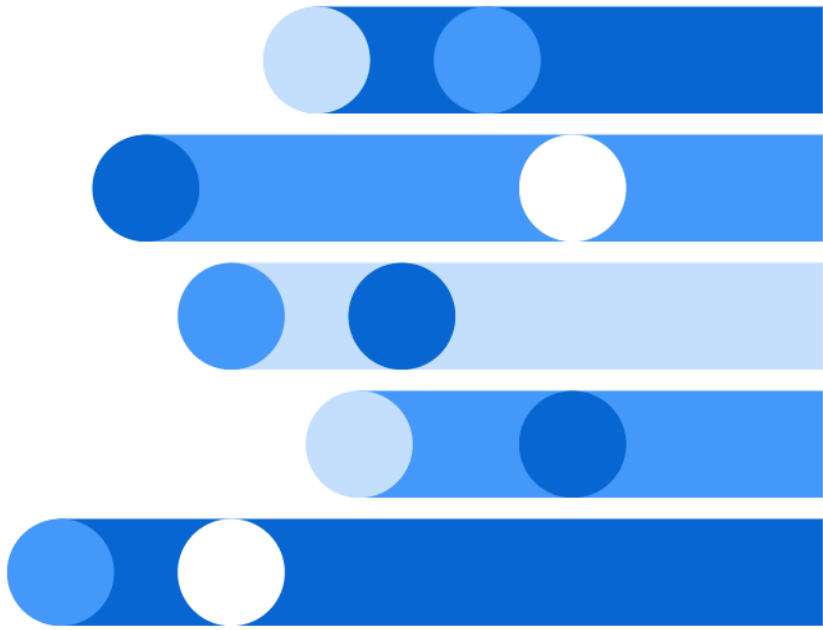
SAS® Event Stream Processing: ソースおよび派生ウィンドウの使用

2026.08*



* このドキュメントは、ソフトウェアの追加バージョンに適用される場合があります。このドキュメントを SAS Help Center で開き、バナーのバージョンをクリックすると、使用できるすべてのバージョンが表示されます。

SAS® ドキュメント
2026 年 9 月 18 日

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2026. *SAS® Event Stream Processing: ソースおよび派生ウィンドウの使用*. Cary, NC: SAS Institute Inc.

SAS® Event Stream Processing: ソースおよび派生ウィンドウの使用

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

August 2026

v_035-P1:espcreatewindows

目次

1 部

ソースおよび派生ウィンドウ

1

概要

ウィンドウタイプ

イベントストリーム処理プロジェクトは、入力イベントストリームがどのように変換され、意味のあるイベントストリームに分析されるかを指定します。各プロジェクトは、1 つまたは複数の連続クエリを保有します。

連続クエリには、1 つ以上のソースウィンドウと 1 つ以上の*派生ウィンドウ*が含まれます。すべてのイベントストリームをパブリッシュするかソースウィンドウにインジェクトして、連続クエリを入力する必要があります。イベントストリームは、他のウィンドウタイプにパブリッシュまたはインジェクトすることはできません。

Windows は*エッジ*によって接続されており、 それは関連する方向を持っています。

SAS Event Stream Processing は、それぞれ特殊な目的を持ったさまざまな派生ウィンドウタイプをサポートしています。

表 1.1　派生ウィンドウタイプ

ウィンドウタイプ	説明
計算ウィンドウ	入力イベントストリームフィールドの計算操作を通じて入力イベントの出力イベントへの 1 対 1 変換を可能にします。
集計ウィンドウ	キー以外のフィールドが計算される点で計算ウィンドウに似ています。集計ウィンドウは、グループごとの条件のキーフィールドを使用します。すべての一意キーフィールドの組み合わせが、集計ウィンドウ内で独自のグループを形成します。同じキーの組み合わせを持つすべてのイベントは、同じグループの一部です。

ウィンドウタイプ	説明
算出ウィンドウ	さまざまなアルゴリズムを使用してデータイベントを変換します。
コピーウィンドウ	親ウィンドウのコピーを作成します。コピーを作成すると、新しいイベント状態保持ポリシーを設定することができます。保持ポリシーは、ソースウィンドウおよびコピーウィンドウでのみ設定できます。 コピーウィンドウのイベント状態の保持は、ウィンドウが挿入専用ではなく、ウィンドウインデックスが pi_EMPTY に設定されていない場合にのみ設定できます。その後のすべての関連ウィンドウは、保持管理の影響を受けます。イベントは、ウィンドウ保持ポリシーを超えると削除されます。
カウンタウィンドウ	モデルを通ってストリーミングされているイベントの数とそれらが処理されているレートを確認できます。
フィルタウィンドウ	登録されたブールフィルタ関数または式を使用します。この関数または式は、フィルタウィンドウにどの入力イベントが許可されるかを決定します。
関数ウィンドウ	さまざまな種類の関数を使用してイベントデータを操作または変換できます。関数ウィンドウ内のフィールドは階層構造にすることができ、Web 分析などの応用に役立ちます。
ジオフェンスウィンドウ	ウィンドウを作成して、イベントストリームの場所が関心領域の内側にあるのか、関心領域に近いのかを判断できます。
結合ウィンドウ	2 つの入力ウィンドウと結合タイプをとります。1 対多、多対 1、多対多の等価結合をサポートします。内部結合と外部結合の両方がサポートされています。
Lua ウィンドウ	Lua プログラムで SAS Event Stream Processing イベントを生成することができます。
モデルリーダーウィンドウ	分析ストアファイルまたは非バイナリファイルの組み合わせとして、SAS Event Stream Processing に導入されたモデルを読み取ります。
モデルスーパーバイザウィンドウ	モデルリーダーウィンドウから受信したモデルを管理します。
通知ウィンドウ	メール、ショートメッセージまたはマルチメディアメッセージを 使用して通知が送信できます。任意の数の配布チャネルを作成して通知を送信することができます。通知ウィンドウは、関数ウィンドウと同じ基本言語と関数を使用します。

ウィンドウタイプ	説明
オブジェクトトラッキングウィンドウ	リアルタイムでマルチオブジェクトトラッキング(MOT)を実行できます。
パターンウィンドウ	関心があるイベント(EOI)の検出を有効にします。このウィンドウタイプで定義されたパターンは、宣言された関心があるイベントを論理的に接続する式です。 パターンウィンドウを定義するには、関心があるイベントを定義し、次にこれらの関心があるイベントを演算子を使用して接続する必要があります。サポートされる演算子は、"AND"、"OR"、"FBY"、"NOT"、"NOTOCCUR"、"IS"です。演算子はオプションの一時的条件を受け入れることができます。
プロシジャウィンドウ	任意の数の入力ウィンドウや各入力ウィンドウの入力ハンドラ関数の指定(すなわち、イベントストリーム)を可能にします。
Python ウィンドウ	Python プログラムで SAS Event Stream Processing イベントを生成することができます。
状態の削除ウィンドウ	モデルのステートフル部分からモデルのステートレス部分への移行を容易にします。
スコアウィンドウ	SAS Event Stream Processing とパッケージ化されたオンライン分析アルゴリズムまたはオフラインモデルのアルゴリズムでイベントをスコアリングします。
StateDB リーダー	受信イベントと指定された基準に基づいて、定義された外部データベースからデータをクエリまたはフェッチします。これらの基準の一部は、StateDB ライターウィンドウで作成したインデックスに基づいています。
StateDB ライター	受信イベントを定義済みの外部データベースに書き込み、イベントに必要なインデックスを作成し、入力イベントを変更せずにプロジェクトの次のウィンドウに渡します。
学習ウィンドウ	ストリーミングイベントデータに基づいてオンラインモデルのアルゴリズムパラメーターを調整します。
転置ウィンドウ	イベントの行を列として、または列を行として交換できます。
和結合ウィンドウ	SQL 和結合(Union)操作のように、同じスキーマを持つ複数のイベントストリームを単一のストリームに結合します。

算出ウィンドウ、モデルリーダーウィンドウ、モデルスーパーバイザーウィンドウ、スコアウィンドウ、学習ウィンドウについては、*SAS Event Stream Processing: ストリーミング分析の適用*を参照してください。

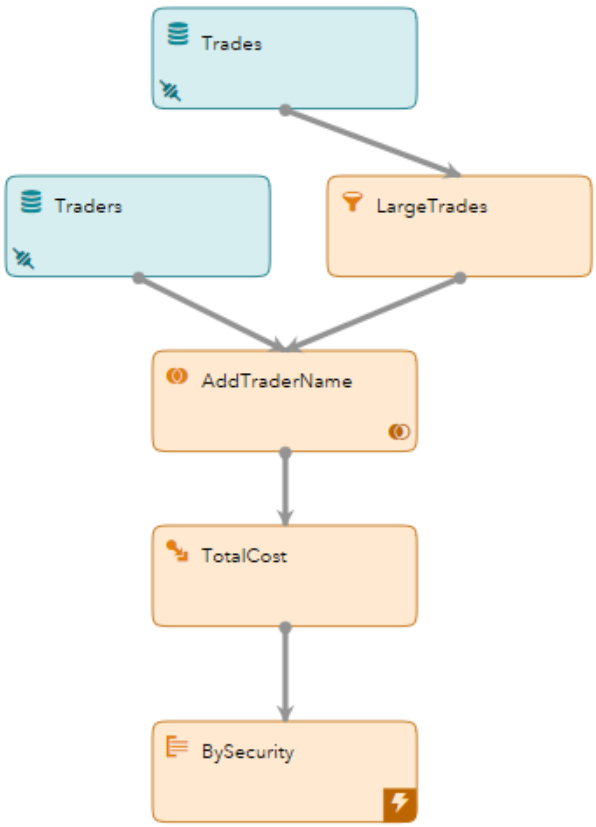
ウィンドウとエッジによる連続クエリの作成

株取引を処理するために連続クエリを作成したいとします。そのクエリに対して、次の操作を実行します。

- 2 つのイベントストリームをクエリにパブリッシュします。1 つは取引用、もう 1 つは対応するトレーダー用です。
- 100 株未満の取引を除外します。
- すべての株が取引された後の総取引費用を計算します。
- セキュリティ別にまとめられた総費用を示すファイルを作成します。

SAS Event Stream Processing Studio でプロジェクトを作成できます。

図 1.1　連続クエリダイアグラム



これは対応する SAS Event Stream Processing XML モデリング言語です。このコードはサポート Web サイトからダウンロードできます。**/FurtherExamples/trades** に移動して、XML ファイルとサポート CSV ファイルを取得します。

```
<project name='trades_project' pubsub='auto' threads='4'> <!-- 1 -->
        <contqueries>
          <contquery name='trades_cq'> <!-- 2 -->
            <windows>
```

1 auto のパブリッシュ /サブスクライブモードの trades_proj という名前のプロジェクトが確立されます。使用可能なスレッドプールから 4 つのスレッドが使用されます。

2 trades_cq という名前の連続クエリが確立されます。

```
<window-source name='Trades' insert-only='true' index='pi_EMPTY'> <!-- 1 -->
    <schema>
      <fields>
        <field name='tradeID' type='string' key='true'/>
        <field name='security' type='string'/>
        <field name='quantity' type='int32'/>
        <field name='price' type='double'/>
        <field name='traderID' type='int64'/>
        <field name='time' type='stamp'/>
      </fields>
    </schema>
    <connectors>
       <connector class="fs" name="publisher">  <!-- 2 -->
        <properties>
         <property name="type">pub</property>
         <property name="fstype">csv</property>
         <property name="fsname">trades.csv</property>
         <property name="transactional">true</property>
         <property name="blocksize">1</property>
         <property name="dateformat">%d/%b/%Y:%H:%M:%S</property>
        </properties>
       </connector>
     </connectors>
</window-source>

<window-source name='Traders' insert-only='true' index="pi_EMPTY">  <!-- 3 -->
    <schema>
      <fields>
        <field name='ID' type='int64' key='true'/>
        <field name='name' type='string'/>
      </fields>
    </schema>
   <connectors>
       <connector class="fs" name="publisher">  <!-- 4 -->
        <properties>
         <property name="type">pub</property>
         <property name="fstype">csv</property>
         <property name="fsname">traders.csv</property>
         <property name="transactional">true</property>
         <property name="blocksize">1</property>
        </properties>
       </connector>
     </connectors>
</window-source>
```

1. Trades という名前のソースウィンドウが pi_EMPTY インデックスタイプで確立されます。このウィンドウは、CSV ファイルの証券トランザクションに関するデータをストリームします。
2. ファイルおよびソケットコネクタは、trades.csv という名前の現在の作業ディレクトリ内のファイルから Trades ウィンドウにイベントをパブリッシュするよう確立されています。
3. トレーダーという名前のソースウィンドウが確立されます。このウィンドウは、そのトランザクションを実行するユーザーに関するデータをストリームします。データは、ファイル、データベース、またはその他のソースからパブリッシュすることができます。
4. ファイルおよびソケットコネクタは、traders.csv という名前の現在の作業ディレクトリ内のファイルから Traders ウィンドウにイベントをパブリッシュするよう確立されています。

LargeTrades という名前のフィルターウィンドウが確立され、トレードウィンドウからイベントを受け取ります。これは、100 未満の株式を含むすべてのイベントを除外します。

```
<window-filter name='LargeTrades' index="pi_EMPTY">
    <expression>quantity >= 100</expression>
</window-filter>
```

AddTraderName という結合ウィンドウは、2 つのソースウィンドウからの値を使用して結合操作を実行します。これは、フィルタリングされたトランザクションを、関連するトレーダと照合します。

```
<window-join name='AddTraderName' index="pi_EMPTY">
    <join type="leftouter" no-regenerate="true">
      <conditions>
        <fields left='traderID' right='ID' />
      </conditions>
    </join>
    <output>
      <field-selection name='security' source='l_security'/>
      <field-selection name='quantity' source='l_quantity'/>
      <field-selection name='price' source='l_price'/>
      <field-selection name='traderID' source='l_traderID'/>
      <field-selection name='time' source='l_time'/>
      <field-selection name='name' source='r_name'/>
    </output>
</window-join>
```

TotalCost という算出ウィンドウは、結合ウィンドウからのデータを使用してトランザクションのコストを算出します。

```
<window-compute name='TotalCost' index='pi_EMPTY'>
    <schema>
      <fields>
        <field name='tradeID' type='string' key='true'/>
        <field name='security' type='string'/> <!-- 1 -->
        <field name='quantity' type='int32'/>
        <field name='price' type='double'/>
        <field name='totalCost' type='double' />
        <field name='traderID' type='int64'/>
        <field name='time' type='stamp'/>
        <field name='name' type='string' />
```

```
        </fields>
      </schema>
      <output>
        <field-expr>security</field-expr>
        <field-expr>quantity</field-expr>
        <field-expr>price</field-expr>
        <field-expr>price*quantity</field-expr> <!-- 2 -->
        <field-expr>traderID</field-expr>
        <field-expr>time</field-expr>
        <field-expr>name</field-expr>
      </output>
  </window-compute>
```

1 このフィールドと次のフィールドはキー以外のフィールドです。

2 これが totalCost の計算方法です。

BySecurity という名前の集計ウィンドウが確立されます。ESP_aSum 関数は、総数とコストを合計するために使用されます。サブスクライバーコネクタは、result.out という名前の CSV ファイルに出力をパブリッシュします。

```
  <window-aggregate name='BySecurity'>
      <schema>
        <fields>
          <field name='security' type='string' key='true'/>
          <field name='quantityTotal' type='double'/>
          <field name='costTotal' type='double'/>
        </fields>
      </schema>
      <output>
        <field-expr>ESP_aSum(quantity)</field-expr>
        <field-expr>ESP_aSum(totalCost)</field-expr>
      </output>
      <connectors>
        <connector class="fs" name="sub">
          <properties>
            <property name="type">sub</property>
            <property name="fstype">csv</property>
            <property name="header">true</property>
            <property name="fsname">result.csv</property>
            <property name="snapshot">true</property>
          </properties>
        </connector>
      </connectors>
  </window-aggregate>
  </windows>
```

エッジがウィンドウを接続します。

```
  <edges>
        <edge source='LargeTrades' target='AddTraderName'/> <!-- 1 -->
        <edge source='Traders' target='AddTraderName'/>
        <edge source='Trades' target='LargeTrades'/> <!-- 2 -->
        <edge source='AddTraderName' target='TotalCost'/>
        <edge source='TotalCost' target='BySecurity'/>
      </edges>
    </contquery>
  </contqueries>
  </project>
```

1　フィルターウィンドウとトレーダーのソースウィンドウでは、イベントを結合ウィンドウに表示します。

2　トレードウィンドウは、イベントをフィルターウィンドウに表示します。結合ウィンドウは、イベントを計算ウィンドウに移動します。そこから集計ウィンドウにイベントが流れます。

出力 CSV ファイルは、2 つの保持されたイベントを示します。1 つ目は、すべての取引が取引された後、IBM の 1000 株が総額 100,100.00 ドルで売却されたことを示しています。2 番目は、SAP の 750 株が総額 25,650.00 ドルで売却されたことを示しています。

図 1.2　集計ウィンドウからの出力

security	quantityTotal	costTotal		
I	N	ibm	1000	100100
I	N	sap	750	25650
UB	N	ibm	2000	200300
D	N	ibm	1000	100100
UB	N	ibm	3000	300600
D	N	ibm	2000	200300
UB	N	ibm	4000	401000
D	N	ibm	3000	300600
UB	N	sap	1750	59950
D	N	sap	750	25650
UB	N	ibm	5000	501300
D	N	ibm	4000	401000
UB	N	sap	2750	91950
D	N	sap	1750	59950
UB	N	ibm	6000	601300
D	N	ibm	5000	501300

2

入力ストリーム

ソースウィンドウの概要

各連続クエリにはソースウィンドウが 1 つ以上必要です。すべてのイベントストリームは、パブリッシュされたり、ソースウィンドウにインジェクトされたりして、連続クエリを入力します。イベントストリームは、他のウィンドウタイプにパブリッシュまたはインジェクトすることはできません。

ソースウィンドウは、通常、1 つまたは複数の派生ウィンドウに接続されます。派生ウィンドウは、データ内のパターンを検出したり、データを変換したり、データを集計したり、データを分析したり、データに基づいて計算を実行したりすることができます。

ソースウィンドウは、インプロセスコネクタまたは実行可能アダプターを介してストリーミングデータまたは生データファイルを受け入れます。ソースウィンドウでは、パブリッシュ/サブスクライブ API、HTTP クライアント、または C++モデリング API からのインジェクションによってデータを受け取ることもできます。

通常、ソースウィンドウは、下流の派生ウィンドウが自由に処理できるようになるまで、受信イベントをキューに入れます。イベントブロックは決して削除されないため、ダウンストリーム処理が重要な場合、またはリソースが制限されている場合には、ソースウィンドウに大量のバックアップが存在する可能性があります。queue-height パラメータを使用して、入力キューのサイズを設定できます。受信イベントをすべてキューに入れず、最新のイベントのみを処理すると便利な場合があ

ります。shed-input パラメーターを使用して、ソースウィンドウに利用可能な最新のイベントブロックのみを強制的に処理させ、バックアップを回避します。

ソースウィンドウに関連付けられている XML エレメントについては、"window-source" (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください

イベントインデックスタイプの定義

ソースウィンドウには、プライマリインデックスと保持インデックスがあります。4 つの完全なステートフルインデックスタイプと 1 つのステートレスインデックスタイプがあります。ソースウィンドウのプライマリインデックスタイプは、プロジェクトの残りの部分のパフォーマンスに影響します。

たとえば、ソースウィンドウにインデックスタイプがある場合 pi_RBTREE ウィンドウはステートフルであり、すべての受信イベントを保持します。保持ポリシーなしでソースウィンドウでイベントを保持すると、ソースウィンドウがステートレスである場合と比べて、データが連続クエリに読み込まれるため、実質的に多くのメモリが使用されます。

インデックスタイプの詳細については、"プライマリインデックスと特殊インデックスについて"を参照してください。

ソースウィンドウの保持ポリシー

保持ポリシーは、時間またはイベント数によって受信データのフローを制限するため、モデルをストリーミングするイベントの総数を制御できます。イベントは、ウィンドウの保持ポリシーを超えたときに削除されます。

ステートフルなソースウィンドウとコピーウィンドウに保持ポリシーを定義できます。保持ポリシーを使用するには、ウィンドウを挿入専用として指定することはできず、インデックスタイプを pi_EMPTY として指定することはできません。通常、保持ポリシーが設定されたコピーウィンドウを持つ挿入専用ソースウィンドウに従います。

保持ポリシーの詳細については、"保持の理解"を参照してください。

挿入専用処理の伝播

挿入イベントのみ受け入れるようにソースウィンドウを明示的に設定できます。これにより、イベントストリーム処理を最適化できます。次のいずれかの条件が満たされない限り、挿入専用処理は派生ウィンドウすべてに伝播されます。

- ウィンドウは、挿入専用データを生成しないと判断されます(たとえば、保持のあるウィンドウなど)。
- *ESP サーバー*は、派生ウィンドウが挿入のみを生成するかどうかを判断できません。
  - プロシジャウィンドウまたは算出ウィンドウで algorithm='MAS'が設定されている場合、非挿入イベントを引き起こす可能性があるユーザー作成コードの任意のブロックを実行します。コードが挿入のみを生成することが確実な場合は、属性 produces-only-inserts='true'をウィンドウの XML 仕様へ明示的に追加してください。
  - 関数ウィンドウが、生成されたイベントの演算コードを変更する関数を使用しているとき。これを属性 produces-only-inserts='true | false'で無効にできます。
  - 集計ウィンドウでは多数の更新が生成されます。必ず produces-only-inserts='false'が設定されます。

キー値の自動生成

ソースウィンドウは、受信イベントの識別キーを自動的に生成することができます。キー値を自動的に生成するには、ソースウィンドウが挿入専用で、タイプが INT64 または文字列のキーが 1 つだけ必要です。window-source エレメントの autogen-key 属性を **true** に設定する必要があります。

INT64 キーの場合、生成される値は増分カウント(0, 1, 2, ...)です。文字列キーの場合、生成される値はグローバル一意識別子(GUID)です。

パブリッシャーコネクタ

パブリッシャーコネクタは、ソースウィンドウで一意です。次が決定されます。

- データソースのパス

- データソースから受け取ったイベントのデータ型
- 受信データをプロジェクト全体で使用されるイベントに変換することに関連するその他の重要なプロパティ

ソースウィンドウのパブリッシャーコネクタは、データの読み込み方法と、イベントが派生ウィンドウにプッシュされる形式を決定します。XML コードでは、コネクタの必須およびオプションのプロパティは、コネクタエレメントで定義されています。

