo_95", "hi_95")]
}
```

RFCAST() R-UDx –タイムシリーズ予測

```
rfcastParams <- function()
{
  data.frame(
    datatype = c("int", "float", "float", "float"),
    length = c(NA, NA, NA, NA),
    scale = c(NA, NA, NA, NA),
    name = c("nprev", "period", "period2", "period3")
  )
}
rfcastFactory <- function()
{
  list ( name = rfcast,
         udxtype = c("transform"),
         intype = c("float"),
         outtype = c("float", "float", "float", "float", "float", "float"),
         outnames = c("point", "fcast", "lo_80", "hi_80", "lo_95", "hi_95"),
         parametertypecallback = rfcastParams
       )
}
```

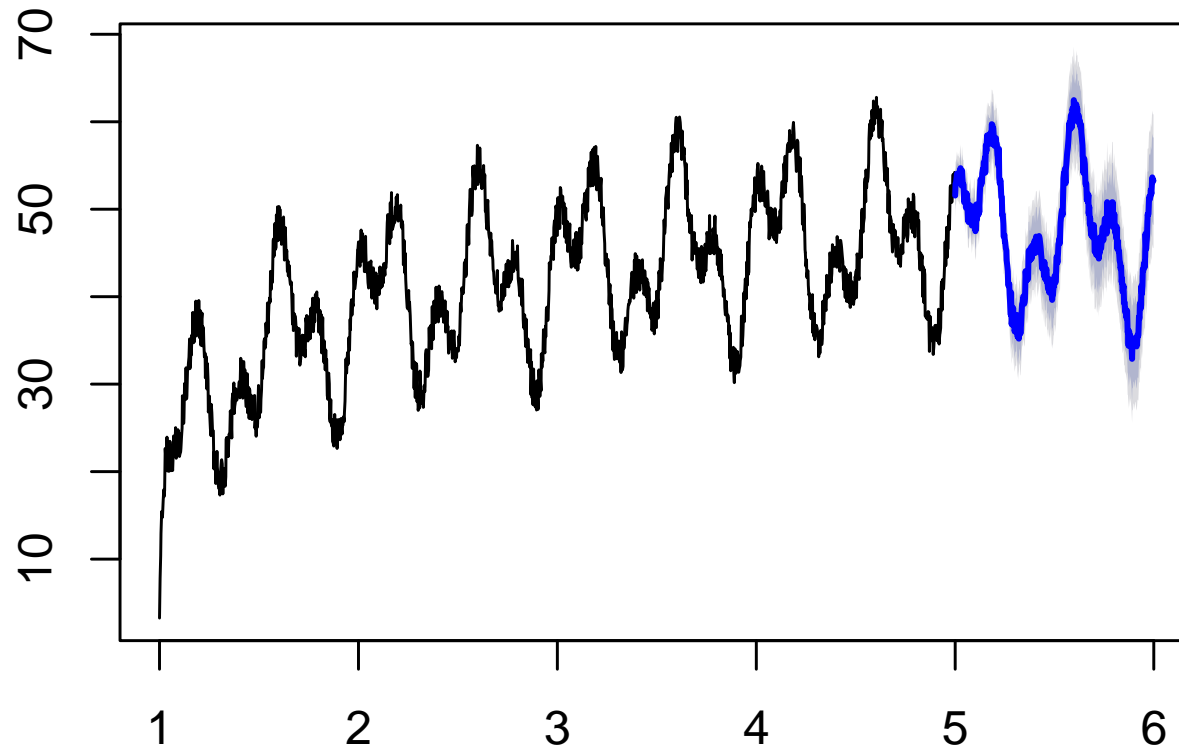
RFCAST() R-UDx –タイムシリーズ予測

```
SELECT rfcast(val USING PARAMETERS nprev=400, period=400, period2=160) OVER(ORDER BY ts)
FROM public.sensor WHERE ts <= '2013-03-30 08:49:15' ; -- to select only 1600 samples out of 2000
```

point	fcast	lo_80	hi_80	lo_95	hi_95
1	51.5458565922203	49.9518808855036	53.139832298937	49.1080811164681	53.9836320679725
2	52.8738790440734	51.2652421062393	54.4825159819075	50.4136811502682	55.3340769378787
3	53.0892758636234	51.4659536555276	54.7125980717192	50.6066187870665	55.5719329401803
4	53.7812554899177	52.1432323223909	55.4192786574446	51.2761152360954	56.2863957437401
5	54.0741319852887	52.4213998863869	55.7268640841904	51.5464963621648	56.6017676084126
6	53.4186154540043	51.7511735732902	55.0860573347184	50.8684831609471	55.9687477470615
7	53.8304603691585	52.1483144178146	55.5126063205024	51.2578401406678	56.4030805976492
8	53.5085825031842	51.811744228631	55.2054207777374	50.9134923053894	56.103672700979