ソースウィンドウのスキーマで定義されたフィールドは、プロジェクトに読み込まれ、派生ウィンドウで使用されるデータの構造を決定します。

測定ウィンドウの有効化

測定ウィンドウは、モデル内のすべてのソースウィンドウで 処理されるイベントの数(および関連するタイムスタンプ)をトラッキングします。この関数を有効にするには、enableMetaProject フィールドの値を **true** に設定する必要があります。**_meta_**という名前のプロジェクトに含まれている**_meta_**という名前の連続クエリに、測定ウィンドウが設定されます。

ソースウィンドウは、デフォルト間隔 5 秒でイベントを測定ウィンドウにインジェクトします。サブスクライブするアプリケーションは、他のウィンドウと同様に測定ウィンドウに登録することができます。

測定ウィンドウは、それ自体、**_eventmetering_**という名前の特別なソースウィンドウです。これには、次のスキーマがあります。

"project*:string,query*:string,window*:string,currenttime:stamp,lasttime:stamp,numevents:int64"

測定サーバーを起動すると、測定ウィンドウのスキーマの最後に文字列フィールドを追加できます。その後、測定ウィンドウによって生成されたすべての測定イベントは、それらのフィールドを含みます。これらのフィールドの値は、いつでも定義または再定義できます。始動時に定義されていない場合は、値を定義するまで値がヌルになります。デフォルトの測定間隔を 5 秒間に再設定することもできます。

測定サーバーを実行し、meteringhost および meteringport パラメーターが設定されると、測定ウィンドウへの各更新は、集計のために測定サーバーに転送されます。

シングルトンの使用

連続クエリ内で定義されているがエッジエレメントには含まれていないソースウィンドウは、シングルトンと呼ばれます。シングルトンは、外部と *ESP サーバー*間の接続を検証し、プロジェクトを設計するときに入力形式を検証するのに役立ちます。

たとえば、プロジェクトへの主な入力が JSON 形式のイベントをストリーミングするメッセージバスであるとします。形式のアイデアはありますが、実際の内容につ

いては確信がありません。この場合、パブリッシャーコネクタを使用してシングルトンを設計して接続をテストし、JSON 渡しパラメーターを調べ、受信データを調べることができます。テストモードで SAS Event Stream Processing Studio を使用し、データの流れを調べることができます。

その後、詳細に検査するためにイベントのスナップショットを静的データセットに収集します。パブリッシャーコネクタをシングルトンに追加し、指定したファイルにデータを保存できます。

連続クエリにシングルトンを含めるかどうかは、contquery エレメントの include-singletons 属性を使用して指定します。詳細については、“contquery” (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

3

変換

フィルターウィンドウの使用

概要

フィルターウィンドウは、Lua コードでコード、Python コード、式エンジン言語(EEL)の式、ユーザー定義関数(グローバル関数)、および登録されたプラグイン関数を使用して、どの入力イベントがプロジェクトを通過できるかを決定します。これらの関数と式は *フィルター条件*と呼ばれます。フィルター条件が True の場合、入力イベントは通過します。フィルター条件が false の場合、イベントはブロックされます。

*ESP サーバー*は、code 要素をフィルターウィンドウで検出すると、式に基づくフィルターウィンドウではなく、フィルター条件のコードを評価します。

フィルターウィンドウで Lua を使用するときに使用できる追加の Lua 機能については、8 章, “共通の Lua 機能”を参照してください。

フィルターウィンドウで Python を使用するときに使用できる追加の Python 機能については、9 章, “一般的な Python 機能”を参照してください

式エンジン言語を使用するフィルター条件の作成の詳細については、“式の概要”を参照してください。

“window-filter” (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)フィルターウィンドウに関連付けられている XML エレメントについては、および“計算およびフィルタウィンドウに関連する XML 言語要素” (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

フィルターウィンドウでの Lua の使用

Lua コードブロックでは、ブール値を返すフィルター関数を指定する必要があります。フィルター関数の名前は filter です。window-filter 要素の func 属性を使用して、フィルター関数の別名を指定できます。*ESP サーバー*は、入力イベントごとにのフィルター関数を呼び出し、戻りコードを使用して出力イベントを生成するかどうかを決定します。

たとえば、次の XML コードは、**count** 入力フィールドの値が 100 以上の場合は常に、すべてのイベントの通過を許可します。

```
<window-filter name="filter_w">
  <code><![CDATA[
    function filter(event)
      return event.count >= 100
```

```
    end
  ]]></code>
</window-filter>
```

フィルターウィンドウに use 要素を含めて、入力イベントのどのフィールドが Lua フィルター関数によって処理されるかを制限します。Lua に渡す入力フィールドを少なくすることで、処理効率を向上させることができます。

たとえば、Trades プロジェクトには数百の入力フィールドが含まれる可能性がありますが、単一の入力フィールドの値に基づいてイベントをフィルター処理したい場合があります。次の例では、フィルター条件を単一の入力フィールドの値 quant に制限しています。

```
<window-filter name="largeTrades" func="bob">
  <use>quant</use>
  <code><![CDATA[
    function bob(data)
      return data.quant >= 1000
    end
  ]]></code>
</window-filter>
```

use 要素がない場合、すべての入力フィールドが Lua フィルター関数に渡されます。

要素をフィールドなしに設定した場合(<use/>)の場合、フィールドは Lua フィルター関数に渡されません。この場合、関数呼び出しを使用して *ESP サーバー*側のフィールドにアクセスする必要があります。たとえば、Lua で処理したいウィンドウを介した JSON データストリーミングがある場合、<use/>を指定し、esp_parseJsonFrom('field')を使用してデータを取得できます。

注: とはいえ、use 要素はオプションなので、ベストプラクティスとして、この要素を使用して、 *ESP サーバー*から Lua フィルター関数に最小数のフィールドを渡します。

filter 関数のパラメーターは次のとおりです。

表 3.1 フィルター関数のパラメーター

パラメーター	説明
object	すべての入力イベントフィールドを含む Lua オブジェクトを指定します。use 要素を指定した場合、指定されたイベントフィールドが含まれます。このパラメーターは必須です。
context	現在のイベントのコンテキストデータを指定します。有効な値は次のとおりです。 ■ **window** - 現在のウィンドウの名前。 ■ **input** - このイベントの入力ウィンドウの名前。 このパラメーターは任意です。

Lua コードで初期化を実行する必要がある場合は、フィルターウィンドウの init 属性を指定します。この属性の値は、有効な Lua 関数でなければなりません。パラメータは init 関数に渡されません。

Lua コードでクリーンアップを実行する必要がある場合は、フィルターウィンドウに destroy 属性を指定します。この属性の値は、有効な Lua 関数でなければなりません。この関数は、フィルターウィンドウが破棄されるときに呼び出されます。

内部 Lua モジュールを取り込みたい場合は、それらをフィルターウィンドウの require 属性にで指定します。含める内部 Lua モジュールのカンマ区切りリストを指定します。

断続的な処理を実行するために、Lua ベースのフィルターウィンドウは、heartbeat 属性をサポートします。この属性の値は、有効な Lua 関数でなければなりません。ハートビート関数に渡されるパラメーターはありません。

フィルターウィンドウでの Python の使用

注: Python ベースのフィルターウィンドウを使用する場合、コードは Python 3.11 と互換性がある必要があります。

Python コードブロックでは、ブール値を返すフィルター関数を指定する必要があります。フィルター関数の名前は filter です。window-filter 要素の func 属性を使用して、フィルター関数の別名を指定できます。ESP サーバーは、入力イベントごとにこの関数を呼び出し、戻りコードを使用して出力イベントを生成するかどうかを決定します。

たとえば、次の XML コードは、**count** 入力フィールドの値が 100 以上の場合は常に、すべてのイベントの通過を許可します。

```
<window-filter name="filter_w" func="bob">
  <code><![CDATA[
def bob(event, context):
    return event["the_count"] >= 100
  ]]></code>
</window-filter>
```

フィルターウィンドウに use 要素を含めて、入力イベントのどのフィールドが Python フィルター関数によって処理されるかを制限します。

たとえば、Trades プロジェクトには数百の入力フィールドが含まれる可能性がありますが、単一の入力フィールドの値に基づいてイベントをフィルター処理したい場合があります。次の例では、フィルター条件を単一の入力フィールドの値 quant に制限しています。

```
<window-filter name="largeTrades">
    <use>quant</use>
    <code><![CDATA[
def filter(data, context):
    return data["quant"] >= 1000
    ]]></code>
</window-filter>
```

use 要素がない場合、すべての入力フィールドが Python フィルター関数に渡されます。

filter 関数のパラメーターは次のとおりです。

表 3.2 フィルター関数のパラメーター

パラメーター	説明
event	入力イベントを指定します。use 要素を指定した場合、指定されたイベントフィールドが含まれます。このパラメーターは必須です。
context	フィルター関数が使用する変数、データおよびリソースが含まれます。フィルターウィンドウでは、コンテキストに次の情報を含めることができます。 ■ **window** - 含まれている Python ウィンドウの ID。 ■ **input** - このイベントの入力ウィンドウの名前。

Python コードで初期化を実行する必要がある場合は、フィルターウィンドウの init 属性を指定します。この属性の値は、有効な Python 関数でなければなりません。パラメータは init 関数に渡されません。

Python コードでクリーンアップを実行する必要がある場合は、フィルターウィンドウに destroy 属性を指定します。この属性の値は、有効な Python 関数でなければなりません。この関数は、フィルターウィンドウが破棄されるときに呼び出されます。

断続的な処理を実行するために、Python ベースのフィルターウィンドウは、heartbeat 属性をサポートします。この属性の値は、有効な Python 関数でなければなりません。ハートビート関数に渡されるパラメーターはありません。

集計ウィンドウの使用

集計ウィンドウの概要

集計ウィンドウは、非キーフィールドが計算される点で計算ウィンドウに似ています。ただし、集計ウィンドウのキーフィールドを指定する必要があります。それらは入力ウィンドウから継承されません。それらのキーフィールドは入力イベントの存在するフィールドに対応する必要があります。受信イベントは集計グループに入れられます。集計グループ内の各イベントの指定されたキーフィールドの値は同じです。

たとえば、次のスキーマが入力イベントに指定されているとします。

"ID*:int32,symbol:string,quantity:int32,price:double"

集計ウィンドウのスキーマを次のように指定するとします。

"symbol*:string,totalQuant:int32,maxPrice:double"

イベントが集計ウィンドウに到着すると、それらは symbol フィールドの値に基づいて集計グループに配置されます。集計フィールド計算関数または集計ウィンドウに登録されている式は、キー以外のフィールドに表示する必要があります。この例では、totalQuant と maxPrice です。すべてのキー以外のフィールドには、式または関数を使用する必要があります。これらは混在することはできません。関数または式は、新規作成イベントが来るたびに引数の 1 つとしてイベントのグループとともに呼び出され、1 つまたは複数のグループを変更します。

これらのグループは、内部的に dfESPwindow_aggregate クラスで dfESPgroupstate オブジェクトとして管理されます。すべてのキー以外のフィールドで指定された集計関数または式を実行することによって、グループから新規作成イベントが追加または削除されるたびに、各グループが折りたたまれます。集計ウィンドウは、グループごとに 1 つの集計イベントを生成します。

集計ウィンドウに関連付けられている XML エレメントについては、“window-aggregate” (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

操作の流れ

集計ウィンドウの処理中の操作の流れは次のとおりです。

1 イベント **E** が到着し、**G** と呼ばれる適切なグループが見つかります。グループは、集計ウィンドウのキーフィールドに対応する着信イベントの値を調べることで見つかります。

2 イベント **E** は、グループ **G** にマージされます。出力イベントのキーは、**G** の groupBy フィールドで形成されます。

3 出力スキーマの各非キーフィールドは、グループ **G** を入力として集計関数を呼び出すことによって計算されます。集計関数は、対応するキー以外のフィールドのスカラー値を計算します。

4 正しい出力イベントが生成され、出力されます。

集計関数の使用

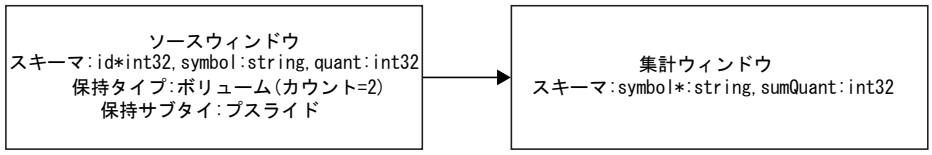
集計関数の概要

集計中に集計ウィンドウに入るイベントは、集計ウィンドウのキーフィールドに基づいてグループに配置されます。SAS Event Stream Processing で提供される集計関数は各イベントグループで実行され、集計ウィンドウのキー以外のフィールドごとに計算します。

集計ウィンドウのキー以外のフィールドに指定されている関数は特殊関数です。これらはイベント値のグループを操作し、それらを単一のスカラー値に集約します。

次のダイアグラムでは、集計ウィンドウのキーは **symbol** です。集計ウィンドウには、**sumQuant** という 1 つのキー以外のフィールドしかありません。このフィールドは、ソースウィンドウから到着したフィールド **quant** の合計を保持します。

図 3.1 集計の例



フィールド値の合計を計算する関数は、ESP_aSum(*fieldname*)です。ここで、集計ウィンドウには ESP_aSum(quant)として計算される非キーフィールドが 1 つあります。概念的には、イベントが集計ウィンドウに入ると、そのイベントがグループに追加され、ESP_aSum(quant)関数が実行され、グループの新規作成合計が生成されます。

配列に対して集計関数を使用できます。たとえば、次のような入力スキーマがあるとします。

name*: string, value: array

ソースウィンドウが次の 5 つのイベントを集計ウィンドウにフィードするとします。

表 3.3 ソースウィンドウを介したイベントのストリーミング

イベント	名前 (キー)	値
e1	A	[1.0, 2.0]
e2	A	[0.0, 7.0]
e3	B	[5.0, 2.0]
e4	A	[3.0, 1.0]

イベント	名前 (キー)	値
e5	B	[3.0, 4.0]

ここで、ソースウィンドウからこれらのイベントを受け取る集計ウィンドウに ESP_aMax(value)を適用するとします。最初に、集計ウィンドウでは、データがキーフィールド name によってグループ化されます。

表 3.4　キーフィールドによるイベントのグループ化

キー	値
A	[1,0, 2.0] [0.0, 7.0] [3.0, 1.0]
B	[5.0, 2.0] [3.0, 4.0]

集計ウィンドウの ESP_aMax(value)関数は、配列の位置ごとに動作します。A の配列値の最初の要素の最大値は 3.0 で、2 番目の要素の最大値は 7.0 です。したがって、A の場合、関数は次の結果を生成します。

[3.0, 7.0]

B の場合、関数は次の結果を生成します。

[5.0, 4.0]

集計関数は、配列が対称でない場合でも結果を返します。次に例を示します。

```
ESP_aSum([1,2,3],[1,2],[3])
```

Returns [5]: 1+1+3=5.このような場合の結果の長さは、最短の配列の長さになります。

aCount、aCountNonNull、aCountNull などのフィールドをカウントする集計関数、または aFirst、aLast、aLastNonNull などのグループからフィールドを選択する関数は、配列全体で動作します。配列の要素は検査しません。集計ウィンドウに次のデータをストリーミングするとします。

```
[1;2;3]
(NULL)
[3;4;5]
(NULL)
```

表 3.5　配列データに適用される集計関数

関数	返される値
aCount	4
aCountNonNull	2

関数	返される値
aCountNull	2
aFirst	[1;2;3]
aLast	(NULL)
aLastNonNull	[3;4;5]

count 関数の場合、戻り値の型が int64 である必要があります。first 関数と last 関数の場合、戻り値の型は入力データと同じ配列型でなければなりません。

自分の集計関数を作成できます。詳細については、“アプリケーションに埋め込むための集計関数の作成”を参照してください。

集計ウィンドウフィールド計算式の集計関数

集計ウィンドウのフィールド計算式では、次の集計関数を使用できます。加法関数は、保持された状態および受信イベントから決定された値から算出できます。グループ状態を維持する必要はありません。集計ウィンドウでこれらの関数のみを使用した場合は、ウィンドウ処理で桁違いの特別な最適化が発生する可能性があります。

一部の集計関数は受信イベントの演算コードに関係なく加法です。その他の関数は、挿入を取得したときにのみ加法です。一部の関数は配列入力を処理できます。

表 3.6 集計関数

集計関数	加法	挿入の加法	配列	返される値
ESP_aAve(*fieldname*)	True	True	位置ごと	グループの平均。
ESP_aAveTimed(*fieldname, N*)	True	True	位置ごと	グループ内のイベントの平均。最初のイベントから *N* 秒以上経過して新しいイベントが来た場合、平均をクリアします。 これにより、イベントの保持を導入しなくてもクリアする時間を区切った平均を集計できます。
ESP_aCat(*fieldname*, *stringConstant*)	False	True	False	指定された *stringConstant* をセパレーターとして使用して、集計グループ内の各イ

集計関数	加法	挿入の加法	配列	返される値
				ベントの *フィールド名*値を連結します。 たとえば、ESP_aCat(ID, "|")は集計グループを構成するすべての ID の区切りリスト"|"を返します。
ESP_aCount()	True	True	False	グループ内のイベントの数。
ESP_aCountDistinct(*fieldname*)	True	True	False	グループ内の *fieldname* で指定された列内の個別の非ヌル値の数。
ESP_aCountNonNull(*fieldname*)	False	True	配列がヌルでない場合にカウントします。	グループ内の指定された *fieldname* の列に非ヌル値を持つイベントの数。
ESP_aCountNull(*fieldname*)	False	True	配列がヌルの場合にカウントします	グループ内の指定された *fieldname* の列にヌル値を持つイベントの数。
ESP_aCountTimed(*N*)	True	True	False	グループ内のイベントの数のカウント。最初のイベントから *N* 秒以上経過して新しいイベントが来た場合、カウントをクリアします。 これにより、イベントの保持を導入しなくてもクリアする時間を区切ったカウントを集計できます。
ESP_aCountOpcodes(*opcode*)	True	True	False	グループの *opcode* に一致するイベントの数のカウント。 整数値に *opcode* を指定します。 ■ NOOP = 0

集計関数	加法	挿入の加法	配列	返される値
				■ INSERT = 1 ■ UPDATE = 2 ■ DELETE = 3 ■ UPSERT = 4 ■ SAFEDELETE = 5 ■ UPDATEBLOCK = 6 たとえば、DELETE に一致するイベントをカウントするには、次のように指定します。 ESP_aCountOpcodes(3)
ESP_aFirst(*fieldname*)	False	True	位置ごと	グループに追加された最初のイベント。
ESP_aGUID()	True	True	False	一意の識別子。
ESP_aLag(*fieldname,lag_value*)	False	True	位置ごと	指定された *lag_value* は、集約されたイベント値のグループへのインデックスを表します。インデックスが 0 の場合は、そのグループに影響を与えた現在のイベントを表します。インデックスが 1 の場合は、グループに影響を与えた直前の出来事などを表します。したがって、次のことが成り立ちます。 ■ ESP_aLag(*fieldname*,0) == ESPaLast(*fieldname*)は、グループに影響を与えた現在のイベントの *fieldname* の値を返します。 ■ ESP_aLag(*fieldname*,1)は、グループに影響を与えた直前のイベントの *fieldname* の値を返します。
ESP_aLast(*fieldname*)	False	True	位置ごと	グループに影響を与えた最後のレコードのフィールド。

集計関数	加法	挿入の加法	配列	返される値
ESP_aLastNonNull(*fieldname*)	False	True	位置ごと	グループに影響を与えた最後のレコード(非ヌル値)のフィールド。
ESP_aLastOpcode()	True	True	False	グループに影響を与える最後のレコードの演算コード。
ESP_aMax(*fieldname*)	False	True	位置ごと	グループの最大値。
ESP_aMaxTimed(*fieldname, N*)	True	True	位置ごと	グループの最大値。最初のイベントから *N* 秒以上経過して新しいイベントが来た場合、最大値をクリアします。 これにより、イベントの保持を導入しなくてもクリアする時間を区切った最大値を集計できます。
ESP_aMin(*fieldname*)	False	True	位置ごと	グループの最小値。
ESP_aMinTimed(*fieldname, N*)	True	True	位置ごと	グループの最小値。最初のイベントから *N* 秒以上経過して新しいイベントが来た場合、最小値をクリアします。 これにより、イベントの保持を導入しなくてもクリアする時間を区切った最小値を集計できます。
ESP_aMode(*fieldname*)	True	True	False	モード、または最も人気のあるグループ。
ESP_aStd(*fieldname*)	True	True	位置ごと	グループの標準偏差。
ESP_aSum(*fieldname*)	True	True	位置ごと	グループの合計。
ESP_aSumTimed(*fieldname, N*)	True	True	位置ごと	グループの合計。最初のイベントから *N* 秒以上経過して新しいイベントが来た場合、合計をクリアします。

集計関数	加法	挿入の加法	配列	返される値
				これにより、イベントの保持を導入しなくてもクリアする時間を区切った合計を集計できます。
ESP_aWAve(*weight_fieldname, payload_fieldname*)	True	True	位置ごと	加重平均グループ。
ESP_aWAveTimed(*weight_fieldname, payload_fieldname, N*)	True	True	位置ごと	加重平均グループ。最初のイベントから *N* 秒以上経過して新しいイベントが来た場合、平均をクリアします。 これにより、イベントの保持を導入しなくてもクリアする時間を区切った加重平均を集計できます。

挿入イベントを供給すると、すべての集計関数は加法になります。したがって、モデルのパフォーマンスに関して次の点を考慮してください。

- 挿入専用の集計ウィンドウに無制限の数のエレメントを持つキーがあると、ウィンドウのメモリは無制限に増えます。受信イベントによって渡される値を予測することはできないため、これは *ESP サーバー*によって自動的に検出できません。この場合、警告が発行されます。
- 挿入専用ではない集計ウィンドウの場合、ウィンドウは次のいずれかになります。
  - 加法、つまり、常に加法としてマークされている集計関数のみを使用するウィンドウ。キーのカーディナリティが無制限の場合、ウィンドウのメモリは無制限に増えます。挿入を受信していないため、警告は発行されません。ここでは、集計ウィンドウ内のイベント総数を制御するために、集計ウィンドウの前に、保持したままコピーウィンドウを配置する必要があります。
  - 非加法、つまり ESP_aMax や ESP_aMin などの加法でない集計関数を使用するウィンドウ。この場合、メモリの増加率は主に、集計ウィンドウで常にアクティブになっているイベントの最大数によって決まります。この場合、集計ウィンドウには、ストリーミング配信される各イベントのコピーが保持されます。また、イベントがどのグループに属しているかも追跡されます。ここでは、集計ウィンドウの前に保持されたままのコピーウィンドウを配置することをお勧めします。これにより、集計ウィンドウには必ず有限の数のイベントがあることが保証されます。イベントの数は保持ポリシーによって決まります。

イベント履歴はパフォーマンスに影響します。

- いくつかの集計関数(たとえば、ESP_aMin および ESP_aMax)では、更新と削除を処理するためにソースイベントの保存(履歴)が必要です。
- すべての集計関数は、履歴なしで、センサー読み取り値などの挿入専用イベントストリームを処理できます。

- 更新と削除を伴うイベントストリームについては、最高のパフォーマンスを得るために、履歴を必要としない集計関数のみを使用します(たとえば、ESP_aSum、ESP_aAve)。

該当する場合、ソースイベント履歴は内部インデックスに保持されます。

キー以外のフィールド計算式に集計関数を簡単に使用できます。たとえば、

```
<window-aggregate name='brokerAlertsAggr' index='pi_HASH'>
  <schema-string>brokerName*:string,frontRunningBuy:int32,frontRunningSell:int32,
          openMarking:int32,closeMarking:int32,restrictedTrades:int32,total:int64
  </schema-string>
  <output>
    <field-expr>ESP_aSum(frontRunningBuy)</field-expr>
    <field-expr>ESP_aSum(frontRunningSell)</field-expr>
    <field-expr>ESP_aSum(openMarking)</field-expr>
    <field-expr>ESP_aSum(closeMarking)</field-expr>
    <field-expr>ESP_aSum(restrictedTrades)</field-expr>
    <field-expr>ESP_aSum(total)</field-expr>
  </output>
</window-aggregate>
```

注: ESP_aSum、ESP_aMax、ESP_aMin、ESP_aAve、ESP_aStd、および ESP_aWAve では、フィールド内のヌル値は無視されます。したがって、それらは計算に寄与しません。

受信イベントに統計を追加するには集計関数を使用

ESP_aLast(*fieldName*)集計関数を使用して、受信フィールドを集計イベントに渡すことができます。これは、集計ウィンドウを使用せずに結合ウィンドウを使用することなく、集計ウィンドウからイベントに統計を追加する場合に便利です。または、集計ウィンドウの後に結合ウィンドウを使用すると、集計計算またはイベントが集計ウィンドウに供給される同じイベントに結合されます。しかし、その場合の結果は最適ではないかもしれません。

株取引を処理しているとします。いくつかのフィールドを含む受信イベントを処理するソースウィンドウ。

```
<schema>
  <fields>
    <field name="ID" type="int32" key="true"/>
    <field name="symbol" type="string"/>
    <field name="currency" type="int32"/>
    <field name="udate" type="int64"/>
    <field name="msecs" type="int32"/>
    <field name="price" type="double"/>
    <field name="quant" type="int32"/>
    <field name="venue" type="int32"/>
    <field name="broker" type="int32"/>
    <field name="buyer" type="int32"/>
    <field name="seller" type="int32"/>
    <field name="buysellflg" type="int32"/>
  </fields>
</schema>
```

集約ウィンドウの受信イベントスキーマを、次の 3 つの変数に制限したいとします。symbol、price、および quant です。これらの変数に、次の 2 つの集計統計を追加します。priceAVE と quantMAX。

```
<schema>
  <fields>
   <field name="symbol" type="string" key="true"/>
   <field name="price" type="double"/>
   <field name="quant" type="int32"/>
   <field name="priceAVE" type="double"/>
   <field name="quantMAX" type="int32"/>
  </fields>
 </schema>
```

注: groupBy は集計のキーであり、この場合 symbol です。

field-expr エレメントを使用して、price および quant というキー以外のフィールドに次の集計関数を登録します。

```
<output>
  <field-expr>ESP_aLast(price)</field-expr>
  <field-expr>ESP_aLast(quant)</field-expr>
  <field-expr>ESP_aAve(price)</field-expr>
  <field-expr>ESP_aMax(quant)</field-expr>
 </output>
```

次のイベントがソースウィンドウにストリーミングされるとします。

```
i    n    21    SAIA    0    55110    932    16.2328    100    4    92    3
58    1
```

次のイベントは、その後集計ウィンドウによって作成されます。

```
I    N     SAIA    16.2328    100    16.2328    100
```

その後、次のイベントがソースウィンドウにストリーミングされます。

```
i    n    392    SAIA    7    55110    934    15.1296    400    3    64    5
75    1
```

次のイベントが、集計ウィンドウによって作成されます。

```
U    N     SAIA    15.1296    400    15.6812    400
```

最後に、次のイベントがソースウィンドウにストリーミングされます。

```
i    n    531    SAIA    1    55110    934    15.2872    1100    5    52    73
82    0
```

次のイベントが、集計ウィンドウによって作成されます。

```
U    N     SAIA    15.2872    1100    15.549867    1100
```

ESP_aLast(*fieldName*)を使用して、関連する集計フィールドを追加することで、後続の結合ウィンドウを回避できます。これによりモデリングがよりクリーンになります。

計算ウィンドウの使用

計算ウィンドウの概要

計算ウィンドウを使用すると、入力イベントストリームフィールドの計算操作を通じて入力イベントの出力イベントへの 1 対 1 変換が可能になります。計算ウィンドウを使用して、あるイベントの入力フィールドを新しいイベントに投影し、計算結果のフィールドを使用して新規作成イベントを補強することができます。

キーフィールドのセットは計算ウィンドウ内で変更できますが、この関数は慎重に使用してください。計算ウィンドウ内でキーフィールドを変更する場合、入力イベントのキーの挿入、更新、および削除は、新規キーセットの挿入、更新、および削除と同等でなければなりません。

計算ウィンドウに関連付けられている XML エレメントについては、"window-compute" (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)および"計算およびフィルタウィンドウに関連する XML 言語要素" (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

計算関数の使用

計算ウィンドウに入るイベントは、計算ウィンドウのキーフィールドに基づいてグループに配置されます。関数または式は各グループで実行され、計算ウィンドウの各非キーフィールドを計算します。

計算ウィンドウは、式、ユーザー定義関数、またはプラグイン関数を使用して入力ストリームの計算を実行します。計算ウィンドウの出力フィールドは、別のウィンドウまたはサブスクライブコネクタにプッシュすることができます。

ユーザー定義関数は、**window-compute** の冒頭にある **udfs** および **expr-initialize** エレメントの **udf** で指定されます。例については、"式エンジン言語でのスプリッター関数の使用"を参照してください。

登録されたプラグイン関数は、**window-compute** の **output** エレメント内の **field-plug** XML 言語エレメントで指定されます。これらの関数は、共有ライブラリ (**DFESP_HOME** ディレクトリにある **libmethod.so** や **libmethod.dll** など)から供給されます。

和結合(Union)ウィンドウの使用

和結合(Union)ウィンドウは、厳密なポリシーまたは緩やかなポリシーを使用して複数のイベントストリームを結合します。和結合(Union)ウィンドウに関連付けられている XML エレメントについては、*“window-union” (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*を参照してください。

和結合(Union)ウィンドウへのすべての 入力ウィンドウは、同じスキーマを持つ必要があります。strict フラグのデフォルト値は **true** です。つまり、各ウィンドウのキーマージを意味的に完全にマージする必要があります。

次の表は、状態と strict フラグの値の様々な組み合わせが与えられた和結合ウィンドウ内での結果を説明しています。

表 3.7　和結合(Union)ウィンドウ内の結果

和結合ウィンドウの状態	strict の値	和結合ウィンドウ結果
ステートレス	**false**	挿入はアップサートに置き換えられます。削除は安全な削除に置き換えられます
ステートレス	**true**	レコードの挿入と削除は変更されずに通過します。
ステートフル	**false**	キーが重複している挿入はアップサートとして処理されます。存在しないキーを持つ削除は、安全な削除として処理されます。
ステートフル	**true**	重複したキーでの挿入や、存在しないキーでの削除は、エラーを発生させ、不良レコードとして処理されます。

結合ウィンドウの使用

結合ウィンドウの概要

結合ウィンドウは、左入力ウィンドウと右入力ウィンドウからイベントを受け取ります。ここでは、結合されたイベントの単一出力ストリームが生成されます。結合されたイベントは、ユーザー指定の結合タイプおよびユーザー定義の結合条件に従って作成されます。

結合順序は、プロジェクトのエッジレベルで決定されます。左ウィンドウは、結合ウィンドウへの接続エッジとして定義された最初のウィンドウです。接続エッジとして定義されている 2 番目のウィンドウが右ウィンドウです。

XML を使用すると、明示的に役割をエッジに割り当てて、**left** と **right** のどちらのウィンドウを結合ウィンドウに接続するかを定義できます。たとえば、

```
<edges>
  <edge source='w_source1' target='w_join' role='left'/>
  <edge source='w_source2' target='w_join' role='right'/>
</edges>
```

使用できる結合のタイプにはいくつかの制限があります。4 つの結合タイプは次のとおりです。

左外部結合
: 左外部結合は、左ウィンドウから到着するすべてのイベントに対して結合された出力イベントを生成します。結合されたイベントは、右ウィンドウから一致するイベントがない場合でも作成されます。

右外部結合
: 右外部結合は、右ウィンドウから到着したすべてのイベントに対して結合された出力イベントを生成します。結合されたイベントは、左ウィンドウから一致するイベントがない場合でも作成されます。

内部結合
: 内部結合は、入力イベントの反対側に 1 つ以上の一致するイベントがある場合にのみ結合イベントを作成します。

完全外部結合
: 完全外部結合は、結合の左ウィンドウまたは右ウィンドウから到着するすべてのイベントの結合イベントを作成します。出力は常に生成されます。

ユーザー定義の結合条件は、結合されたイベントを生成するために使用される左右の入力ウィンドウのキーフィールドを指定します。結合条件は、式の n-タプルです。各式は、左ウィンドウからの 1 つのフィールドと、右からの 1 つのフィールドを含みます。例: *(left.f* $_{1}$ *= = right.f* $_{10}$ *)*、*(left.f* $_{2}$ *= = right.f* $_{7}$ *)*、*...(左のフィールド* $_{10}$*==右のフィールド* $_{1}$*)*

新規作成入力イベントが到着したときに非キー結合フィールドを算出するには、結合ウィンドウに次のいずれかが表示されます。

- フィールドを結合する入力フィールドの 1 対 1 のマッピングである結合選択文字列
- フィールド算出式
- フィールド算出関数

結合ウィンドウに関連付けられている XML エレメントについては、*“window-join” (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*および*“結合ウィンドウに関連する XML 言語要素” (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*を参照してください。

ストリーミング結合について

ストリーミング結合の概要

ストリーミング結合を理解するには、次の定義が不可欠です。X-Y 結合は、次のものが存在する結合です。

- 左ウィンドウからの 1 つのイベントは、結合ウィンドウの多くの X イベントに影響を与えます。
- 右ウィンドウ内の 1 つのイベントは、結合ウィンドウのほとんどの Y イベントに影響を与えます。

左ウィンドウ、右ウィンドウ、および結合条件のセットが与えられると、ストリーミング結合は 3 つの異なるカテゴリのうちの 1 つに分類することができます。

結合カテゴリー	説明
1 対 1 の結合	結合のいずれかの側のイベントは、結合の反対側からの最大 1 つのイベントと一致することができます。このタイプの結合には、常に 2 つのディメンションテーブルが含まれます。
1 対多の結合 (または多対 1 の結合)	結合の片側に到着するイベントは、結合の多くの行に一致する可能性があります。結合の反対側に到着するイベントは、最大で 1 つの結合の行と一致することができます。 結合を 1 対多(または多対 1)結合として分類するには、次の条件を満たしている必要があります。 ■ 結合の一方の側への変更は、多くの行に影響する可能性があること。 ■ 他の側への変更は多くても 1 行に影響すること。

結合カテゴリー	説明
多対多の結合	結合の両側に到着する単一のイベントは、結合の反対側にある複数のイベントと一致する可能性があります。

ストリーミングコンテキストでは、すべてのウィンドウにイベントの挿入、削除、および更新を可能にする主要なキーがあります。結合ウィンドウのキーフィールドは、入力ウィンドウから派生し、ユーザーは決して指定しません。イベントがいずれかの側に到着すると、到着するデータの性質(挿入、更新、または削除)を考慮して、結合がどのように変化するかを計算できる必要があります。

SAS Event Stream Processing では、ユーザーが指定した結合タイプと結合条件に基づいて、結合ウィンドウのキーが決定されます。SAS Event Stream Processing は、最も一般的な結合ケースの一貫性を維持するための一連の公理に従います。

結合カテゴリー	キー導出公理
1 対 1 の結合	左外部結合の場合は、左入力ウィンドウのキーが結合ウィンドウに渡されます。右ウィンドウのキーは、決して結合ウィンドウに渡されません。これは、一致する右ウィンドウイベントがなくても、左ウィンドウイベントが到着したときに結合ウィンドウイベントが生成されるためです。 右外部結合の場合は、右入力ウィンドウのキーが結合ウィンドウに渡されます。左ウィンドウのキーは、決して結合ウィンドウに渡されません。これは、一致する左ウィンドウイベントがなくても、右ウィンドウイベントが到着したときに結合ウィンドウイベントが生成されるためです。
1 対多の結合 (または多対 1 の結合)	1 対多と多対 1 の結合の場合、“多数の側“のキーが結合ウィンドウキーとして使用されます。結合ウィンドウのイベントを区別するために、多くの側は、結合条件で指定されたすべてのキーがない側です。
多対多の結合	多対多の結合の場合、左と右の入力ウィンドウのキーの和結合(Union)が結合ウィンドウのキーフィールドとして使用されます。左ウィンドウイベントは複数の右ウィンドウイベントに一致する可能性があるため、右ウィンドウからのキーは結合ウィンドウイベントを区別するために必要です。同様に、右ウィンドウイベントは複数の左ウィンドウイベントと一致する可能性があるため、左ウィンドウからのキーは結合ウィンドウイベントを区別するために必要です。

左右の入力ウィンドウは、ディメンションウィンドウまたはファクトウィンドウとして分類されます。ディメンションウィンドウは、キーフィールド全体が結合条件に参加するウィンドウです。ファクトウィンドウは、結合条件に参加しない少なくとも 1 つのキーフィールドを持つウィンドウです。入力ウィンドウは、イベントを結合ウィンドウに送信する前にディメンションウィンドウまたはファクトウィンドウとして分類されません。

次のテーブルは、許容結合サブタイプと、公理および指定された結合条件に基づくキー派生をまとめたものです。

結合カテゴリー	左ウィンドウ	右ウィンドウ	許容されるタイプ	キー導出	ストリーミングウィンドウ
1 対 1	ディメンション	ディメンション	左外部	結合キーは左ウィンドウのキーです。	左
			右外部	結合キーは右ウィンドウのキーです。	右
			完全外部	結合キーは、左ウィンドウのキーです(任意の選択肢)。	左
			内部	結合キーは、左ウィンドウのキーです(任意の選択肢)。	左
1 対多数	ファクト	ディメンション	左外部	結合キーは左ウィンドウのキーです(右ウィンドウはルックアップです)。	左
			内部	結合キーは左ウィンドウのキーです(右ウィンドウはルックアップです)。	左
多対 1	ディメンション	ファクト	右外部	結合キーは右ウィンドウのキーです(左ウィンドウはルックアップです)。	右
			内部	結合キーは右ウィンドウのキーです(左ウィンドウは	右

結合カテゴリー	左ウィンドウ	右ウィンドウ	許容されるタイプ	キー導出	ストリーミングウィンドウ
				ルックアップです)。	
多対多	ファクト	ファクト	内部	結合キーは、左右のウィンドウからのキーの完全なセットです。	なし

注: ウィンドウのすべてのキーが結合条件で使用されると、条件の側に追加の非キーフィールドが追加されません。たとえば、結合条件に関与する左側のフィールドのセットに、左ウィンドウのすべてのキーが含まれているとします。そのセット内のキー以外のフィールドは無視されます。

セカンダリインデックスの使用

許可された 1 対多および多対 1 の結合の場合、ファクトテーブルを変更すると、プライマリインデックスを使用してディメンションテーブルの 一致するレコードを即座にルックアップできます。ディメンションテーブルのすべてのキー 値は、結合条件にマップされます。ただし、ディメンションテーブルの 変更には、ファクトテーブルの一致するレコードの単一の主要なキーは含まれません。これは、結合の多対 1 の性質を示しています。デフォルトでは、ファクトテーブル内の一致するレコードは、テーブルスキャンによって 検索されます。

ディメンションテーブルの 変更が非常に限られている場合は、追加のセカンダリインデックスのメンテナンスが必要なくなるため、結合処理を最適化できます。ここで、ディメンションテーブルは、プレロードできるスタティックルックアップテーブルです。以降の変更はすべてファクトテーブルで 行われます。

ディメンションテーブルに 対して多数の変更が可能な場合は、結合でセカンダリインデックスを有効にすることをお薦めします。自動セカンダリインデックスの生成は、新規作成結合ウィンドウを作成するときに結合パラメーターを指定することによって使用可能になります。これにより、結合タイプにディメンションテーブルが含まれている場合に、セカンダリインデックスが自動的に生成および維持されます。

セカンダリインデックスをオンにして実行すると、パフォーマンスがわずかに低下します。ファクトテーブルの更新ごとにインデックスを維持する必要があります。ただし、セカンダリインデックスを生成すると、ディメンションテーブルが 変更されたときにすべてのテーブルスキャンを 削除できるという利点があります。セカンダリインデックス保守は、テーブルスキャンの除去と比較して重要ではありません。大きなテーブルの場合、セカンダリインデックスを使用することで、桁違いの時間を節約できます。

多対多の結合では、セカンダリインデックスを有効にすることをお勧めします。

再生と非再生の使用

デフォルトの結合動作では、結合のいずれかの側に変更が加えられると、結合ウィンドウの適切な行が常に再生成されます。これの典型的な例は、左外部結合です。右ウィンドウはルックアップウィンドウで、左のテーブルはファクト(ストリーミング)ウィンドウです。通常、結合のルックアップ側には事前入力が行われ、左ウィンドウからイベントがストリームされると、それらが一致し、結合されたイベントが出力されます。通常、これは結合のストリーミング側の 1 対 1 の関係です。1 つのイベントは 1 つの結合されたイベントです。ディメンション側で変更が行われることがあります。この変更は、イベントの更新、イベントの削除、または新規作成イベントの挿入の形で行うことができます。既定の動作では、結合の一貫性を維持するイベントの変更セットを発行します。

再生成モードでは、右ウィンドウ(ルックアップ側)への変更時の左外部結合の動作は次のようになります。

- Insert:新しいイベントと一致する既存のファクトイベントをすべて 検索します。見つかった場合は、これらのイベントごとに更新を発行します。ルックアップ側のフィールドには、以前に処理されたときにヌルが使用されていました。
- 削除:削除するイベントに一致するファクトイベントを検索します。見つかった場合は、これらのイベントごとに更新を発行します。ルックアップイベントに一致するフィールド値を使用していたので、ルックアップイベントが削除されたときにヌルを使用する必要がありました。
- 更新:新しいイベントが挿入された古いイベントの削除のように動作します。ストリーミング側のキーにマッピングされるルックアップ側の非キーフィールドのいずれかが考慮されます。これらのフィールドのいずれかが値を変更したかどうかが判定されます。

再生しないモードでは、右ウィンドウ(ルックアップサイド)への変更に左外部結合がある場合、ディメンション(ルックアップ)テーブルの変更は新規作成ファクトイベントにのみ影響します。結合によって処理された以前のファクトイベントはすべて再生成されません。これは、新規作成ディメンションウィンドウが定期的にフラッシュされて再ロードされるときに頻繁に発生します。

結合ウィンドウには、デフォルトでは false である no-regenerates フラグがあります。これにより、結合完全リレーショナル結合セマンティクスが得られます。結合ウィンドウのこのフラグを true に設定すると、no-regenerates セマンティクスが有効になります。フラグを true に設定することは、1 対多、多対 1、および 1 対 1 の内部結合とともに、左または右の外部結合のいずれに対しても許可されます。結合ウィンドウが実行中の場合 no-regenerates モードでは、結合ウィンドウで通常維持されるファクトウィンドウの索引の参照カウントコピーを省略して、メモリ使用量を最適化します。

空のインデックス結合の作成

ルックアップテーブルと挿入専用のファクトストリームがあるとします。ファクトストリームをルックアップテーブルと照合して(Insert を生成する)、後で処理するた

めにストリームを結合から外したいとします。この場合、結合でファクトデータを格納する必要はありません。ファクトデータが格納されていないため、ディメンションデータの変更は後続の行にのみ影響します。変更は既存のファクトデータを元に戻すことはできません(結合にはステートレスであるため)。更新を発行します。結合が既存のデータを元に戻さないようにするには、no-regenerates プロパティを有効にする必要があります。

LeftOuter または RightOuter の結合タイプがあるとします。インデックスタイプは pi_EMPTY に設定され、ステートレス結合ウィンドウがレンダリングされます。no-regenerates フラグは TRUE に設定されます。これはできるだけ軽量な結合です。結合で保持される唯一のデータは、ディメンションテーブルデータのローカル参照カウントコピーです。このコピーは、ファクトデータが結合に流入してから流出するときにルックアップを実行するために使用されます。

結合ウィンドウでは、左と右の入力に個別に挿入専用を指定することはできません。結合ウィンドウを“挿入のみ“に設定することにより、結合の両側に挿入専用を指定することは、あまりにも制限的です。これは、参照の変更、または結合の非ストリーミング側の変更を許可しません。期待される結果を確実にするために、これらのルールに従わなければなりません。

- 多対多の結合には空の索引を使用できません。
- 結合分類テーブルで指定されているように、結合のストリーミング側は挿入のみを受け取ることができます。

結合ウィンドウの例

次の例は、左外部結合を示しています。左ウィンドウはファクトイベントを処理し、右ウィンドウはディメンションイベントを処理します。

```
left input schema: "ID*:int32,symbol:string,price:double,quantity:int32,
                    traderID:int32"

right input schema: "tID*:int32,name:string"
```

結合ウィンドウのコードは次のようになります。

```
<window-join name='myJoinWindow' index='pi_RBTREE'>
  <join type='leftouter'>
      <conditions>
       <fields left='traderID' right='tID'/>
      </conditions>
  </join>
 <output>
      <field-selection name='sym' source='l_symbol'/>
      <field-selection name='price' source='l_price'/>
      <field-selection name='tID' source='l_traderID'/>
      <field-selection name='traderName' source='r_name'/>
 </output>
</window-join>
```

次の点に注意してください。

- 結合条件は、次の形式をとります。左右のイベントのどのフィールドを使用して一致を生成するかを指定します。

"l_*fieldname*=r_*fieldname*, ...,l_*fieldname*=r_*fieldname*"

- 結合選択は次の形式をとります。これは、結合ウィンドウによって生成されたイベントに含まれるキー以外のフィールドのリストを指定します。

  "{l|r}_*fieldname*, ...{l|r}_*fieldname*"