9	53.3553008233611	51.6437875169628	55.0668141297593	50.7377671011217	55.9728345456005
10	53.1344312856261	51.4082653220092	54.8605972492431	50.4944882579803	55.774374313272
11	53.9931136174406	52.2523220239204	55.7339052109608	51.3308026191002	56.655424615781
12	54.6905029936277	52.9351170482306	56.4458889390248	52.0058718602062	57.3751341270491
13	52.9127870014836	51.1428418580599	54.6827321449073	50.2058894961554	55.6196845068118
14	52.7177023197074	50.9332366586536	54.5021679807612	49.9885975990342	55.4468070403806
15	54.4207249616688	52.6217806648129	56.2196692585247	51.669477078379	57.1719728449586
16	53.166946829243	51.3535686774865	54.9803249809995	50.3936242698155	55.9402693886706
17	52.840316254953	51.0125516473137	54.6680808625923	50.0449915099299	55.6356409999762

RFCAST() R-UDx -タイムシリーズ予測

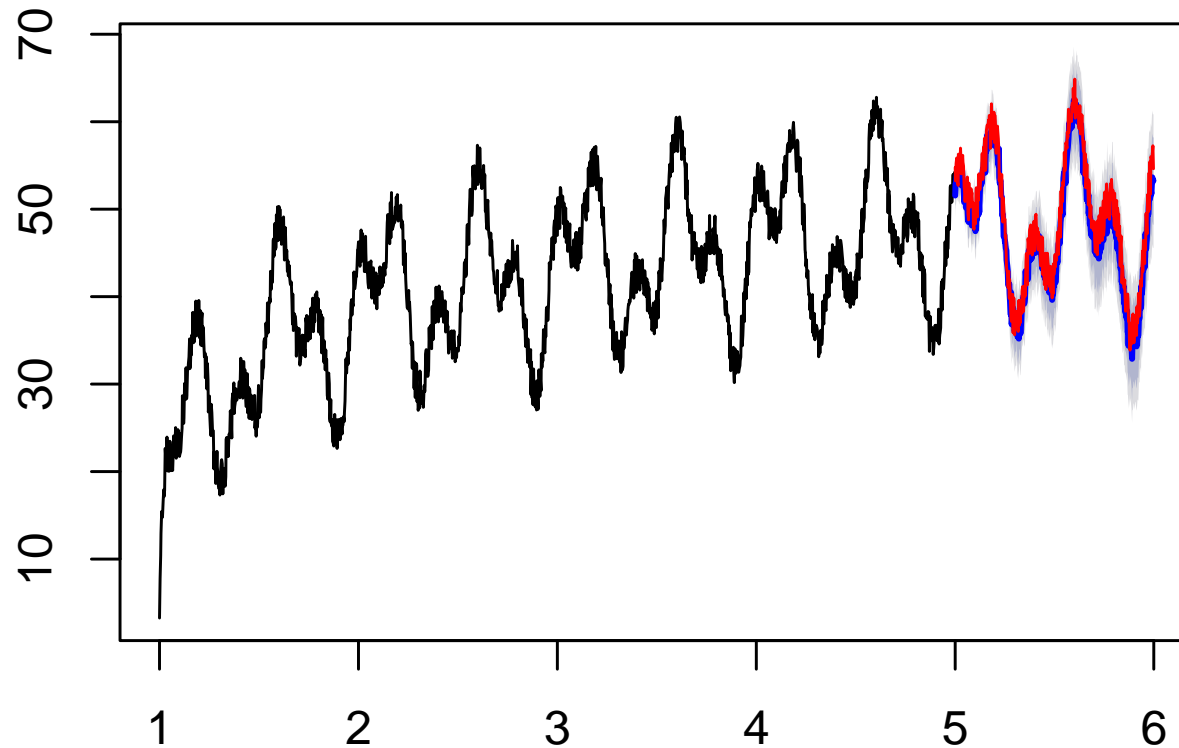
Forecasts from STL + ETS(A,Ad,N)



- 黒い線：予測に利用した（最初の）1600サンプルを表します
- 青い線：（400）の予測値
- 予測値の周囲の濃い灰色の領域： $\pm 80\%$ の信頼区間です
- 予測値の周囲のライトグレー領域： $\pm 95\%$ 信頼区間です

RFCAST() R-UDx -タイムシリーズ予測

Forecasts from STL + ETS(A,Ad,N)



- 赤い線：実際の（最後の400）データ点を表します

並列処理に関する補足情報

並列処理

- 並列性はホットな話題です（MPPデータベース、Map-Reduce、...）
- 我々は2つの重要な特性を特定することができます
 - パラレルデータの配布：異なる「ノード」上で並列にデータを格納および取得する機能
 - パラレル・データ処理：異なる「ノード」上で並列に実行される異なるタスクにデータ精緻化を分割する能力
- パラレルデータ処理は次の要素に依存します。
 - アルゴリズムを「断片」に分割して並列に実行し、結果を「マージ」できる数学的な可能性
 - 上記の条件が満たすとき「マルチスレッド」アルゴリズムをコーディングする能力
- Rコードは「本質的に」マルチスレッドではありません。標準のRコードでは、基本的に2つの制限があります
 - 単一のノード上で単一のコアを使用します
 - すべてのデータオブジェクトは、使用可能なメモリに収まる必要があります
- これらの問題を回避しようとするパッケージ（「並列」または「ビッグメモリ」）があります

並列データ処理と結合法則

- 異なるノード間でデータを配布するのは比較的簡単ですが...