- フィールド署名は、次の形式をとります。これらは、出力イベントの非キーフィールドの名前とタイプを指定します。タイプは、結合選択で指定されたフィールドから推論できます。ただし、式またはユーザー記述関数（C++で）を使用する場合は、型指定を推論することはできませんので、必須です。

  "*fieldname*:*fieldtype*, ..., *fieldname*:*fieldtype*

キー以外のフィールド算出式を使用すると、コードは次のようになります。

```
<window-join name='myJoinWindow' index='pi_RBTREE'>
 <join type='leftouter'>
   <conditions>
     <fields left='traderID' right='tID'/>
   </conditions>
 </join>
 <output>
     <field-expr name='sym' type='string'>l_symbol</field-expr>
     <field-expr name='price' type='double'>l_price</field-expr>
     <field-expr name='tID' type='int32'>l_traderID</field-expr>
     <field-expr name='traderName' type='string'>r_name</field-expr>
 </output>
</window-join>
```

これは、非キーフィールドを結合する入力フィールドの 1 対 1 のマッピングを示します。任意の複雑な入力フィールドの組み合わせを使用して、非キー結合フィールドを生成するために算出式と関数を使用できます。

転置ウィンドウの使用

イベントを複数の列からなる行として概念化できます。転置ウィンドウを使用して、イベントの行を列に、またはその列を行に交換します。転置ウィンドウの属性を使用して、データの並べ替えを管理します。

転置ウィンドウに関連付けられている XML エレメントについては、“window-transpose” (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

転置ウィンドウは挿入イベントのみを受け入れ、インデックスは必ず **pi_EMPTY** になります。幅と長さの 2 つのモードがあります。

表 3.8　転置ウィンドウモード

モード	説明
幅	受信イベントごとに 1 つのイベントを生成します。

モード	説明
長さ	受信イベントごとに 1 つ以上のイベントを生成します。

飛行中の航空機のピッチ、ヨー、ロール、速度に関する情報を処理するためにワイドモードを使用しているとします。ソースウィンドウは次のスキーマを使用します。

```
<schema>
    <fields>
      <field name='ID'       type='int64' key='true'/>
      <field name='PlaneID'  type='string'/> <!-- 1 -->
      <field name='TAG'      type='string'/> <!-- 2 -->
      <field name='value'    type='double'/> <!-- 3 -->
      <field name='time'     type='stamp'/>
      <field name='lat'      type='double'/> <!-- 4 -->
      <field name='long'     type='double'/>
    </fields>
</schema>
```

1 スキーマの PlaneID フィールドの値は、どの航空機がデータを提供するかを識別します。

2 TAG フィールドの値は、データに航空機のピッチ、ヨー、ロール、速度のいずれかが含まれるかどうかを指定します。TAG フィールドの種類は文字列(string)でなければなりません。

3 value フィールドは、TAG の値およびデータが記録される特定の time を記録します。

4 ピッチ、ヨー、ロール、速度のいずれかが記録されている場合、イベントは航空機の緯度(lat)と経度(long)を取得します。

ここで、次のイベントがソースウィンドウからストリーミングされるとします。

```
i,n,1,turboprop #1, pitch,1.1, 2017-08-30 15:21:00.000000,10,10
i,n,2,turboprop #2, velocity,-1.2, 2017-08-30 15:21:00.000001,20,20
i,n,3,turboprop #1, roll,-1.1, 2017-08-30 15:21:00.000002,11,11
i,n,4,turboprop #2, yaw,2.2, 2017-08-30 15:21:00.000003,21,21
i,n,5,turboprop #2, pitch,1.2, 2017-08-30 15:21:00.000004,22,22
i,n,6,turboprop #1, velocity,-1.1, 2017-08-30 15:21:00.000005,12,12
i,n,7,turboprop #2, roll,-1.2, 2017-08-30 15:21:00.000006,23,23
i,n,8,turboprop #1, yaw,2.1, 2017-08-30 15:21:00.000007,13,13
i,n,9,jet #1, pitch,1.3, 2017-08-30 15:21:00.000012,30,30
i,n,10,jet #2, velocity,-4.4, 2017-08-30 15:21:00.000013,40,40
i,n,11,jet #1, roll,-4.3, 2017-08-30 15:21:00.000014,31,31
i,n,12,jet #2, yaw,8.4, 2017-08-30 15:21:00.000015,41,41
i,n,13,jet #2, pitch,4.4, 2017-08-30 15:21:00.000016,42,42
i,n,14,jet #1, velocity,-4.3, 2017-08-30 15:21:00.000017,32,32
i,n,15,jet #2, roll,-4.4, 2017-08-30 15:21:00.000018,43,43
i,n,16,jet #1, yaw,8.3, 2017-08-30 15:21:00.000019,33,33
i,n,17,turboprop #1, pitch,23.1, 2017-08-30 15:21:06.000022,14,14
```

これらの 17 件のイベントでは、**turboprop #1**、**turboprop #2**、**jet #1**、**jet #2** の 4 つのプレーンがソースウィンドウからデータをストリーミングします。ストリーミングする各イベントには、特定の時間におけるピッチ、速度、ロール、ヨーのいずれかの値が含まれています。

ここで、転置ウィンドウを介して入力データをストリーミングするとします。mode='wide'を使用して、特定の時間における各航空機のピッチ、速度、ロール、ヨーを含む単一のイベント(行)を作成します。

```
<window-transpose name='transposeW'
                  mode='wide' <!-- 1 -->
                  tag-name='TAG'<!-- 2 -->
                  tags-included='pitch,yaw,roll,velocity' <!-- 3 -->
                  tag-values='value,time' <!-- 4 -->
                  clear-timeout='never'
                  group-by='PlaneID'> <!-- 5 -->
   <connectors>
    ...
   </connectors>
</window-transpose>
```

1 wide モードを使用して、複数の列を含む出力イベントを生成します。各列は、入力イベントを通じてストリーミングされたデータに基づいています。

2 入力イベントの TAG の値は、出力イベントの列を決定します。

3 Tags-included は、出力イベントに含める TAG の特定の値を指定します。ここでは、TAG の 4 つの値すべてが指定されています。

4 ピッチ、ヨー、ロール、速度に関連付けられている value および time は出力イベントに含まれます。関連付けられている緯度と経度は渡され、含まれません。

5 出力イベントは PlaneID の値でグループ化されています。

出力列は、次の外積を取ることによって形成されます。

{pitch、yaw、roll、velocity} x {value、time}

これは pitch_value、pitch_time、yaw_value、yaw_time などとなります。

ソースウィンドウから 4 つのイベントを処理した後に、転置ウィンドウからストリーミングされる最初の 4 つのイベントは次のとおりです。

```
<event opcode="insert" window="newproject/cq/transposeW">
 <value name="ID">1</value>
 <value name="PlaneID">turboprop #1</value>
 <value name="lat">10.000000</value>
 <value name="long">10.000000</value>
 <value name="pitch_time">1504106460000000</value>
 <value name="pitch_value">1.100000</value>
</event> <!-- 1 -->
<event opcode="insert" window="newproject/cq/transposeW">
 <value name="ID">2</value>
 <value name="PlaneID">turboprop #2</value>
 <value name="lat">20.000000</value>
 <value name="long">20.000000</value>
 <value name="velocity_time">1504106460000001</value>
 <value name="velocity_value">-1.200000</value>
</event> <!-- 2 -->
$ <event opcode="insert" window="newproject/cq/transposeW">
 <value name="ID">3</value>
 <value name="PlaneID">turboprop #1</value>
 <value name="lat">11.000000</value>
 <value name="long">11.000000</value>
 <value name="pitch_time">1504106460000000</value>
```

```
 <value name="pitch_value">1.100000</value>
 <value name="roll_time">1504106460000002</value>
 <value name="roll_value">-1.100000</value>
</event> <!-- 3 -->
<event opcode="insert" window="newproject/cq/transposeW">
 <value name="ID">4</value>
 <value name="PlaneID">turboprop #2</value>
 <value name="lat">21.000000</value>
 <value name="long">21.000000</value>
 <value name="velocity_time">1504106460000001</value>
 <value name="velocity_value">-1.200000</value>
 <value name="yaw_time">1504106460000003</value>
 <value name="yaw_value">2.200000</value>
</event> <!-- 4 -->
```

1 最初の入力イベントには、**turboprop #1** のピッチ値 1.1 が含まれています。

この出力イベントとそれ以降のすべての出力イベントでは、転置ウィンドウは lat および long をそのまま渡します。

TAG の値は **pitch** であり、time および value は tags-included であるため、新しい変数 pitch_time および pitch_value が生成されます。pitch_time および pitch_value の値が挿入されます。

最初のイベントで処理する roll、velocity、または yaw の値はありません。

2 2 つ目の入力イベントには、**turboprop #2** の速度値-1.2 が含まれています。TAG の値は **velocity** であり、time および value は tags-included であるため、新しい変数 velocity_time および velocity_value が生成されます。

2 つ目のイベントで処理する pitch、roll、または yaw の値はありません。

3 3 つ目の入力イベントには、**turboprop #1** のロール値-1.1 が含まれています。

航空機の pitch_time および pitch_value は最初のイベントから保持されます。

TAG の値は **roll** であり、time および value は tags-included であるため、新しい変数 roll_time および roll_value が生成されます。

3 つ目のイベントで処理する velocity または yaw の値はありません。

4 4 つ目の入力イベントには、**turboprop #2** のヨー値 2.2 が含まれています。

航空機の velocity_time および velocity_value は 2 つ目のイベントから保持されます。

TAG の値は **yaw** であり、time および value は tags-included であるため、新しい変数 yaw_time および yaw_value が生成されます。

4 つ目のイベントで処理する pitch または roll の値はありません。

転置ウィンドウはこのようにして、航空機ごとに該当するあらゆる TAG 値を含むイベントを構築します。**PlaneID** は group-by の値であるためです。17 個すべての入力イベントが処理されるまでに、各航空機のピッチ、ロール、速度、ヨーを取得する 4 つのイベントがあります。

```
<event opcode="insert" window="newproject/cq/transposeW">
 <value name="ID">7</value>
 <value name="PlaneID">turboprop #2</value>
 <value name="lat">23.000000</value>
 <value name="long">23.000000</value>
 <value name="pitch_time">1504106460000004</value>
```

```
 <value name="pitch_value">1.200000</value>
 <value name="roll_time">1504106460000006</value>
 <value name="roll_value">-1.200000</value>
 <value name="velocity_time">1504106460000001</value>
 <value name="velocity_value">-1.200000</value>
 <value name="yaw_time">1504106460000003</value>
 <value name="yaw_value">2.200000</value>
</event>
<event opcode="insert" window="newproject/cq/transposeW">
 <value name="ID">8</value>
 <value name="PlaneID">turboprop #1</value>
 <value name="lat">13.000000</value>
 <value name="long">13.000000</value>
 <value name="pitch_time">1504106460000000</value>
 <value name="pitch_value">1.100000</value>
 <value name="roll_time">1504106460000002</value>
 <value name="roll_value">-1.100000</value>
 <value name="velocity_time">1504106460000005</value>
 <value name="velocity_value">-1.100000</value>
 <value name="yaw_time">1504106460000007</value>
 <value name="yaw_value">2.100000</value>
</event>
<event opcode="insert" window="newproject/cq/transposeW">
 <value name="ID">15</value>
 <value name="PlaneID">jet #2</value>
 <value name="lat">43.000000</value>
 <value name="long">43.000000</value>
 <value name="pitch_time">1504106460000016</value>
 <value name="pitch_value">4.400000</value>
 <value name="roll_time">1504106460000018</value>
 <value name="roll_value">-4.400000</value>
 <value name="velocity_time">1504106460000013</value>
 <value name="velocity_value">-4.400000</value>
 <value name="yaw_time">1504106460000015</value>
 <value name="yaw_value">8.400000</value>
</event>
<event opcode="insert" window="newproject/cq/transposeW">
 <value name="ID">16</value>
 <value name="PlaneID">jet #1</value>
 <value name="lat">33.000000</value>
 <value name="long">33.000000</value>
 <value name="pitch_time">1504106460000012</value>
 <value name="pitch_value">1.300000</value>
 <value name="roll_time">1504106460000014</value>
 <value name="roll_value">-4.300000</value>
 <value name="velocity_time">1504106460000017</value>
 <value name="velocity_value">-4.300000</value>
 <value name="yaw_time">1504106460000019</value>
 <value name="yaw_value">8.300000</value>
</event>
```

最後の入力イベントは、**turboprop #1** のピッチを更新します。これは、time の後の値で収集されました。

```
<event opcode="insert" window="newproject/cq/transposeW">
 <value name="ID">17</value>
 <value name="PlaneID">turboprop #1</value>
```

```
  <value name="lat">14.000000</value>
  <value name="long">14.000000</value>
  <value name="pitch_time">1504106466000022</value>
  <value name="pitch_value">23.100000</value>
  <value name="roll_time">1504106460000002</value>
  <value name="roll_value">-1.100000</value>
  <value name="velocity_time">1504106460000005</value>
  <value name="velocity_value">-1.100000</value>
  <value name="yaw_time">1504106460000007</value>
  <value name="yaw_value">2.100000</value>
</event>
```

後続の入力イベントは、各航空機の出力イベントを更新し続けます。

入力イベント値を受け取らない限り、転置ウィンドウの clear-timeout 属性を使用して、出力イベント値がクリアされるまでの時間を指定します。

入力データが複数のデバイスまたは機器から到着しないとします(前述の例の 2 つの航空機からなど)。その場合、group-by 属性を使用する必要はありません。

たとえば、次の入力データを考えてみましょう。ソースウィンドウは以前と同じスキーマを使用しますが、PlaneID は使用しません。

```
i,n,1,velocity, 234.0
i,n,2,roll,2.3
i,n,3,pitch,3.4
i,n,4,yaw,-1.1
```

次の属性で転置ウィンドウを指定して、このデータを転置できます。

```
<window-transpose name='transposeW'
      mode='wide'
      tag-name='TAG'
      tags-included='pitch,yaw,roll,velocity'
      tag-values='value'
      clear-timeout='never'
/>
```

> 重要　ワイドモードの転置ウィンドウでは、group-by フィールドを使用してイベントをグループ化します。これは group-by フィールドのカーディナリティが制限されていることを必要とします。フィールドが制限されていない場合、ウィンドウは制限がない状態でメモリを消費し続けます。

長さモードを使用すると、ワイドモードの逆の結果が得られます。転置ウィンドウは、受け取った幅イベントごとに多数のイベントをストリーミングします。ソースウィンドウの入力スキーマはフィールドの組み合わせを反映する必要があります。

たとえば、このソースウィンドウのスキーマを考えます。

```
<schema>
          <fields>
            <field name='ID'         type='int64' key='true'/>
            <field name='PlaneID'    type='string'/>
            <field name='pitch_value' type='double'/>
            <field name='pitch_time'  type='stamp'/>
            <field name='yaw_value'   type='double'/>
            <field name='yaw_time'    type='stamp'/>
```

```
    <field name='roll_value'  type='double'/>
    <field name='roll_time'   type='stamp'/>
    <field name='velocity_value' type='double'/>
    <field name='velocity_time'  type='stamp'/>
    <field name='lat'      type='double'/>
    <field name='long'     type='double'/>
   </fields>
  </schema>
```

そのソースウィンドウを介して幅イベントをストリーミングするとします。

```
I N 1 turboprop #2 1.2 21:00.0 2.2 21:00.0 -1.2 21:00.0 -1.2 21:00.0 20 20
```

下流の転置ウィンドウは、受信スキーマの tag-values 値と tags-included 値の組み合わせを検索します。

```
<window-transpose name='transposeL'
                  mode='long' tag-name='TAG'
                  tag-values='value,time'
                  tags-included='pitch,yaw,roll,velocity'>
...               </window-transpose>
```

その入力イベントを処理すると、転置ウィンドウは次の 4 つの出力イベントをストリーミングします。

```
I,N, 1,0,pitch,1.200000,2017-08-30 15:21:00.000004,turboprop #2,20.000000,20.000000
I,N, 1,1,yaw,2.200000,2017-08-30 15:21:00.000003,turboprop #2,20.000000,20.000000
I,N, 1,2,roll,-1.200000,2017-08-30 15:21:00.000006,turboprop #2,20.000000,20.000000
I,N, 1,3,velocity,-1.200000,2017-08-30 15:21:00.000001,turboprop #2,20.000000,20.000000
```

下流の転置ウィンドウで tag-values 値と tags-included 値の適切な組み合わせが見つからない場合、ウィンドウは完成時に失敗し、モデルの実行が許可されません。

コピーウィンドウの使用

コピーウィンドウの概要

コピーウィンドウを使用して、他のタイプの親ウィンドウをコピーします。特定のイベント状態保持ポリシーでイベントを保持すると便利です。保持ポリシーは、ソースウィンドウおよびコピーウィンドウでのみ設定できます。

コピーウィンドウに関連付けられている XML エレメントについては、“window-copy” (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

コピーウィンドウの保持ポリシー

コピーウィンドウのイベント状態の保持は、ウィンドウが挿入専用に指定されていない場合やウィンドウインデックスが設定されていない場合にのみ設定できます pi_EMPTY 。その後のすべての関連ウィンドウは、保持管理の影響を受けます。イベントは、ウィンドウ保持ポリシーを超えると削除されます。

保持ポリシーの詳細については、“保持の理解”を参照してください。

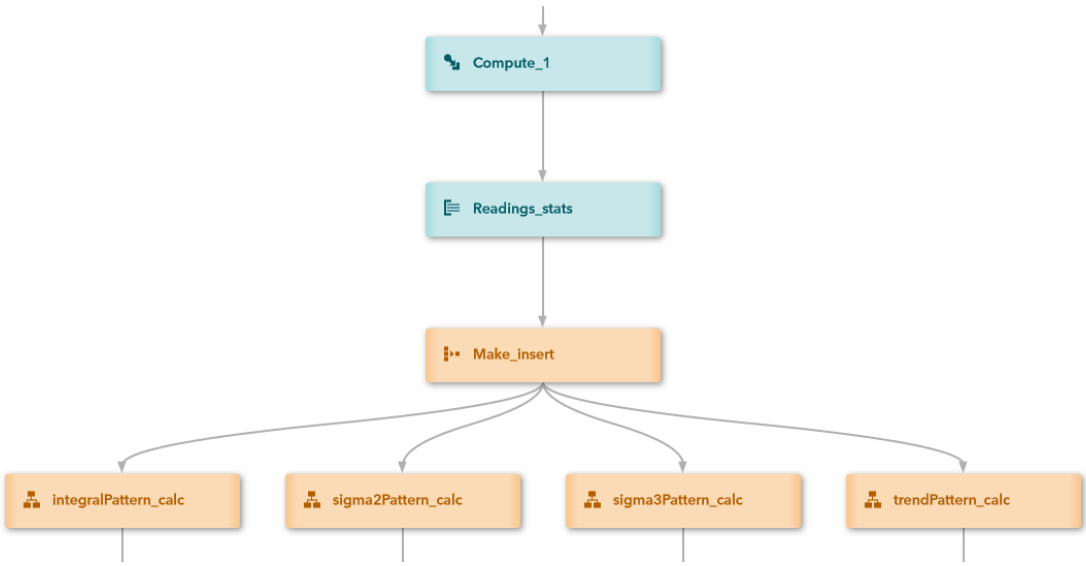
状態の削除ウィンドウの使用

モデルのステートフル部分からモデルのステートレス部分への移行を容易にするには、状態の削除ウィンドウを使用します。状態の削除ウィンドウは、受信したすべてのイベントを挿入に変換し、eventNumber という名前のフィールドを追加します。これは、単調増加の連続整数です。この追加フィールドは、状態の削除ウィンドウの唯一のキーです。

状態の削除ウィンドウに関連付けられている XML エレメントについては、“window-remove-state” (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

次のモデルを考えてみましょう。

図 3.2　ステートレスパターンウィンドウへストリーミングされるステートフルウィンドウ



計算ウィンドウ、Compute_1 はセンサーと位置データのストリームを受信します。受信データの操作に基づく新しいフィールドで受信イベントを補強します。

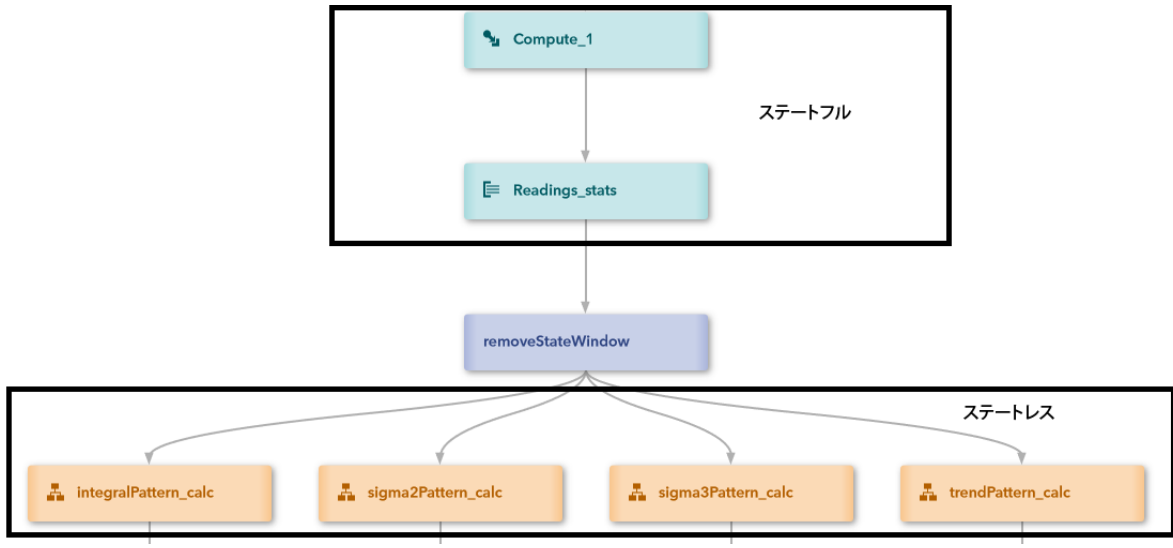
Compute_1 は、補強イベントを集計ウィンドウである Readings_stats にストリーミングします。Readings_stats は場所ごとにイベントをまとめ、そのグループの合計、平均、標準偏差などの値を算出します。特定の場所の新しいイベントが Reading_stats にストリーミングされるたびに、これらの算出値を含む更新イベントが Make_insert(算出ウィンドウ)にストリーミングされます。

Make_insert には、更新イベントを挿入イベントに変換する DS2 関数が含まれています。

Make_insert はこれらの挿入イベントを一連のパターンウィンドウにストリーミングします。パターンウィンドウは挿入イベントのみを受信します。これらのパターンウィンドウはさらにデータを処理します。

このモデルでは、更新イベントを挿入イベントに変換することが算出ウィンドウの唯一の目的です。Make_insert を状態の削除ウィンドウに置き換えても同じ結果が得られます。

図 3.3　状態の削除ウィンドウの使用



状態の削除ウィンドウスキーマの一般的な形式は次のとおりです。

eventNumber*:int64, [originalOC:*string*, originalFL:*string*,] <*incoming_schema*>

次の XML コードは一般的なケースを示しています。*incoming_schema* は次のとおりです。

symbol:*string*, true_test:*integer*

```
<window-remove-state name='removeStateWindow' add-log-fields='true' <!-- 1 -->
    remove='retentionUpdates retentionDeletes'><!-- 2 -->
</window-remove-state>
```

1　add-log-fields='true'を設定すると、スキーマで指定された originalOC フィールドと originalFL フィールドが追加されます。

2　演算コード、保持ポリシー、またはその 2 つの組み合わせでイベントをフィルター処理するように、状態の削除ウィンドウを構成できます。remove='retentionDeletes retentionUpdates'を設定すると、保持フラグが設定されている削除イベントと更新イベントを除外します。

この状態の削除ウィンドウからの出力例は次のとおりです。3 つのイベントがフィルター処理されます。

```
<event opcode="insert" window="project/contQuery/removeStateWindow">
 <value name="eventNumber">13</value> <!-- 1 -->
 <value name="originalFL">N</value> <!-- 2 -->
 <value name="originalOC">D</value> <!-- 3 -->
 <value name="symbol">id</value>
 <value name="true_test">21</value>
</event>
<event opcode="insert" window="project/contQuery/removeStateWindow">
 <value name="eventNumber">14</value>
 <value name="originalFL">N</value>
 <value name="originalOC">D</value>
 <value name="symbol">pd</value>
 <value name="true_test">23</value>
</event>
<event opcode="insert" window="project/contQuery/removeStateWindow">
 <value name="eventNumber">15</value>
 <value name="originalFL">N</value>
 <value name="originalOC">D</value>
 <value name="symbol">pd</value>
 <value name="true_test">23</value>
</event>
```

1 eventNumber は、状態の削除ウィンドウによって通過する各イベントに挿入されます。

2 イベント番号 13 の元のフラグは N(正常)でした。

3 イベント番号 13 の元の演算コードは D(削除)でした。

関数ウィンドウの使用

関数ウィンドウの概要

関数ウィンドウを使用して、イベントデータを操作または変換するためにエンティティを定義します。関数ウィンドウでは、スキーマを定義し、イベントデータに適用する*関数コンテキスト*を定義します。関数コンテキストの設定は 3 つの方法があります。

- 正規表現を指定します。
- コネクタ、関数、プロジェクトプロシジャウィンドウ、またはストリーミング分析ウィンドウのプロパティを指定します。
- サポート関数を含む関数を使用することができます。

関数ウィンドウに関連付けられている XML エレメントについては、次を参照してください。

- “window-functional” (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)
- “関数ウィンドウに関連する XML 言語要素” (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)

イベントの生成

XML エレメント<generate>を使用して、元のイベントを出力イベントに含めるかどうかを決定するブール関数または式を指定します。

- 関数または式が true と評価された場合、元のイベントは追加の出力イベントに含まれます。
- False と評価された場合、元のイベントは含まれません。

また、<event-loop>エレメントを使用して、単一の入力イベントから複数の出力イベントを生成できます。ある種類のデータを作成し、作成したデータから任意の数のエンティティを取得する関数を指定できます。これらのエンティティごとに、そのイベントループ固有の関数コンテキストを使用してイベントを生成できます。

または、Lua ウィンドウ内でイベントを生成することもできます。詳細については、Lua ウィンドウを使用して JSON からイベントを生成するを参照してください。

event-loops の使用

event-loops を使用すると、単一の入力イベントから任意の数のイベントを生成できます。任意の数の event-loops を指定できます。各ループは、特定の種類のデータを処理します。

各入力イベントについて、関数ウィンドウは各イベントループエントリに対して次のことを行います。

1. 関数または参照を使用して、ループへの入力として使用されるデータを生成します。たとえば、event-loop-xml ループでは、use-xml エレメントを使用して有効な XML を生成します。このコンテンツは、関数またはウィンドウの function-context 内のプロパティへの参照です。
2. 0 以上のエンティティを取得するために、XPATH や JSON などの適切な式をデータに適用します。
3. これらのエンティティごとに、データ属性で指定されたデータ項目をエンティティの文字列値に設定します。次に、function-context 内の関数が実行され、イベントが生成されます。ウィンドウの関数コンテキスト内のプロパティまたはイベント値は、ループの関数コンテキストからアクセス可能です。また、データ属性で指定された変数は '$'表記でアクセスできます。

イベントループは各繰り返しに対してコンテキスト XML オブジェクトを作成します。関数内で**#_context** としてこのオブジェクトに参照できます。このオブジェク

トの名前空間は最上位コンテナーの名前空間に設定されます。**#_context** オブジェクトの代わりに文字列表現を使用すると、名前空間を設定できません。

関数コンテキストの理解と使用

式の指定

<expressions>エレメントを使用して、1 度コンパイルされた POSIX 正規式を指定します。

```
<expressions>
   <expression name='expname'>[posix_regular_expression]</expression>
   ...
 </expressions>
```

正規表現を指定した後、次の表記法と rgx サポート関数を使用して関数内から正規表現を参照できます。

```
<function name='myData'>rgx(#expname,$inputField,1)</function>
```

注: POSIX 正規式は、IEEE で指定された標準に準拠していなければなりません。

URI を含むデータフィールドを取得していて、その URI からプロトコルを抽出するとします。<function name='protocol'>rgx('(.*):',$uri,1)</function>を使用すると、関数が実行されるたびに正規式がコンパイルされます。ただし、次のコードを使用する場合

```
<expressions>
  <expression name='getProtocol'>(.*):</expression>
</expressions>

<function name='protocol'>rgx(#getProtocol,$uri,1)</function>
```

式は 1 回コンパイルされ、関数が実行されるたびに使用されます。

プロパティの指定

プロパティは'#'表記法#[property-*type*]を使用して関数内から参照されるという点で式と似ています。

5 種類のプロパティがあります。

- map は、名前による値検索に使用される名前と値のペアのマップを生成する関数を実行します。
- set は、値ルックアップに使用する一連の文字列を生成する関数を実行します。
- XML は、XPATH クエリに使用できる XML オブジェクトを生成する関数を実行します。

- JSON は、JSON 検索に使用できる JSON オブジェクトを生成する関数を実行します。
- string は関数内で一般的に使用される文字列を生成する関数を実行します。
- list はインデックス付きアクセスに使用する文字列のリストを生成する関数を実行します。

各プロパティは関数を使用して生成されます。これらの関数は、XML で前に定義されたプロパティを参照できます。

表 3.9 各プロパティタイプのコーディング方法

プロパティの種類	説明
<property-map name='*name*' outer='*outdelim*' inner='*indelim*'>[*code*]</property-map>	■ name-プロパティの名前 ■ outer-データの解析に使用する外部区切り文字 ■ inner-データの解析に使用する内部区切り文字 ■ *code*-名前値マップに解析されるデータを生成するために実行する関数 たとえば、firstname=joe;lastname=smith;occupation=software のような入力フィールド data が存在するとします。 次の property-map を作成できます。<property-map name='myMap' outer=';' inner='='>$data</property-map>
<property-xml name='*name*'>[*code*]</property-xml>	■ name-プロパティの名前 ■ *code*-有効な XML を生成するために実行する関数
<property-json name='*name*'>[*code*]</property-json>	■ name-プロパティの名前 ■ *code*-有効な JSON を生成するために実行する関数
<property-string name='*name*'>[*code*]</property-string>	■ name-プロパティの名前 ■ *code*-文字列値を生成するために実行する関数
<property-set name='*name*' delimiter='*delim*'>[*code*]</property-set>	■ name-プロパティの名前 ■ delimiter-データの解析に使用する区切り文字 ■ *code*-値セットに解析されるデータを生成するために実行する関数

プロパティの種類	説明
	たとえば、次のような入力フィールドデータが存在するとします。ibm,sas,oracle。これにより、次のプロパティの設定が生成されます。<property-set name='mySet' delimiter=','>$data</property-set>

ある従業員の情報がモデルにストリーミングされているとします。

```
<event>
  <value name='map'>name:[employee name];
       position:[employee position]
  </value>
  <value name='developerInfo'>
     <![CDATA[<info>this is developer info</info>]]>
  </value>
  <value name='managerInfo'>
     <![CDATA[<info>this is manager info</info>]]>
  </value>
</event>
```

従業員データを格納するための property-map を作成し、次に位置フィールドを調べて、適切なデータを含む property-xml を作成することができます。従業員が開発者の場合、XML は developerInfo からのものです。それ以外の場合は、managerInfo を使用します。function-context は if、equals、mapValue、xpath の各サポート関数を使用して関数がコード化されているようになります。

```
<function-context>
  <properties>
    <property-map name='myMap' outer=';' inner=':'>$map</property-map>
    <property-xml name='myXml'>if(equals(mapValue(#myMap,'position'),'developer'),
     $developerInfo,$managerInfo)
    </property-xml>
  </properties>
  <functions>
    <function name='employee'>mapValue(#myMap,'name')</function>
    <function name='info'>xpath(#myXml,'text()')</function>
  </functions>
</function-context>
```

次のイベントでストリーミング

```
<event>
  <value name='map'>
    name:curly;position:developer<
  /value>
  <value name='developerInfo'>
    <![CDATA[<info>this is developer info</info>]]>
  </value>
  <value name='managerInfo'>
    <![CDATA[<info>this is manager info</info>]]>
  </value>
</event>
```

```
<event>
  <value name='map'>name:moe;position:manager</value>
  <value name='developerInfo'><![CDATA[<info>this is developer info</info>]]></value>
  <value name='managerInfo'><![CDATA[<info>this is manager info</info>]]></value>
</event>
```

次の結果が得られます。

```
<event opcode='insert' window='project/query/transform'>
  <value name='employee'>curly</value>
  <value name='id'>fd26bf36-3d65-4d17-8dc6-317409bbf5b6</value>
  <value name='info'>this is developer info</value>
</event>

<event opcode='insert' window='project/query/transform'>
  <value name='employee'>moe</value>
  <value name='id'>84c56bb7-9f3c-4cb8-93a5-8dc2f75d353b</value>
  <value name='info'>this is manager info</value>
</event>
```

関数の使用

関数を使用すると、関数ウィンドウはイベントを処理して、スキーマの各フィールドに対応する名前の関数を探します。関数が存在する場合は、ウィンドウはそれを実行します。結果の値は出力イベントに入力されます。フィールドに関数が指定されておらず、同じ名前のフィールドが入力スキーマに存在する場合、入力値は出力イベントに直接コピーされます。

関数コンテキスト内では、次の 2 種類の関数を使用できます。

- 一般的な関数
- イベントストリーム処理に固有の関数

$[*name of field*]の'$'表記を使用して、入力イベントまたは出力イベントのいずれかでイベントフィールドを参照できます。

表 3.10 関数コンテキストで関数を使用することに関連するデータマッピング

string	ESP_UTF8STR
float	ESP_DOUBLE
long	ESP_INT64
integer	ESP_INT32
Boolean	ESP_INT32

たとえば、入力イベントに name フィールドがあり、出力イベントに occupation フィールドを生成するとします。ifNext サポート関数と equals サポート関数を使用して、次のように関数をコーディングできます。

```
<function name='occupation'>
ifNext
(
  equals($name,'larry'),'plumber',
  equals($name,'moe'),'electrician',
  equals($name,'curly'),'carpenter'
)
</function>
```

また、出力イベントのフィールドを参照することもできます。この例を続けると、再度 ifNext サポート関数と equals サポート関数を使用して、occupation 値に応じて hourlyWage フィールドを出力イベントに追加できます。

```
<function name='hourlyWage'>
ifNext
(
  equals($occupation,'plumber'),85.0,
  equals($occupation','electrician'),110.0,
  equals($occupation,'carpenter'),60.0
)
</function>
```

注: 関数を定義する場合は、フィールドの順序に注意することが重要です。関数が出力イベントフィールドを参照する場合、参照フィールドの前にそのフィールドを計算する必要があります。

関数ウィンドウでイベントフィールドの参照が表示されると、最初に出力イベント(生成されているイベント)で値が検索されます。このイベントに値が見つからない場合、関数は入力イベントで値を検索します。出力イベントの値は、XML コードに表示される順序で生成されるため、別のフィールドの後に定義された任意のフィールドが、関数内の値を参照できます。

Lua ウィンドウの使用

Lua ウィンドウの概要

Lua ウィンドウを使用すると、Lua プログラムで SAS Event Stream Processing イベントを生成できます。Lua ウィンドウでは、SAS Event Stream Processing 入力イベントをテーブルで表現します。1 つの入力イベントから複数の出力イベントを生成するには、それらのイベントをインデックス付きの配列にします。

ウィンドウをデザインしながら、Lua コードをデバッグすることができます。Luaprint()関数は、コンソールに出力を表示します。この関数をコード内で使用した場合、*ESP サーバー*のコンソールウィンドウに出力が表示されます。

Lua ウィンドウから使用できる追加の Lua 機能については、8 章, “共通の Lua 機能”を参照してください。

重要　Lua コードには、次の 2 つの例外を除き、標準 Lua ライブラリで提供される関数を含めることができます:

- os.execute
- io.popen

これらの関数のいずれかを使用すると実行時エラーが発生し、プロジェクトをロードできなくなります。

Lua ウィンドウの構成

Lua ウィンドウを指定するサンプル XML コードを次に示します。

```
<window-lua name="name" index="index" events="create_function" init=""
destroy="" heartbeat="hb_function" process-blocks="" encode-binary=""
on-script-exception="log-and-continue | stop-and-remove"> <!-- 1 -->
 <schema> <!-- 2 -->
  <fields>
   <field name="name" type="type" key="key"/>
   ...
  </fields>
 </schema>
 <copy exclude="true|false">...</copy> <!-- 3 -->
 <use exclude="true|false">...</use> <!-- 4 -->
 <code><![CDATA[
  function create_function(data,context) -- 5
     local   e = {}
     ...
     return(e)
  end

  function hb_function(context) -- 6
     ...
  end

 ]]></code>
 <forwarding suppress="true|false"> 7
   <destinations>
     <destination window="window" contquery="contquery" blocksize="blocksize"/>
     ...
   </destinations>
 </forwarding>   </window-lua>
```

1　トップレベルのエレメントである<window-lua>は、これが Lua ウィンドウであることを示しています。

events 属性は、code エレメント内で指定するイベント作成 Lua 関数の名前を指定します。この関数は、Lua ウィンドウが入力イベントを受信すると呼び出されます。

init 属性は、Lua ウィンドウの初期化時に呼び出される Lua 関数の名前を指定します。

destroy 属性は、Lua ウィンドウが破棄されたときに呼び出される Lua 関数の名前を指定します。この関数を使用して、実行する必要がある Lua クリーンアップタスクを実行できます。

heartbeat 属性は、code エレメント内で指定するハートビート Lua 関数の名前を指定します。この関数は、Lua ウィンドウがプロジェクトレベルのハートビートメッセージを受信したときに呼び出されます。この属性は必須ではありません。

process-blocks 属性は、イベントをブロックで処理するか個別に処理するかを示します。イベントブロックを処理するとき(process-blocks=true)、イベント作成関数は、ブロック内のイベントを表す Lua テーブルの配列を受け取ります。それ以外の場合、関数は単一のイベント(process-blocks=false、デフォルト)を受け取ります。

encode-binary 属性は、イベント配信用にバイナリデータを Base64 エンコードするかどうかを示します。デフォルト値は **false** です。

on-script-exception 属性は、ウィンドウコードの実行中に発生する例外の処理方法を指定するオプション属性です。log-and-continue に設定すると、エラーがログに記録され、実行が続行されます。stop-and-remove に設定すると、致命的なエラーがログに記録され、プロジェクトが停止し削除されます。この値が指定されていない場合、ウィンドウは含まれるプロジェクトで定義されたスクリプト処理のビヘイビアーを継承します。

2. Lua ウィンドウのスキーマを指定しなければなりません。
3. copy エレメントを含めると、指定したフィールドが入力イベントから出力イベントにコピーされます。exclude=true 属性を使って、コピーされないフィールドを指定します。空の copy エレメントを含めると、すべてのフィールドが出力イベントにコピーされます。
4. use 要素を含めると、入力イベントから指定されたファイルは *ESP サーバー*から Lua コードに渡されます。exclude=true 属性を使って、渡されないフィールドを指定します。空の use エレメントを含めると、Lua コードにフィールドは渡されません。
5. ウィンドウが入力イベントを取得したときに呼び出すイベント作成関数を指定します。
6. ウィンドウがプロジェクトレベルのハートビートメッセージを受信したときに呼び出すハートビート関数を指定します。
7. forwarding 要素を含める場合、イベントが生成され、指定された宛先に転送されます。suppress=true 属性を使用して、イベント転送ウィンドウでイベントの生成を防止します。詳細については、“イベント転送”を参照してください。

コード要素には、ウィンドウが所有するコードを含めることも、Lua スニペットに存在する共有コードを指すこともできます。コードがウィンドウによって所有されている場合は、インラインまたは外部ファイルのいずれかに置くことができます。外部ファイルにある場合は、ファイル属性を使用してそのファイルを参照できます。

```
<code file="mycode.lua" />

<code snippet="mysharedcode"/>
```

注: Lua スニペットまたは外部ファイルを参照する場合、code 要素にはコードを含めることはできません。

Lua ウィンドウがスニペット内で共有コードを使用する場合、そのスニペットを指す他の Lua エンティティとすべてのデータ 、関数、および変数が共有されます。共有コードは、現在それを使用しているエンティティによって 常にロックされます。

Lua テーブルのパラメーターは次のとおりです。

- data- ウィンドウがイベントブロックを処理している場合、このパラメーターはブロック内のイベントを表す Lua テーブルの配列を指定します。それ以外の場合は、単一のイベントを表す単一の Lua テーブルを指定します。どちらの場合も、入力イベントには use で指定されたフィールド、または use が指定されていない場合はすべてのフィールドが 含まれます。
- context - 現在のイベントのコンテキストデータ。有効なキーは次のとおりです。
  - window - 含まれている Lua ウィンドウのウィンドウ ID。
  - input - このイベントの入力ウィンドウの名前。
  - counter - Lua ウィンドウのイベントカウント。

ハートビート関数の context パラメーターは、入力値を指定しないことを除いて、イベント作成関数に渡されるものと同じです。イベント関数と同様に、ハートビート関数を使用してイベントを作成できます。

イベントの生成

シングルイベントの生成

入力イベントごとに 1 つの出力イベントを生成したい場合を考えてみましょう。次のソースウィンドウを考えてみましょう。

```
<window-source name="source" insert-only="true">
 <schema>
  <fields>
   <field name="id" type="string" key="true"/>
   <field name="string" type="string"/>
   <field name="number" type="int32"/>
   <field name="array" type="array(i32)"/>
  </fields>
 </schema>
</window-source>
```

ソースウィンドウと次の Lua ウィンドウの間にエッジがある状態で、次の Lua ウィンドウは次のタスクを実行します。

- 入力イベントの id を出力イベントにコピーします。
- string フィールドを二重引用符で囲みます。
- number の値を 3 倍します。
- array の各値を 2 倍します。

```
<window-lua name="lua" events="create">
 <schema>
```

```
  <fields>
   <field name="id" type="string" key="true"/>
   <field name="new_string" type="string"/>
   <field name="new_number" type="int32"/>
   <field name="new_array" type="array(i32)"/>
  </fields>
 </schema>
 <copy>id</copy> <!-- 1 -->
 <code><![CDATA[

  function create(data,context)
     local   e = {} -- 2
     e.new_string = "\""..data.id.."\"" -- 3
     e.new_number = data.number * 3 -- 4
     e.new_array = {} -- 5
     for index,value in ipairs(data.array) do
        e.new_array[index] = value * 2 -- 6
     end
     return(e) -- 7
  end

 ]]></code>
</window-lua>
```

1 copy エレメントは、入力イベントから出力イベントへの id を指定します。

2 このコード行は、出力イベントデータを保持する Lua オブジェクトを作成します。

3 出力フィールドを Lua オブジェクトに追加する必要があります。new_string フィールドは、このコード行で生成されます。なお、Lua の連結演算子は'..'です。

4 new_number フィールドは、入力フィールド番号に 3 を掛けて生成されます。

5 new_array フィールドは配列フィールドです。このコード行は、Lua のテーブルを作成します。

6 これらの行は、入力配列の値を繰り返し処理し、それぞれの値に 2 を掛けて、新しい値を設定します。

7 このコード行は、作成された Lua オブジェクトを返します。

作成関数が nil を返した場合、イベントは作成されません。

たとえば、次のコードでは、入力された数字フィールドが 10 未満の場合、イベントを作成しません。

```
function create(data,context)
   if (data.number < 10) then
      return(nil)
   end
   ...
end
```

次の入力イベントがあるとします。

```
i,n,1,string data,33,[11;24;55]
```

前述コードの出力イベントは次のようになります。

```
<event opcode="insert" window="p/cq/lua">
 <value name="id">1</value>
```

```
 <value name="new_array">[22;48;110]</value>
 <value name="new_number">99</value>
 <value name="new_string">"string data"</value>
</event>
```

複数のイベントの生成

複数のイベントを生成するには、インデックス付きの Lua テーブルを作成して返します。インデックス付きテーブルには、0 個以上の Lua テーブルが含まれています。各テーブルは、一連の出力イベントを表しています。

たとえば、次のコードを考えてみましょう。

```
function create(data,context)
  local  events = {}

  for i=1,5 do
    local  event = {}
    -- Add event data here
    events[i] = event
  end
  return(events)
end
```

次の Lua ウィンドウでは、このコードを使って、配列の各エントリのイベントを生成しています。

```
<window-lua name="lua" events="create">
 <schema>
  <fields>
   <field name="id" type="string" key="true"/>
   <field name="new_string" type="string"/>
   <field name="new_number" type="int32"/>
   <field name="array_entry" type="int32"/> <!-- 1 -->
  </fields>
 </schema>
 <code><![CDATA[

  local  eventId = 1

  function create(data,context)
    local  events = {}

    for index,value in ipairs(data.array) do
      local  e = {}

      e.id = esp_toString(eventId)
      eventId = eventId + 1 -- 2

      e.new_string = "\""..data.string.."\""
      e.new_number = data.number * 3
      e.new_array = {}
      e.array_entry = value * 2

      events[index] = e
```

```
    end

    return(events)
  end

 ]]></code>
</window-lua>
```