- 分散処理はプロセスの性質に依存します。
- 処理を分散させるための重要ポイントとして、
 - 数学的特性
 - 結合性
- 集合S上にて、汎用バイナリ処理は、‘*’で結合的な関係になります。

$$(x * y) * z = x * (y * z) \text{ for all } x, y, z \text{ in } S$$

並列データ処理と結合法則

- 多くの“結合的”操作があります。例えば：
 - ベクトルの合計要素（データベース用語ではsum（））
 - ベクトルの要素を数える（データベース用語でcount（））
 - 特定の制約を満たす2つのベクトルの要素を数える（データベースの項で結合する）
 - 等...
- 上記は、実行順序に関係なく、異なる（並列の）タスクの結果を「結合」することができるため、計算上の用語では「並列化可能」です
- しかし、いくつかの「非連想」操作もあります。例えば：
 - 以前に計算された値に依存する計算は以下のようになります。

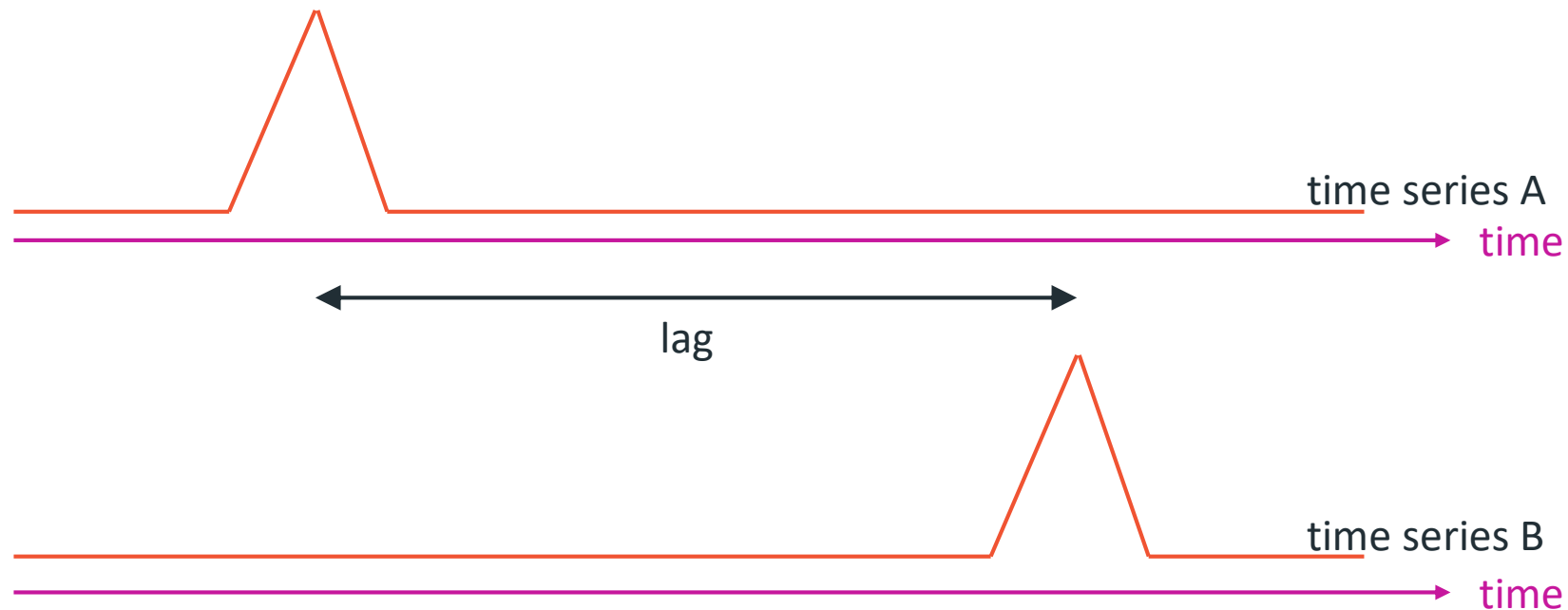
$$f(k+2) = f(k+1) + f(k)$$

- 自動または相互相関（時系列の因果関係を決定するために非常に重要）

$$(f \star g)[n] \stackrel{\text{def}}{=} \sum_{m=-\infty}^{\infty} f^*[m] g[m+n].$$

並列データ処理と結合法則

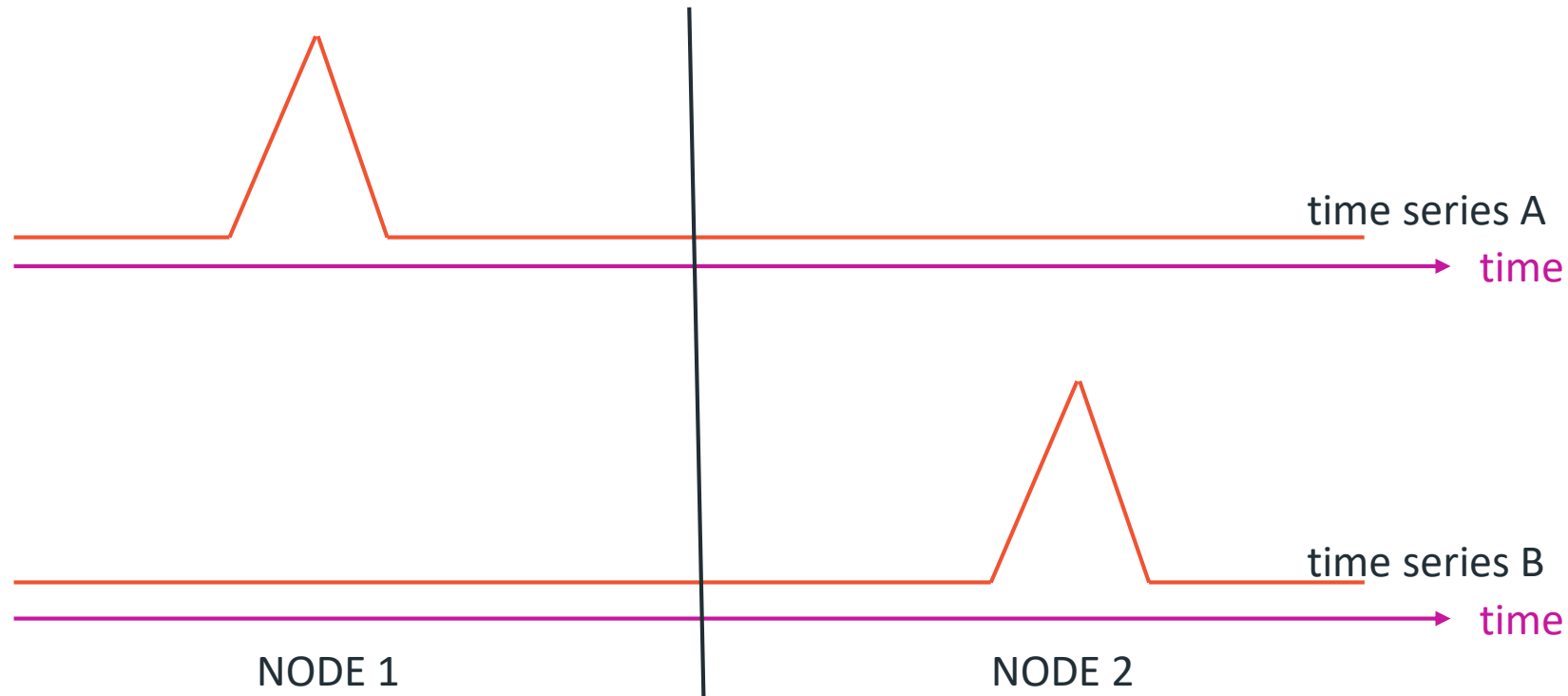
- 相互相関の非関連性を「可視化」します。次のような2つの時系列があるとしています



- 明らかに一定期間の遅れと相互相関しています

並列データ処理と結合法則

- ここで、時系列AおよびBのデータが2つのノードにこのように分散されているとします



- 2つのノードにローカルなデータに対して別々に相互相関を計算し、結果を意味のある方法で再結合する方法はありません



VERTICA

Thank you.

www.vertica.com