1 この行を実行すると、配列ではなく単一のフィールドエントリを持つ出力イベントになります。

2 これらの行では、インデックス付きの配列を作成した後、入力された配列フィールドを繰り返し処理し、各エントリに対して個別のイベントを生成します。各イベントに一意の ID を生成するために、eventId という名前のローカル変数を作成し、それを使って出力イベント ID を設定し、使用後は毎回増分します。

eventID 変数は、Lua コードの呼び出しの間、その値を維持するので、出力イベントのイベント ID が一意に増分していきます。このように、変数の値を使って Lua ウィンドウの状態を維持することができます。

イベントの演算コードの割り当て

デフォルトでは、Lua ウィンドウは、着信イベントの演算コードを、発信イベントまたはイベントセットの演算コードに割り当てます。また、_opcode フィールドを使用して、別の演算コードを発信イベントまたはイベントセットに設定することもできます。

```
<code><![CDATA[

  function create(data,context)
    local   e = {}
    e.new_string = "\""..data.id.."\""
    e.new_number = data.number * 3
    e.new_array = {}
    for index,value in ipairs(data.array) do
      e.new_array[index] = value * 2
    end
    e._opcode = "upsert"
    return(e)
  end

 ]]></code>
```

イベント転送

イベントをより効率的にパブリッシュするには、モデルを介したイベントのフローを制御するイベントの転送を使用できます。たとえば、複数の大きなイベントを含むイベントブロックがある場合、blocksize の値を **1** に設定してブロックを分割し、イベントを個別に処理できます。

たとえば、次のコードを考えてみましょう。

```
<forwarding suppress="false"> 1
 <destinations>
    <destination window="sourcewin" contquery="cq_1" blocksize="5" /> 2
 </destinations>
</forwarding>
```

1 この行により、Lua ウィンドウでのイベントの転送が可能になり、イベント転送ウィンドウで確実にイベントが生成されます。

2 この行は、イベントを転送するソースウィンドウ、ソースウィンドウを含む連続クエリの名前、および各イベントブロックに含めるイベントの数を指定します。

Lua ウィンドウでの RESTful API の使用

概要

次のタスクを実行するために、RESTful API を使用して Lua ウィンドウと通信できます。

- Lua コードで定義されている関数を呼び出す
- ウィンドウ内の Lua コードを追加または変更する
- Lua コードの検証

Lua 関数の呼び出し

次の PUT リクエストを使用して、Lua ウィンドウ内で関数を呼び出すことができます。

```
PUT /windows/project/contquery/window/code?op=run&function=<function_name>
```

トークンをイベントに入れる次のコードを考えてみましょう。

```
local   token = "a_token"

function set_token(data)
   token = data.token
   local   status = {}
   status.code = 0
   status.message = "token has been set to "..token
   return(status)
end
```

set_token 関数を呼び出して、トークンの値を変更できます。ウィンドウが **p/cq/lua** であるとします。次の呼び出しでトークンを設定できます。

```
PUT /windows/p/cq/lua/code?op=run&function=set_token
```

次のリクエストの本文を **data.json** という名前のファイルに入れるとします。

```
{
   "token":"a_token"
}
```

完全なリクエストとレスポンスは次のようになります。

```
$ curl "http://{INGRESS_URL}/eventStreamProcessing/v1/windows/p/cq/lua/code?
op=run&function=set_token" --data @data.json -X PUT

{
   "code":"0",
   "message":"token has been set to a_token"
}
```

*ESP サーバー*は、コードによって返された Lua オブジェクトを JSON に変換し、そのオブジェクトをクライアントに送り返します。この呼び出しの後、イベントの作成でトークンの新しい値を使用できます。

Lua コードの設定

PUT リクエストを使用して、Lua ウィンドウ内で実行されるコードを変更できます。

```
PUT /windows/project/contquery/window/code?op=load
```

数字の ID だけのソースウィンドウがあったとします。

```
<window-source name="s" autogen-key="true" insert-only="true">
 <schema>
  <fields>
   <field name="id" type="int64" key="true"/>
  </fields>
 </schema>
</window-source>
```

ID を Lua ウィンドウにフィードし、その ID と数値 5 の合計を算出し、その結果を out_value という名前の数値フィールドに入れたいとします。

```
<window-lua name="lua" events="create">
 <schema-string>id*:int64,out_value:int32</schema-string>
 <code><![CDATA[

  function create(data,context)
     local   event = {}
     event.id = data.id
     event.out_value = data.id + 5
     return(event)
  end

 ]]></code>
</window-lua>
```

イベントは次のコードのようになります。

```
<event opcode="insert" window="p/cq/lua">
  <value name="id">12</value>
  <value name="out_value">17</value>
</event>
```

ここで、何かが変わって、ID に 5 ではなく 10 を加えなければならなくなったとします。新しい Lua コードを含む **code.lua** というファイルを作成します。

```
function create(data,context)
  local  event = {}
  event.id = data.id
  event.out_value = data.id + 10
  return(event)
end
```

RESTful API を介してコードをロードします。

```
$ curl "http://{INGRESS_URL}/eventStreamProcessing/v1/windows/p/cq/lua/code?op=load"
-X PUT --data-binary @code.lua

<response>
 <message>Lua code loaded</message>
</response>
```

この後、イベントは次のようになります。

```
<event opcode="insert" window="p/cq/lua">
  <value name="id">13</value>
  <value name="out_value">23</value>
</event>
```

Lua コードの検証

次の POST リクエストを使用して、*ESP サーバー*を一般的な Lua コード検証ツールとして使用することもできます。

```
POST /luaValidationResults
```

検証したい Lua コードがある場合は、それをリクエストの本文で送信し、結果を受け取ります。次のコードを記述し、それを **code.lua** という名前のファイルに入れるとします。

```
this is not valid Lua code

function create(data,context)
  local  event = {}
  event.id = data.id
  event.out_value = data.id + 10
  return(event)
end
```

そして、コンソールで次のコマンドを実行します。

```
$ curl "http://{INGRESS_URL}/eventStreamProcessing/v1/luaValidationResults" -X POST --
data-binary @code.lua
```

```
<error errorCode="-1" httpStatusCode="404">
  <message>[string "test"]:2: syntax error near 'is'</message>
  <version>1</version>
</error>
```

無効な行を削除してから、コンソールでコマンドを再実行します。

```
<response>
  <message>Lua code validated successfully</message>
</response>
```

Python ウィンドウの使用

Python ウィンドウの概要

Python ウィンドウを使用すると、SAS Event Stream Processing イベントを作成する関数を Python プログラムで作成できます。ESP サーバーは、Python オブジェクトを通じて Python コードと通信します。Python ウィンドウ内では、Python プログラムの実行はシングルスレッドで行われます。

Python コードを使用する場合は、SAS Micro Analytic Service モジュールの代わりに Python ウィンドウを使用できます。Python ウィンドウを使用すると、次の利点があります。

- SAS Micro Analytic Service モジュールを作成したり、算出ウィンドウで入力ハンドラーを指定したりする必要はありません。
- コードのデバッグは、SAS Micro Analytic Service モジュール内よりも Python ウィンドウ内での方が簡単です。
- BLOB を使用すると、Python ウィンドウのパフォーマンスが向上します。
- SAS Event Stream Processing Studio を使用して Python ウィンドウで Python コードを編集する場合、コードエディタを使用して Python コードを検証できます。

Python では、辞書はキーと値のペアの順序付けされていないコレクションを表す組み込みデータ型です。イベントは、キーがスキーマフィールドである Python 辞書として表されます。

ウィンドウをデザインしながら、Python コードをデバッグすることができます。Python の print()関数を使用して、ESP サーバーのコンソールウィンドウに出力を送信します。詳細については、“コンソールでのデバッグ”を参照してください。

注: Python ウィンドウを使用する場合、コードは Python 3.11 と互換性がある必要があります。

Python ウィンドウの構成

Python ウィンドウを指定するサンプル XML コードを次に示します。

```
<window-python name="py_window" index="pi_EMPTY" events="createEvent"
init="initalize" destroy="cleanup" heartbeat="hb"
process-blocks="false" encode-binary="false"
on-script-exception="log-and-continue | stop-and-remove"> <!-- 1 -->
 <schema> <!-- 2 -->
  <fields>
   <field name="" type="" key=""/>
   ...
  </fields>
 </schema>
 <copy exclude="true|false">...</copy> <!-- 3 -->
 <use exclude="true|false" expand="true|false">...</use> <!-- 4 -->
 <code><![CDATA[ # 5
  def initialize():
     ...
     # initialization work
     ...

  def cleanup():
     ...
     # cleanup work
     ...

  def createEvent(data,context): # 6
     e = {}
     ...
     return e
  end

  def hb(context): # 7
     print("got a heartbeat")
 ]]></code>
 <forwarding suppress="true|false"> 8
   <destinations>
     <destination window="window" contquery="contquery" blocksize="blocksize"/>
     ...
   </destinations>
 </forwarding></window-python>
```

1 トップレベルのエレメントである<window-python>は、これが Python ウィンドウであることを示しています。

events 属性は、code エレメント内で指定するイベント作成 Python 関数の名前を指定します。この関数は、Python ウィンドウが入力イベントを受信すると呼び出されます。

init 属性は、Python ウィンドウの初期化時に呼び出される Python 関数の名前を指定します。

destroy 属性は、Python ウィンドウが破棄されたときに呼び出される Python 関数の名前を指定します。この関数を使用して、Python クリーンアップタスクを実行できます。

heartbeat 属性は、code エレメント内で指定するハートビート Python 関数の名前を指定します。この関数は、Python ウィンドウがプロジェクトレベルのハートビートメッセージを受信したときに呼び出されます。この属性は必須ではありません。プロジェクトレベルのハートビートメッセージの詳細については、"project" (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

process-blocks 属性は、イベントをブロックで処理するか個別に処理するかを示します。イベントブロックを処理するとき(process-blocks=true)、イベント作成関数は、ブロック内のイベントを表す Python オブジェクトの配列を受け取ります。それ以外の場合、関数は単一のイベント(process-blocks=false を受け取ります(これがデフォルトです)。

encode-binary 属性は、イベント配信用にバイナリデータを Base64 エンコードするかどうかを示します。デフォルト値は **false** です。

on-script-exception 属性は、ウィンドウコードの実行中に発生する例外の処理方法を指定するオプション属性です。log-and-continue に設定すると、エラーがログに記録され、実行が続行されます。stop-and-remove に設定すると、致命的なエラーがログに記録され、プロジェクトが停止し削除されます。

2 Python ウィンドウのスキーマを指定しなければなりません。

3 copy 要素を含めると、指定したフィールドが入力イベントから出力イベントにコピーされます。exclude=true 属性を使って、コピーされないフィールドを指定します。空の copy 要素を含めると、すべてのフィールドが出力イベントにコピーされます。

4 use エレメントを含めると、入力イベントの指定されたフィールドはサーバーから Python コードに渡されます。たとえば、100 個のフィールドを持つ入力イベントがあり、コードでそのうちの 2 つのフィールドしか使用しない場合、それら 2 つのフィールドを use フィールドにリストします。copy フィールドと同様に、exclude を**真**に設定すると、リストされたフィールドを除くすべてのフィールドが Python コードに渡されます。

5 Python コード内のすべての文字を適切に解析できるように、コードを常に XML CDATA 表記で囲みます。

6 ウィンドウが入力イベントを受信したときに呼び出すイベント作成関数を指定します。

7 ウィンドウがプロジェクトレベルのハートビートメッセージを受信したときに呼び出すハートビート関数を指定します。

8 forwarding 要素を含める場合、イベントが生成され、指定された宛先に転送されます。suppress=true 属性を使用して、イベント転送ウィンドウでイベントの生成を防止します。詳細については、"イベント転送"を参照してください。

コード要素には、ウィンドウが所有するコードを含めることも、Python スニペット内に存在する共有コードを指すこともできます。コードがウィンドウによって所有されている場合は、インラインまたは外部ファイルのいずれかに置くことができます。外部ファイルにある場合は、ファイル属性を使用してそのファイルを参照できます。

```
<code file="mycode.py" />
```

```
<code snippet="mysharedcode"/>
```

注: Python スニペットまたは外部ファイルを参照する場合、code 要素にはコードを含めることはできません。

Python ウィンドウがスニペット内で共有コードを使用する場合、そのスニペットを指す他の Python エンティティとすべてのデータ 、関数、および変数が共有されます。共有コードは、現在それを使用しているエンティティによって 常にロックされます。

イベント作成関数は、ウィンドウが入力イベントを取得したときに ESP サーバーが呼び出すエントリポイントです。たとえば、イベント作成関数が呼び出された場合、createEvent の場合、トップレベルの Python ウィンドウ要素は次のようにその関数を指します:

```
<window-python name="python" events="createEvent">
```

イベント作成関数は 2 つの形式のいずれかを取ることができます。最初の形成は次のとおりです。

```
<code><![CDATA[
def createEvent(data,context):
    e = {}
    ...
    return e
end
]]></code>
```

この形式は、2 つのパラメーターを取ります。

- data - Python ウィンドウがイベントブロックを処理するとき、data パラメーターに Python オブジェクトの配列を指定します。各オブジェクトはブロック内のイベントを表します。それ以外の場合は、data パラメーターは、単一のイベントを表す単一の Python オブジェクトを指定します。

  イベントには、use 要素内で指定されたフィールド、use 要素が指定されていない場合はすべてのフィールドが 含まれます。

- context- 関数が使用する変数、データおよびリソースが含まれます。Python ウィンドウでは、次のコンテキストを指定できます。
  - window - 含まれている Python ウィンドウのウィンドウ ID。
  - input - 入力ウィンドウの名前。
  - counter - Python ウィンドウのイベントカウント。
  - opcode - イベントの演算子コード
  - flags - イベントのフラグ

イベント作成関数の 2 番目の形式は、expand 属性が **true** に設定されている場合に、use 要素で指定された任意の数のパラメーターを受け取ります。

```
<use expand="true">id,value</use>
<code><![CDATA[
 def createEvent(id,value):
     newevent = {}
     newevent["id"] = id
     newevent["value"] = value * 10
```

```
    return newevent
 end
]]></code>
```

どちらの形式のイベント作成関数でも、0 個以上のイベントを生成できます。イベントが生成されない場合、関数は **None** を返す必要があります。単一のイベントを生成するには、関数は単一の Python オブジェクトを返す必要があります。複数のイベントを生成するには、関数によって Python オブジェクトの配列が返される必要があります。

注: use 要素の expand 属性は、process-blocks が false の場合のみサポートされます。複数のイベントを処理する場合、イベント作成関数は最初の形式を使用する必要があります。

```
def createEvent(data,context):
   ...
end
```

イベントの生成

シングルイベントの生成

入力イベントごとに 1 つの出力イベントを生成したい場合を考えてみましょう。次のソースウィンドウを考えてみましょう。

```
<window-source name="source" insert-only="true">
 <schema>
  <fields>
   <field name="id" type="string" key="true"/>
   <field name="string" type="string"/>
   <field name="number" type="int32"/>
   <field name="array" type="array(i32)"/>
  </fields>
 </schema>
</window-source>
```

ソースウィンドウと次の Python ウィンドウの間にエッジがある状態で、次の Python ウィンドウは次のタスクを実行します。

- 入力イベントの id を出力イベントにコピーします。
- string フィールドを二重引用符で囲みます。
- number の値を 3 倍します。
- array の各値を 2 倍します。

```
<window-python name="python" events="create">
 <schema>
  <fields>
   <field name="id" type="string" key="true"/>
```

```
    <field name="new_string" type="string"/>
    <field name="new_number" type="int32"/>
    <field name="new_array" type="array(i32)"/>
   </fields>
  </schema>
  <copy>id</copy> <!-- 1 -->
  <code><![CDATA[
def  create(data,context):                    # 2
   e = {}                                  # 3
   e["new_string"] = "\"" + data["id"] + "\""    # 4
   e["new_number"] = data["number"] * 3        # 5
   e["new_array"] = []                        # 6
   for value in data.array:
      e["new_array"].append(value * 2)
   return e                                  # 7
  ]]></code>
</window-python>
```

1 copy 要素は、id 入力イベントから出力イベントにコピーされることを指定しました。

2 data および context という 2 つのパラメーターを取る create という名前の関数を定義します。

3 e という名前の空の辞書を初期化します。

4 new_string キーが辞書に追加されます。生成されるフィールドは、入力フィールド文字列を二重引用符で囲みます。なお、Python の連結演算子は'+'です。

5 new_number キーが辞書に追加されます。フィールドは、入力フィールド番号に 3 を掛けて生成されます。

6 new_array キーが辞書に追加されます。フィールドは配列(Python オブジェクト)を作成します。

7 作成した Python オブジェクトを返します。

作成関数が **None** を返した場合、イベントは作成されません。

たとえば、次のコードでは、入力された数字フィールドが 10 未満の場合、イベントを作成しません。

```
def create(data,context):
    if (data["number"] < 10):
       return None
   ...
end
```

この入力イベントにより、後続の出力イベントが作成されます。

```
i,n,1,string data,33,[11;24;55]
```

出力は次のようになります:

```
<event opcode="insert" window="p/cq/python">
 <value name="id">1</value>
 <value name="new_array">[22;48;110]</value>
 <value name="new_number">99</value>
 <value name="new_string">"string data"</value>
</event>
```

複数のイベントの生成

複数のイベントを生成するには、インデックス付きの Python オブジェクトを作成して返します。インデックス付きオブジェクトには、出力イベントを表す 0 個以上の連想 Python オブジェクトが含まれます。

たとえば、次のコードを考えてみましょう:

```
def create(data,context):
    events = []

    for i in range(1,5):
        event = []
        -- Add event data here
        events.append(event)
    return events
```

次の Python ウィンドウでは、このコードを使用して、配列の各エントリのイベントを生成しています。

```
<window-python name="python" events="create">
 <schema>
  <fields>
   <field name="id" type="string" key="true"/>
   <field name="new_string" type="string"/>
   <field name="new_number" type="int32"/>
   <field name="array_entry" type="int32"/> <!-- 1 -->
  </fields>
 </schema>
 <code><![CDATA[
     eventId = 1

     def create(data,context):
        global  eventId

        events = []

        for value in data["array"]: # 2
           e = {}

           e["id"] = str(eventId)
           eventId += 1 # 3

           e["new_string"] = "\"" + data["string"] + "\""
           e["new_number"] = data["number"] * 3
           e["array_entry"] = value * 2

           events.append(e)

        return events
 ]]></code>
</window-python>
```

1 この行を実行すると、配列ではなく単一のフィールドエントリを持つ出力イベントになります。

2 これらの行では、インデックス付きの配列を作成した後、入力された配列フィールドを繰り返し処理し、各エントリに対して個別のイベントを生成します。各イベントに一意の ID を生成するために、eventId という名前のローカル変数を作成し、それを使って出力イベント ID を設定し、使用後は毎回増分します。

3 eventID 変数は、Python コードの呼び出しの間、その値を維持するので、出力イベントのイベント ID が一意に増分していきます。このように、変数の値を使用して Python ウィンドウの状態を維持することができます。

別のイベント演算子コードの割り当て

デフォルトでは、Python ウィンドウは、着信イベントの演算コードを、発信イベントまたはイベントセットの演算コードに割り当てます。また、_opcode フィールドを使用して、別の演算コードを発信イベントまたはイベントセットに設定することもできます。

```
def create(data,context):
    e = {}
    e["new_string"] = "\"" + data["id"] + "\""
    e["new_number"] = data["number"] * 3
    e["new_array"] = []
    for value in data["array"]:
        e["new_array"].append(value * 2)
  e["_opcode"] = "upsert"
    return e
```

イベント転送

イベントをより効率的にパブリッシュするには、モデルを介したイベントのフローを制御するイベントの転送を使用できます。たとえば、複数の大きなイベントを含むイベントブロックがある場合、blocksize の値を **1** に設定してブロックを分割し、イベントを個別に処理できます。

たとえば、次のコードを考えてみましょう。

```
<forwarding suppress="false"> 1
 <destinations>
    <destination window="sourcewin" contquery="cq_1" blocksize="5" /> 2
 </destinations>
</forwarding>
```

1 この行により、Python ウィンドウでのイベント転送が可能になり、イベント転送ウィンドウでイベントが確実に生成されるようになります。

2 この行は、イベントを転送するソースウィンドウ、ソースウィンドウを含む連続クエリの名前、および各イベントブロックに含めるイベントの数を指定します。

Python ウィンドウでイベント転送を使用する完全プロジェクトの例として、転送の例をインストールします。詳細については、“Install Example Projects” (*SAS Event Stream Processing: Using SAS Event Stream Processing Studio*)を参照してください。

4

ユーティリティ

通知ウィンドウの使用

通知ウィンドウの概要

通知ウィンドウを使用して、次の配信チャネルを通じて通知を送信します。

- SMTP(Simple Mail Transfer Protocol)を使用した電子メール
- SMS (Short Message Service)を使用したテキスト
- MMS (Multimedia Messaging Service)を使用したマルチメディアメッセージ

通知ウィンドウは、関数ウィンドウのように、受信イベントを変換して通知を処理するための function-context を定義することを可能にします。

注: 通知ウィンドウは、SMTP プロトコルを使用して、すべての配信チャネルにコンテンツを配信します。

さまざまな種類の通知のそれぞれに、独自の構成要件があります。たとえば、メールメッセージでは、設定に'送信先'のメールアドレスを含むイベントフィールドを指定する必要があります。SMS と MMS には、電話番号と電話会社のゲートウェイ情報が必須です。

必要に応じて通知を書式設定し、メッセージ内にイベント値を含めることができます。イベント値を含めるには、フィールドの名前の前に**$**文字をメッセージの書式に含めます。

```
<b>$broker</b> sold $quant1 shares of <b>$symbol</b>
$tstap1 for self for $$price1, then sold $quant2 shares for customer at $tstamp2.
```

通知ウィンドウでは、通知を送信するアクティブな配布チャネルをいくつでも作成できます。前述したように、通知を送信するかどうかを決定する関数を指定できます。潜在的に膨大な量のストリーミングデータが通知の集中を引き起こす可能性がある場合、各チャネルの throttle-interval を指定できます。インターバルを**'1 時間'**に設定すると、そのチャネルからの通知を 1 時間に 1 人の受信者に最大で 1 回送信します。

通知ウィンドウはイベントを生成しません。それにもかかわらず、schema 要素を使用して、生成する function-context の値を指定できます。これらの値を使用して通知メッセージをフォーマットすることができます。

通知ウィンドウに関連付けられている XML エレメントについては、"window-notification" (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)および"通知ウィンドウに関連する XML 言語要素" (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

通知ウィンドウ配布チャネル

SMTP サーバーの指定

配布チャネルを使用するには、smtp エレメントを指定して SMTP サーバーに関する情報を提供する必要があります。

```
<smtp host='host' user='user' password='password' port='port' (opt, default='25') />
```

多くの SMTP サーバーはデフォルトポートで実行され、認証を必要としないため、エレメントの host 属性のみが必要です。

```
<smtp host='mailhost.fyi.sas.com' />
```

ただし、smtp エレメントのすべての属性に値を指定することをお勧めします。

```
<smtp host='smtp-server.host.com'
      user='esptest@hostname.com'
      password='esptest1' port='587' />
```

SMTP サーバーに関する情報を指定したら、次の配信チャネルを通じて通知を送信できます。

- email は、指定されたメールアドレスにテキスト、HTML、およびイメージを含むマルチパートメールメッセージを送信します
- sms は、テキストを含む SMS テキストメッセージを、形式 ***phoneNumber@gateway*** のアドレスに送信します。
- mms はテキストとイメージを含む MMS メッセージを、***phoneNumber@gateway*** という形式のアドレスに送信します。

接続の暗号化

通知ウィンドウは、次の 2 つのプロトコルのいずれかを使用して SMTP 接続を暗号化します。SSL (Secure Sockets Layer)または TLS (Transport Layer Security)。SSL は廃止されましたが、まだ広く使用されています。

電子メールが送信されるたびに、クライアントはサーバーとのハンドシェイクを開始します。クライアントは、互換性のある SSL または TLS のバージョンを共有しま

す。サーバーはデジタル証明書で応答し、その ID を確認します。証明書が確認されると、両者は一意のキーを生成して交換します。このキーは、両者間のメッセージを復号化するために使用されます。

まず、クライアントとサーバーの間で接続を確立する必要があります。デフォルトでは、SMTP 接続は保護されていないため、攻撃に対して脆弱です。したがって、両側は安全な接続を確立しようとします。次の 2 つのアプローチがあります。

- 明示的なセキュリティでは、クライアントがハンドシェイクを開始し、TLS を使用して接続を安全な接続にアップグレードしようとします。サーバーの TLS 証明書に互換性があり、エラーが発生しない場合、セキュリティで保護された接続が確立されます。プロセス中に何かが失敗した場合は、プレーンテキスト接続が確立されます。
- 暗黙的なセキュリティを使用すると、クライアントは、使用する暗号化方式をサーバーに尋ねることなく、ハンドシェイクを開始します。接続が成功すると、SSL 接続がセットアップされます。サーバーの SSL 証明書に互換性がない場合、または接続がタイムアウトした場合、送信は中止されます。

ウィンドウが接続する SMTP サーバーが暗黙的なセキュリティを使用している場合は、ウィンドウの<ssl>要素の<smtp>要素を指定します。SMTP サーバーが明示的なセキュリティを使用している場合は、<smtp>要素の<tls>要素を指定します。

メール配布チャネルの使用

email 配布チャネルを使用する XML コードは次のとおりです。

```
<email throttle-interval='' test='true | false'>
  <deliver>[code]</deliver>
  <email-info>
    <sender>[sender]</sender>
    <recipients>[recipients]</recipients>
    <subject>[subject]</subject>
    <from>[from]</from>
    <to>[to]</to>
  </email-info>
  <email-contents>
    <text-content name=''>...</text-content>
    <html-content name=''>...</html-content>
    <image-content name=''>...</image-content>
    ...
  </email-contents>
</email>
```

email エレメントには次の属性が含まれます。

- throttle-interval は、わずか 1 つの通知が受信者に送信される期間を指定します。
- test は、テストモードで実行するかどうかを指定するブール属性です。テストモードで実行すると、通知は送信されずにコンソールに書き込まれます。この機能は、通知メッセージを作成するときに役立ちます。

deliver エレメントはオプションです。通知が送信されるべきかどうかを決定する関数を含みます。

email-info エレメントには、電子メール通知を送信するために使用されるデータを表す関数またはハードコードされた値が含まれます。それは次のエレメントを含んでいます。

- sender のメールアドレス。
- メールメッセージが送信される recipients 複数の受信者は、','文字または';'文字で区切ることができます。
- メールの subject。
- メールメッセージの from のテキスト。
- メールの to のテキスト。
- email-contents エレメントには、次のエレメントが含まれます。
  - その text-content エレメントは、メッセージのプレーンテキストの内容を囲みます。
  - html-content エレメントはメッセージの HTML コンテンツを囲みます。
  - その image-content エレメントは画像データへの URI を囲みます。この URI は、**file:///*directory/image_filename*** または **http://*website/image_filename*** のように外部エンティティを指すことができます。次のように、通知ウィンドウまたは直前のウィンドウのスキーマで指定されたイベントから直接埋め込まれたイメージデータを指すこともできます。**field://*fieldname***

これらのエレメントは、任意の方法で散在させることができます。各エレメントの内容は、表示される順序でメッセージに含まれます。画像データはすべて取得され、Base64 でエンコードされてからメッセージに挿入されます。

SMS 配布チャネルの使用

sms 配布チャネルを使用する XML コードは次のとおりです。

```
<sms throttle-interval='' test='true | false'>
  <deliver>[code]</deliver>
  <sms-info>
    <sender>[sender]</sender>
    <subject>[subject]</subject>
    <from>[from]</from>
    <gateway>[gateway]</gateway>
    <phone>[phone]</phone>
  </sms-info>
  <sms-contents>
    <text-content name=''>...</text-content>
  </sms-contents>
</sms>
```

sms エレメントには次の属性が含まれます。

- throttle-interval は、わずか 1 つの通知が受信者に送信される期間を指定します。
- test は、テストモードで実行するかどうかを指定するブール属性です。テストモードで実行すると、通知は送信されずにコンソールに書き込まれます。この機能は、通知メッセージを作成するときに役立ちます。

deliver エレメントはオプションです。通知が送信されるべきかどうかを決定する関数を含みます。

sms-info エレメントには、SMS テキスト通知を送信するために使用されるデータを表す関数またはハードコードされた値が含まれます。それは次のエレメントを含んでいます。

- sender の SMS アドレス
- メッセージの subject
- メッセージの from のテキスト
- その gateway エレメントは、受信者のプロバイダーの SMS ゲートウェイを指定します。たとえば、AT&T は txt.att.net です。Sprint は messaging.sprintpcs.com です。
- その sms-contents エレメントには、送信されるメッセージの本文が含まれます。それは次のエレメントを含んでいます。
  - その text-content エレメントは、メッセージのプレーンテキストの内容を囲みます。

MMS 配布チャネルの使用

MMS 配布チャネルを使用する XML コードは次のとおりです。

```
<mms throttle-interval='' test='true | false'>
  <deliver>[code]</deliver>
  <mms-info>
    <sender>[sender]</sender>
    <subject>[subject]</subject>
    <gateway>[gateway]</gateway>
    <phone>[phone]</phone>
  </mms-info>
  <mms-contents>
    <text-content name=''>...</text-content>
    <image-content name=''>...</image-content>
    ...
  </mms-contents>
</mms>
```

mms エレメントには次の属性が含まれます。

- throttle-interval は、わずか 1 つの通知が受信者に送信される期間を指定します。
- test は、テストモードで実行するかどうかを指定するブール属性です。テストモードで実行すると、通知は送信されずにコンソールに書き込まれます。この機能は、通知メッセージを作成するときに役立ちます。

deliver エレメントはオプションです。通知が送信されるべきかどうかを決定する関数を含みます。

mms-info エレメントには、MMS 通知を送信するために使用されるデータを表す関数またはハードコードされた値が含まれます。それは次のエレメントを含んでいます。

- sender の MMS アドレス。

- メッセージの subject
- その gateway エレメントは、受信者のプロバイダーの SMS ゲートウェイを指定します。たとえば、AT＆T は txt.att.net です。Sprint は messaging.sprintpcs.com です。
- 受信者の phone 番号
- その mms-contents エレメントには、送信されるメッセージの本文が含まれます。それは次のエレメントを含んでいます。
  - その text-content エレメントは、メッセージのプレーンテキストの内容を囲みます。
  - その image-content エレメントは画像データへの URI を囲みます。この URI は、**file:///*directory/image_filename*** または **http://*website/image_filename*** のように外部エンティティを指すことができます。次のように、通知ウィンドウまたは直前のウィンドウのスキーマで指定されたイベントから直接埋め込まれたイメージデータを指すこともできます。**field://*fieldname***

これらのエレメントは、任意の方法で散在させることができます。各エレメントの内容は、表示される順序でメッセージに含まれます。画像データはすべて取得され、Base64 でエンコードされてからメッセージに挿入されます。

function-context エレメントの使用

function-context エレメントでは、イベントデータを操作する関数を定義することができます。正規式、XML および XPath、または JSON を使用して、複雑な入力情報からより使用可能なデータにデータを変換できます。

function-context エレメントを使用する XML コードを次に示します。

```
<function-context>
  <expressions>
    <expression name="">[Regular Expression]</expression>
    ...
  </expressions>
  <properties>
    <property-map name='' outer='' inner="">[code]</property-map>
    <property-xml name="">[code]</property-xml>
    <property-json name="">[code]</property-json>
    <property-string name="">[code]</property-string>
    <property-list name='' delimiter="">[code]</property-list>
    <property-set name='' delimiter="">[code]</property-set>
    ...
  </properties>
  <functions>
    <function name="">[code]</function>
    ...
  </functions>
</function-context>
```

function-context エレメントには、次の 2 種類の関数を使用できます。

- 一般的な関数(たとえば、 abs 、 ifNext 、など)

- イベントストリーム処理に固有の関数(たとえば、 eventNumber)

**$**表記を使用して、入力イベントまたは出力イベントのいずれかでイベントフィールドを参照できます(たとえば、$[*name_of_field*])。

入力イベントに名前フィールドがあり、名前の値に基づいて出力イベントに占有フィールドを生成するとします。この場合、次の関数を使用できます。

```
<function name='occupation'>
ifNext
(
  equals($name,'larry'),'plumber',
  equals($name,'moe'),'electrician',
  equals($name,'curly'),'carpenter'
)
</function>
```

職業に依存する出力イベントに hourlyWage を追加するとします。

```
<function name='hourlyWage'>
ifNext
(
  equals($occupation,'plumber'),85.0,
  equals($occupation','electrician'),110.0,
  equals($occupation,'carpenter'),60.0
)
</function>
```

注: シーケンスは、function-context エレメントで関数を定義するときに重要です。関数が出力イベントフィールドを参照する場合、参照フィールドの前にそのフィールドを計算する必要があります。

コード内で POSIX 正規式を使用します。正規式にはいくつかの関数があります。正規式は使用する前にコンパイルする必要があるため、expressions エレメントを使用して、関数コンテキストの作成時に式が 1 回コンパイルされるように指定します。次に、次の表記法を使用して関数内から式を参照できます。

```
#[name_of_expression]
<function name='myData'>rgx(#myExpression,$inputField,1)
</function>
```

たとえば、URI を含むデータフィールドを受け取り、そこからプロトコルを抽出するとします。次の関数を使用すると、関数が実行されるたびに正規式がコンパイルされます。

```
<function name='protocol'>rgx('(.*):',$uri,1)
</function>
```

次のコードを使用すると、式は 1 回コンパイルされ、関数が実行されるたびに使用されます。

```
<expressions>
  <expression name='getProtocol'>(.*):
  </expression>
</expressions>

<function name='protocol'>rgx(#getProtocol,$uri,1)
</function>
```

#表記#[*name_of_property*]を使用して関数内のプロパティを参照します。

properties エレメントは、次のエレメントのコンテナーです。

エレメント	説明
property-map	値ルックアップに使用される名前と値のペアのマップを生成する関数を実行します。
property-list	インデックス付きアクセスに使用する文字列のリストを生成する関数を実行します。
property-set	値ルックアップに使用する文字列のセットを生成する関数を実行します。
property-xml	XPath クエリに使用できる XML オブジェクトを生成する関数を実行します。
property-json	この関数を実行して、JSON 検索に使用できる JSON オブジェクトを生成します。
property-string	関数内で一般的に使用される文字列を生成する関数を実行します。

各プロパティは関数を使用して生成されます。これらの関数は、XML で前に定義されたプロパティを参照できます。

従業員情報がモデルにストリーミングされているとします。

```
<event>
  <value name='map'>name:[employee name];position:[employee position]</value>
  <value name='developerInfo'><![CDATA[<info>this is developer info</info>]]></value>
  <value name='managerInfo'><![CDATA[<info>this is manager info</info>]]></value>
</event>
```

適切なデータを含む property-xml を作成するために、property-map エレメントを使用して従業員データを格納し、イベントの position フィールドを調べることができます。従業員が開発者の場合、XML は developerInfo から作成されます。それ以外の場合は、managerInfo を使用します。

次のように、function-context エレメントを指定します。

```
<function-context>
  <properties>
    <property-map name='myMap' outer=';' inner=':'>$map</property-map>
    <property-xml name='myXml'>
     if(equals(mapValue(#myMap,'position'),'developer'),
     $developerInfo,$managerInfo)</property-xml>
  </properties>
  <functions>
    <function name='employee'>mapValue(#myMap,'name')</function>
    <function name='info'>xpath(#myXml,'text()')</function>
  </functions>
</function-context>
```

次のイベントをストリーミングする場合

```
<event>
  <value name='map'>name:curly;position:developer</value>
  <value name='developerInfo'><![CDATA[<info>this is developer info</info>]]></value>
  <value name='managerInfo'><![CDATA[<info>this is manager info</info>]]></value>
</event>

<event>
  <value name='map'>name:moe;position:manager</value>
  <value name='developerInfo'><![CDATA[<info>this is developer info</info>]]></value>
  <value name='managerInfo'><![CDATA[<info>this is manager info</info>]]></value>
</event>
```

function-context は次のようになります。

```
<event opcode='insert' window='project/query/transform'>
  <value name='employee'>curly</value>
  <value name='id'>fd26bf36-3d65-4d17-8dc6-317409bbf5b6</value>
  <value name='info'>this is developer info</value>
</event>

<event opcode='insert' window='project/query/transform'>
  <value name='employee'>moe</value>
  <value name='id'>84c56bb7-9f3c-4cb8-93a5-8dc2f75d353b</value>
  <value name='info'>this is manager info</value>
</event>
```

各プロパティを定義する方法の詳細については、*"式と関数の XML 言語要素" (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*.を参照してください。

パターンウィンドウの使用

パターンウィンドウの概要

パターンウィンドウを使用して、関心のあるイベント(EOI)がプロジェクトを通過する際に検出し、パターンロジックを適用した結果として出力イベントを生成します。

EOI を指定するには、次の 3 つの方法があります。

- <code>要素内に Lua 関数を記述します。
- <code>要素内に Python 関数を記述します。
- <event>要素内で式エンジン言語(EEL)を使用します。

いずれの場合も、演算子を使用してパターン ロジックを適用することにより、EOI を<logic>要素内の式にアセンブルします。式は一時的条件を含むこともできます。

Lua または Python を使用して EOI を指定する場合、別の Lua または Python 関数を使用して出力イベントを生成します。EEL を使用して EOI を指定する場合、<output>要素を使用して出力イベントを生成します。

注: パターン ウィンドウでは Lua または Python を使用することをお勧めします。パターン ウィンドウでの EEL の使用は、従来の機能です。

パターンウィンドウに関連付けられている XML 要素については、*"window-pattern" (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*および*"パターンウィンドウに関連する XML 言語要素" (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*を参照してください。

パターン ロジックの適用

<logic>要素内で、次の論理演算子をイベントにオペランドとして使用して、パターン ロジックを作成します。

表 4.1 パターン論理に対する有効な論理演算子

論理演算子	関数
and	すべてのオペランドが True です。任意の数のオペランドをとります。
or	そのオペランドのいずれかが True です。任意の数のオペランドをとります。
not	オペランドは True ではありません。1 つのオペランドをとります。
is	オペランドに続くイベントが指定どおりであることを確認します。
fby	各オペランドの後ろにはオペランドが続きます。任意の数のオペランドをとります。
notoccur	オペランドは決して発生しません。1 つのオペランドをとります。

FBY 演算子(fby)は性質上逐次的です。単一のイベントは両側で一致することはできません。左側はイベントで最初にマッチしなければならず、その後のイベントは右側でマッチする可能性があります。

AND 演算子と OR 演算子は連続していません。すべての受信イベントは、論理演算子のどちらの側の EOI とも一致することができ、最初に一致する EOI は変数バインディングをトリガーします。この場合、パターンを書くときに意図することはまれですので、特に注意してください。

一時的な条件をパターン ロジックに適用できます。次の式を考えてみましょう:

fby(e1, e2, e3)

fby 演算子に一時的な条件を適用するには、中括弧内に条件を追加します。たとえば、次の式を指定したとします:

fby{10 seconds}(event1,event2,event3)

ここでは、タイマーを開始するには event1 が発生する必要があります。次に、event2 が event1 から 10 秒以内に発生する必要があります。その後、event3 が続きます。

ここで、次の式を指定したとします:

fby{10 seconds}(fby{10 seconds}(event1,event2),event3)

ここでは、 event1 が発生し、event2 が 10 秒以内に続くとタイマーが開始します。次に、式全体が true と評価されるには、event3 が 10 秒以内に発生する必要があります。

一時的な条件は、リアルタイム実行、または DATE-TIME または TIMESTAMP フィールドによって定義することができます。このフィールドは、パターンウィンドウに供給されるウィンドウに関連付けられているスキーマに表示されます。フィールドベースの DATE-TIME または TIMESTAMP を使用する場合は、受信イベントの順序が正しいことを確認する必要があります。

パターンの式を時間で縛るのは良い習慣です。たとえば、次の式を考えてみましょう:

and{2 seconds}(event1,event2,event3)

オープンパターンは、3 つのイベントがすべて発生した後に終了するか、最初のイベントが発生してから 2 秒後に終了するかのいずれかです。反対に、次の式を考えてみましょう:

and(event1,event2,event3)

この場合、event1 と event2 は頻繁に発生するが event3 は稀に発生する場合、多くのオープンパターンが作成されます。この動作では、新しい event1 や event2 を受信するたびにメモリ使用量が増加する可能性があります。最終的に、その増加により、使用可能なすべてのメモリが消費されてしまう可能性があります。コンソールに表示される INFO メッセージは、オープンパターンの数を追跡します。

```
[Pattern0025] dfESPpattern::trackHWmark(0x00000000033f7200): [limit: 64] has 33
active instances in pattern window <patternWindow_01>
```

開いているインスタンスの数が増えると、モデルのパフォーマンスが低下することがあります。

```
[Pattern0025] dfESPpattern::trackHWmark(0x00000000033f7200): [limit: 4096] has
2049 active instances in pattern window <patternWindow_01>
```

パターンウィンドウでの Lua の使用

パターンウィンドウ内での Lua 関数の使用

Lua 関数を記述して、EOI を指定し、パターンロジックを適用した結果として出力イベントを生成できます。Lua を使用する場合は、次のアクションを完了します:

- <code>要素を使用して、次の関数を含めます:
  - イベントが EOI であるかどうかを判断するために true または false を返す関心のあるイベント(EOI)関数
  - 適用されたパターン一致基準が満たされたときに出力イベントを生成する出力関数
- イベントを<lua-events>要素で囲みます。<event>要素は、イベントを明示的に指定するのではなく、Lua 関数を参照します。

パターンウィンドウで Lua を使用するときに使用できる追加機能については、8 章, “共通の Lua 機能”を参照してください。

Lua で EOI 関数を記述する

Lua の EOI 関数は、次のパラメーターを取ります:

- event は、ストリーム内の現在のイベントを表す Lua テーブルを指定します。

  次に例を示します:

  ```
  symbol=IBM
  quantity=500
  price=97.34
  id=0
  broker=John
  ```

- context は、次の情報を含む Lua テーブルを指定します:
  - window - パターンウィンドウの名前

    次に例を示します:

    ```
    <window-pattern name='pattern_win'>
    ```

  - name - 呼び出しイベントの名前

    次に例を示します:

    ```
    <lua-events>
       <event name='e1' func='f1'/>
     </lua-events>
    ```

  - input - 受信イベントのソースウィンドウの名前
  - 他の関数から返されたコンテキストデータ

- events - 該当する場合、パターン内の先行するすべてのイベント

たとえば、次のイベントについて考えてみます:

```
events=
   paul=
      price=57.34
      symbol=AMZN
      broker=Paul
      id=1
      quantity=500
   john=
      price=97.34
      symbol=IBM
      broker=John
      id=0 quantity=500
   name=george
   window=2
   data=
      mydata=foo
```

EOI 関数は、少なくとも true または false を返す必要があります。

```
function f1(event,context)
   local code = false
   if (event.broker == "John")
   then
      code = true
   end
   return code
end
```

これらの関数を使用して、情報をコンテキストに入れ、後続の EOI 関数で使用することもできます。たとえば、次のコンテキストで取引価格を保存するとします:

```
function f1(event,context)
   local code = false
   if (event.broker == "John")
   then
      return true, {tradeprice=event.price}
   end
   return false
end
```

EOI 関数は、2 番目の戻り値として Lua テーブルを返すことができます。この場合、テーブルはコンテキストに格納され、後続の EOI 関数で使用できます。前の例では、次のように取引価格にアクセスします:

```
function lowerprice(event,context)
    return(event.price < context.data.tradeprice)
end
```

Lua での出力関数の記述

出力関数は、関連するすべての EOI 関数が実行され、一連のイベントがパターンロジックと一致したときに呼び出されます。この時点で、ESP サーバーは出力関数を呼び出し、それをパターンコンテキストに渡します。

```
data=
   mydata=foo
   window=1
   events= paul=
            id=1
            price=57.34
            quantity=500
            symbol=AMZN
            broker=Paul
         john=
            id=0
            price=97.34
            quantity=500
            symbol=IBM
            broker=John
```

コンテキストには、一致するすべてのイベントだけでなく、EOI 関数から以前に返されたコンテキストデータが含まれます。イベントはイベント名でキー付けされます。次に示すように、イベントフィールドにアクセスできます:

```
function output(context)
    local event = {}
    event.symbol_1 = context.events.john.symbol
    event.symbol_2 = context.events.paul.symbol
    return event
end
```

返されたイベントには一意の ID が割り当てられ、イベントストリームに挿入されます。

イベントの定義

Lua ベースのパターンウィンドウで、<event>要素は、式コードを含む代わりに、Lua で記述された EOI 関数を参照します。次のコードスニペットでは、イベント john は f1 という名前の EOI 関数を参照し、イベント paul は f2 という名前の EOI 関数を参照します。

```
<lua-events>
   <event name="john" func="f1" />
   <event name="paul" func="f2" />
</lua-events>
```

デフォルトでは、すべてのイベントフィールドが ESP サーバーから EOI 関数に渡されます。効率を最大化するために、特定のイベントフィールドのみを関数に送信するように指定できます。たとえば、次の定義は、ESP サーバーに次の 2 つのフィールド broker と price のみを EOI 関数 f1 に送信するように指示します。

```
<lua-events>
    <event name="e1" func="f1">
     <use>broker,price</use>
    </event>
</lua-events>
```

このアクションは、数十のイベントフィールドがあるが、EOI 関数で使用するイベントフィールドが少ない場合に効果的です。

該当する場合は、以前に一致したイベントのセット全体を<event>要素の sendevents 属性を持つ関数に送信できます。

```
<lua-events>
    <event name="paul" func="f2" sendevents="true" />
</lua-events>
```

sendevents を指定する場合、ESP サーバーは、以前に一致したすべてのイベントをイベントフィールドのコンテキストに入れます:

```
name=paul
events=
  john=
    broker=John
    price=97.34
    quantity=500
    symbol=IBM
    id=0
```

これらのイベントは、関連付けられているイベント名(一致したイベント名)によって参照されます。これにより、スタック全体のイベント間でデータを簡単に比較できます。

注: 少数のフィールドセットを使用する場合は、毎回すべてのイベントをスタックに置くよりも、コンテキストデータを使用する方が効率的です。

繰り返しイベント

多くのパターンは、繰り返される EOI を探します。Lua ベースのパターンウィンドウは、イベント定義とパターンロジックを使用して繰り返しを検出する単純なメカニズムを提供します。:*number* を繰り返す要素の後に追加するだけです。たとえば、5 回繰り返されるイベント e1 を探したいとします。

fby(e1:5)

演算子またはイベントに番号を追加できます。

fby(is:5(e1))

それぞれの価格が前の価格よりも低い 3 つの連続する株式取引をキャプチャしたいとします(価格が高かった取引が間にない)。次のコードを使用できます。

```
<pattern index="symbol">
  <lua-events>
    <event name="start" func="start" /> <!-- 1 -->
    <event name="decrease" func="f1" /> <!-- 2 -->
  </lua-events>
  <logic>fby(start,is:2(decrease))</logic> <!-- 3 -->
  <code output="output">
    <![CDATA[
    function start(event,context)
       return true, {price=event.price}
    end
```

```
        function f1(event,context)
          if (event.price < context.data.price)
          then
            return true, {price=event.price}
          end

          return false
        end ...
    </pattern>
```

1 あらゆるイベントに対して true を返し、コンテキストで取引価格を設定する開始イベントを定義します。

2 現在の取引価格が以前の取引価格よりも低い場合に true を返し、(コンテキストで取引価格を設定する)減少イベントを定義します。

3 繰り返しロジック構文を使用して、減少イベントが 2 回発生することをリクエストします。

繰り返しイベントがある場合、繰り返しシーケンス内で発生するすべての EOI は、コンテキストデータ内で配列として表されます。したがって、前の例では、出力関数は次のように decrease にアクセスできます。

```
function output(context)
  local event = {}
  -- Loop through all decrease events
  for index,value in ipairs(context.events.decrease) do
  -- set event values
  end
  return event
end
```

この機能により、コードを修正せずに探している繰り返し回数を変更できます。前の例では、次のように logic で繰り返し修飾子を変更することで、パターンを増やして 4 つの decreasing 価格取引を探すことができます。

```
<logic>fby(start,is:3(decrease))</logic>
```

配列の使用

パターンウィンドウ内で Lua コードを使用して、イベント配列フィールドを書き込むことができます。この機能は、出力のため複数の EOI から同じフィールドを引き出す必要がある場合に役立ちます。

5 つの株式取引を検出するパターンがあるとします:

```
<field name="price_1" type="double"/>
<field name="price_2" type="double"/>
<field name="price_3" type="double"/>
<field name="price_4" type="double"/>
<field name="price_5" type="double"/>
```

列挙された価格フィールドのリストの代わりに、価格配列を指定できます。

```
<field name="prices" type="array(dbl)"/>
```

Lua コードで配列の要素に値を割り当てた後、次のように処理できます。

```
...
local event = {}
event.prices[1] = event1.price
event.prices[2] = event2.price
event.prices[3] = event3.price
event.prices[4] = event4.price
event.prices[5] = event5.price
...
```

パターンウィンドウでの Python の使用

パターンウィンドウ内での Python 関数の使用

Python 関数を記述して、EOI(Event of Interest)関数を指定し、パターンロジックを適用した結果として出力イベントを生成できます。Python を使用する場合は、次のアクションを完了します。

- <code>要素を使用して、次の関数を含めます:
  - イベントが EOI であるかどうかを判断するために true または false を返す EOI 関数
  - 適用されたパターン一致基準が満たされたときに出力イベントを生成する出力関数
  - モデルの開始前に呼び出すオプションの初期化関数。
  - ウィンドウが破棄されるときに呼び出すオプションの破棄関数。
- イベントを<python-events>要素で囲みます。<event>要素は、イベントを明示的に指定するのではなく、Python 関数を参照します。

パターンウィンドウで Python を使用するときに使用できる追加機能については、9章, “一般的な Python 機能”を参照してください。

Python で EOI 関数を記述する

Python の EOI 関数は、次のパラメーターを取ります。

- event は、ストリーム内の現在のイベントを表す Python オブジェクトを指定します。次に例を示します:

  {'id': 0, 'broker': 'John', 'symbol': 'IBM', 'price': 97.34, 'quantity': 500}

- context には、EOI 関数で使用できる変数、データおよびリソースが含まれます。パターンウィンドウでは、コンテキストに次の情報を含めることができます。
  - window - パターンウィンドウの名前

    次に例を示します:

    <window-pattern name='pattern_win' >

- name - 呼び出しイベントの名前

  次に例を示します:

  ```
  <python-events>
     <event name='e1' func='f1'/>
   </python-events>
  ```

- input - 受信イベントのソースウィンドウの名前
- 他の関数から返されたコンテキストデータ
- events - 該当する場合、パターン内の先行するすべてのイベント

たとえば、次を考えてみましょう:

```
{
   'name': 'george',
   'input': 's',
   'window': 'pattern',
   'events': {
      'john': {'id': 0, 'broker': 'John', 'symbol': 'IBM', 'price': 97.34, 'quantity': 500},
      'paul': {'id': 3, 'broker': 'Paul', 'symbol': 'IBM', 'price': 57.34, 'quantity': 500}
   }
}
```

EOI 関数は、少なくとも true または false を返す必要があります。

```
def f1(event,context):
   code = False

   if (event["broker"] == "John"):
      code = True

   return code
```

これらの関数を使用して、情報をコンテキストに入れ、後続の EOI 関数で使用することもできます。たとえば、次のコンテキストで取引価格を保存するとします:

```
def f1(event,context):
   code = False

   if (event["broker"] == "John"):
      return True, {"tradeprice":event["price"]}

   return False
```

EOI 関数は、2 番目の戻り値として Python オブジェクトを返すことができます。この場合、オブジェクトはコンテキストに格納され、後続の EOI 関数で使用できます。前の例では、次のように取引価格にアクセスします:

```
def lowerprice(event,context):
   return(event["price"] < context["data"]["tradeprice"])
```

Python で出力関数を記述する

出力関数は、関連するすべての EOI 関数が実行され、一連のイベントがパターンロジックと一致したときに呼び出されます。この時点で、ESP サーバーは出力関数を呼び出し、それをパターンコンテキストに渡します。

コンテキストには、一致するすべてのイベントだけでなく、EOI 関数から以前に返されたコンテキストデータが含まれます。イベントはイベント名でキー付けされます。次に示すように、イベントフィールドにアクセスできます:

```
def output(context):
  event = {}
  event["symbol_1"] = context["events"]["john"]["symbol"]
  event["symbol_2"] = context["events"]["paul"]["symbol"]
  return event
```

返されたイベントには一意の ID が割り当てられ、イベントストリームに挿入されます。

イベントの定義

Python ベースのパターンウィンドウで、<event>要素は、式コードを含む関数ではなく、Python で記述された EOI 関数を参照します。次のコードスニペットでは、イベント john は f1 という名前の EOI 関数を参照し、イベント paul は f2 という名前の EOI 関数を参照します。

```
<python-events>
  <event name="john" func="f1" />
  <event name="paul" func="f2" />
</python-events>
```

デフォルトでは、すべてのイベントフィールドが ESP サーバーから EOI 関数に渡されます。効率を最大化するために、特定のイベントフィールドのみを関数に送信するように指定できます。たとえば、次の定義は、ESP サーバーに次の 2 つのフィールド broker と price のみを EOI 関数 f1 に送信するように指示します。

```
<python-events>
   <event name="e1" func="f1">
    <use>broker,price</use>
   </event>
</python-events>
```

このアクションは、数十のイベントフィールドがあるが、EOI 関数で使用するイベントフィールドが少ない場合に効果的です。

該当する場合は、以前に一致したイベントのセット全体を<event>要素の sendevents 属性を持つ関数に送信できます。

```
<python-events>
    <event name="paul" func="f2" sendevents="true" />
</python-events>
```

sendevents を指定する場合、ESP サーバーは、以前に一致したすべてのイベントをイベントフィールドのコンテキストに入れます:

```
{
  'name': 'george',
  'input': 's',
  'window': 'pattern',
  'events': {
     'john': {'id': 0, 'broker': 'John', 'symbol': 'IBM', 'price': 97.34, 'quantity': 500},
     'paul': {'id': 3, 'broker': 'Paul', 'symbol': 'IBM', 'price': 57.34, 'quantity': 500}
  }
```

```
}
```

これらのイベントは、関連付けられているイベント名(一致したイベント名)によって参照されます。これにより、スタック全体のイベント間でデータを簡単に比較できます。

注: 少数のフィールドセットを使用する場合は、毎回すべてのイベントをスタックに置くよりも、コンテキストデータを使用する方が効率的です。

繰り返しイベント

多くのパターンは、繰り返される EOI 関数を探します。Python ベースのパターンウィンドウは、イベント定義とパターンロジックを使用して繰り返しを検出する単純なメカニズムを提供します。*:number* を繰り返す要素の後に追加するだけです。たとえば、5 回繰り返されるイベント e1 を探したいとします。

```
fby(e1:5)
```

演算子またはイベントに番号を追加できます。

```
fby(is:5(e1))
```

それぞれの価格が前の価格よりも低い 3 つの連続する株式取引をキャプチャしたいとします(価格が高かった取引が間にない)。次のコードを使用できます。

```
<pattern index="symbol">
  <python-events>
    <event name="start" func="start" /> <!-- 1 -->
    <event name="decrease" func="f1" /> <!-- 2 -->
  </python-events>
  <logic>fby(start,is:2(decrease))</logic> <!-- 3 -->
  <code output="output">
    <![CDATA[
  def start(event,context):
      return True, {"price":event["price"]}

  def f1(event,context):
      if (event["price"] < context["data"]["price"]):
        return True, {"price":event["price"]}
      ...
</pattern>
```

1 あらゆるイベントに対して true を返し、コンテキストで取引価格を設定する開始イベントを定義します。

2 現在の取引価格が以前の取引価格よりも低い場合に true を返し、(コンテキストで取引価格を設定する)減少イベントを定義します。

3 繰り返しロジック構文を使用して、減少イベントが 2 回発生することをリクエストします。

繰り返しイベントがある場合、繰り返しシーケンス内で発生するすべての EOI 関数は、コンテキストデータ内で配列として表されます。したがって、前の例では、出力関数は次のように decrease にアクセスできます。

```
def output(context):
  event = {}

  -- Loop through all decrease events

  for value in context["events"]["decrease"]::
    -- set event values

  return event
```

この機能により、コードを修正せずに探している繰り返し回数を変更できます。前の例では、次のように logic で繰り返し修飾子を変更することで、パターンを増やして 4 つの decreasing 価格取引を探すことができます。

```
<logic>fby(start,is:3(decrease))</logic>
```

配列の使用

パターンウィンドウ内で Python コードを使用して、イベント配列フィールドを書き込むことができます。この機能は、出力のため複数の EOI 関数から同じフィールドを引き出す必要がある場合に役立ちます。

5 つの株式取引を検出するパターンがあるとします。

```
<field name="price_1" type="double"/>
<field name="price_2" type="double"/>
<field name="price_3" type="double"/>
<field name="price_4" type="double"/>
<field name="price_5" type="double"/>
```

列挙された価格フィールドのリストの代わりに、価格配列を指定できます。

```
<field name="prices" type="array(dbl)"/>
```

Python で配列の要素に値を割り当てた後、次のように処理できます。

```
...
event = {}
event["prices"] = []
event["prices"].append(event1["price"])
event["prices"].append(event2["price"])
event["prices"].append(event3["price"])
event["prices"].append(event4["price"])
event["prices"].append(event5["price"])
...
```

式エンジン言語の使用

式エンジン言語(EEL)を使用する場合は、次の情報を提供して <event>要素内で EOI を指定します。

- イベントの発生元のウィンドウへのポインター。
- EOI の文字列名。

- 受信イベントのフィールド上の WHERE 句。WHERE 句には、多数の統一変数(バインディング)を含めることができます。

たとえば、次の EOI のリストでは、次の情報を確認できます。

- ポインターは src_win です。
- EOI の名前は e1、e2、および e3 です。
- WHERE 句には統一変数 symbol、price、および b が含まれます。

```
<event source="src_win" name="e1">symbol=="IBM" and price > 101.00 and b == buy</
event>
<event source="src_win" name="e2">symbol=="T" and price > 30.000 and b == buy</event>
<event source="src_win" name="e3">symbol=="AMZN" and price > 1800.00 and b == buy</
event>
```

式エンジン言語(EEL)の詳細については、*式言語: リファレンスガイド*を参照してください。

ブローカー監視モデルのパターンの XML 例を次に示します。

```
<pattern>
    <events>
      <event name='e1'>((buysellflg==1) and (broker == buyer)
      and (s == symbol) and (b == broker) and (p == price))</event>
      <event name='e2'>((buysellflg==1) and (broker != buyer)
      and (s == symbol) and (b == broker))</event>
      <event name='e3'><![CDATA[((buysellflg==0) and (broker == seller)
      and (s == symbol) and (b == broker) and (p < price))]]></event>
    </events>
    <logic>fby{1 hour}(fby{1 hour}(e1,e2),e3)</logic>
    ...
  </pattern>
  <pattern>
    <events>
      <event name='e1'>((buysellflg==0) and (broker == seller)
      and (s == symbol) and (b == broker))</event>
      <event name='e2'>((buysellflg==0) and (broker != seller)
      and (s == symbol) and (b == broker))</event>
    </events>
    <logic>fby{10 minutes}(e1,e2)</logic>
    ...
  </pattern>
</patterns>
</window-pattern>
```

電子商取引モデルのパターンの XML 例を次に示します。

```
<pattern>
    <events>
      <event name='e1'>eventname=='ProductView'
      and c==customer and p==product</event>
      <event name='e2'>eventname=='AddToCart'
      and c==customer and p==product</event>
      <event name='e3'>eventname=='CompletePurchase'
      and c==customer</event>
      <event name='e4'>eventname=='Sessions'
      and c==customer</event>
      <event name='e5'>eventname=='ProductView'
```

```
        and c==customer and p!=product</event>
        <event name='e6'>eventname=='EndSession'
        and c==customer</event>
      </events>
      <logic>fby(e1,fby(e2,not(e3)),e4,e5,e6)</logic>
      ...
    </pattern>
```

パターンウィンドウ内に複数のパターンを定義することができます。各パターンは、通常、複数のウィンドウまたは単一の入力ウィンドウから複数の EOI を持ちます。

パターンウィンドウに供給するウィンドウが 1 つあり、関連するスキーマが次のようになっているとします。

```
<schema>
  <fields>
    <field name="ID" type="int32" key="true"/>
    <field name="symbol" type="string"/>
    <field name="price" type="double"/>
    <field name="buy" type="int32"/>
    <field name="tradeTime" type="date"/>
  </fields>
</schema>
```

さらに、2 つの EOI があり、それらの関係が一時的であると仮定します。ある期間内の 1 つのイベントとそれに続くイベントに興味があります。このシナリオは、次のコードセグメントに示されています。

```
<pattern>
  <events>
    <!-- Someone buys IBM at price > 150.00 followed within
        5 seconds of buying MSFT at price > 110.00 -->
    <event name="e1">symbol=="IBM" and price > 150.00 and b==buy</event>
    <event name="e2">symbol=="MSFT" and price > 110.00 and b==buy</event>
  </events>
  <logic>fby{5 seconds}(e1, e2)</logic>
...
</pattern>
```

ここには 2 つの EOI、e1 と e2 があります。WHERE 句の先頭は標準です。symbol==*constant* and price>*constant* になります。各 WHERE 句の最後の部分は、イベントの統合が行われる場所です。

b は受信イベントのフィールドではないため、イベントが到着したときにバインドされる空き変数です。イベント e1 の WHERE 句の最初の部分と一致します(たとえば、価格が>150.00 である IBM のイベント)。この場合、b は、一致したイベントのフィールド buy の値に設定されます。次に、この b の値は、関心のある第 2 のイベント e2 と一致する候補である後続のイベントの WHERE 節を評価する際に使用されます。関心のある各イベントの追加された統一句 and b == buy は、フィールド buy の同じ値が両方の一致するイベントに表示されるようにします。

たとえば、入力スキーマが前述の定義どおりで、次のパターンを定義するとします。

```
<pattern>
  <events>
    <!-- Someone buys or sells IBM at price > 150.00 and
        buys or sells IBM at a price > 152.00 within 5 seconds -->
    <event name="e1"> symbol=="IBM" and price > 150.00</event>
```

```
    <event name="e2"> symbol=="IBM" and price > 152.00</event>
   </events>
   <logic>and{5 seconds}(e1, e2)</logic>
...
</pattern>
```

シンボルが"IBM"で price が"152.10"であるウィンドウにイベントが入ったとします。これは AND 演算子であるため、固有のシーケンスは含まれず、WHERE 句はそのイベントによって演算子のどちらの側でも満たされます。したがって、パターンは True となり、イベント e1 はイベント e2 と同じになります。これはおそらく意図したものではありません。したがって、次のようにパターンを少し変更することができます。

```
<pattern>
  <events>
    <!-- Someone buys or sells IBM at price > 150.00 and <=152.00 and
        buys or sells IBM at a price > 152.00 within 5 seconds -->
    <event name="e1"> symbol=="IBM" and price > 150.00 and price <= 152.00</event>
    <event name="e2"> symbol=="IBM" and price > 152.00)</event>
  </events>
  <logic>and{5 seconds}(e1, e2)</logic>
...
</pattern>
```

これらの変更を行うと、曖昧な 2 つの WHERE 節がなくなり、単一のイベントが両側に一致しないようになります。パターンマッチが発生するためには、2 つの一意のイベントが必要です。

イベント e1 とイベント e2 が互いに 5 秒以内に発生しなければならないように、AND 演算子の一時的な条件を指定するとします。その場合、各イベントの時間的条件は任意です。

演算子ツリーの状態定義

オペレーターツリーは、ストリーミングデータが一連の接続されたオペレーターを介してストリーミングデータがどのように流れるかを定義する階層構造です。各オペレーターは特定のタスクを実行し、結果を下位に渡し、モジュール化された構成ブロックを通じて複雑なイベント処理を可能にします。

演算子ツリーは、次のいずれかの状態を持つことができます。

- 初期-ツリーにイベントが適用されていません。
- 待機中-イベントが適用されて状態が変化したが、左(および該当する場合には右)側の引数は、ツリーが TRUE または FALSE に評価されることをまだ許可していません。
- TRUE または FALSE -ツリーがこれらの論理ブール値のいずれかに評価するのに十分なイベントが適用されています。

演算子サブツリーの状態値は、FIXED であっても FIXED でなくてもよいです。状態値が FIXED の場合、これ以上のイベントは適用されません。状態値が FIXED でない場合、状態値はイベントの適用に基づいて変化する可能性があります。新規作成イベントをサブツリーに適用する必要があります。

パターンインスタンスがマッチをは発生できず、それ自体が破棄されると、フォールドされます。インスタンスは解放され、アクティブパターンインスタンスリストから削除されます。パターンインスタンスの最上位のツリー(ルートノード)がFALSE になると、パターンがフォールドされます。それが TRUE になると、パターンは一致を出し、それ自身を破壊します。

演算子ツリー(*OPT*)は、演算子と EOI のツリーです。*EOI* が関心のあるイベントまたは演算子ツリー(*EOI*|*OPT*)を参照する場合

表 4.2　式と関連する結果

式	結果
not *EOI*	Boolean 否定式です。この式は、*EOI* が TRUE または FALSE と評価されるまで、待機状態のままです。次に、論理否定を実行します。*EOI* が FIXED になったときにのみ FIXED になります。 **注:** 時間の制限がある not を必ず使用することをお勧めします。使用しない場合、何も起こらなくなるまで無期限に待つ場合があります。
not *OPT*	Boolean 否定式です。この式は、*OPT* が TRUE または FALSE と評価されるまで、待機状態のままです。次に、論理否定を実行します。*OPT* が FIXED になったときにのみ FIXED になります。 **注:** 時間の制限がある not を必ず使用することをお勧めします。使用しない場合、何も起こらなくなるまで無期限に待つ場合があります。
notoccur *EOI*	*EOI* を満たさないイベントを適用すると TRUE になりますが、FIXED とマークされません。この式は、より多くのイベントをそれに適用できることを意味します。*EOI* と一致するイベントが表示されるとすぐに、FALSE および FIXED になります。
notoccur *OPT*	使用不可
EO or *EO*	すべての非 FIXED サブツリーに常に適用されるイベントです。この式は、2 つのサブツリーの 1 つが TRUE になると、TRUE になります。両方のサブツリーが FALSE になると、FALSE になります。そのサブツリーの 1 つが TRUE の場合は FIXED であり、そのサブツリーの両方が FALSE でありかつ FIXED でない場合には FIXED です。
EO and *EO*	すべての非 FIXED サブツリーに常に適用されるイベントです。この式は、2 つのサブツリーの両方が TRUE になると、TRUE になります。サブツリーの 1 つが FALSE になると、FALSE になります。そのサブツリーの 1 つが FALSE および FIXED であるか、またはサブツリーの両方が TRUE および FIXED であるときには FIXED です。

式	結果
EO FBY *EO*	右辺(RHS)にイベントを適用する前に最小限の数のイベント適用で左辺(LHS)を完了しようとします。適用ルールは次のとおりです。 ■ LHS が TRUE または FALSE でない場合は、TRUE または FALSE になるまで LHS にイベントを適用します。 ■ LHS が FALSE になったら、それに続く状態を FALSE に設定して FIXED にします。 ■ LHS が TRUE になると、RHS が TRUE または FALSE になるまで、すべてのイベントを RHS に適用します。RHS が FALSE になったら、FBY 状態を FALSE および FIXED に設定します。TRUE になった場合は、FBY 状態を TRUE および FIXED にします。 このアルゴリズムは、FBY パターンを満たすイベントの最小長シーケンスを探します。
is *EOI*	EOI を満たすイベントを適用すると TRUE になり、それ以外の場合は FALSE になります。イベントの最初のアプリケーションで FIXED になります。
is *OPT*	使用不可

表 4.3　一連のサンプル演算子ツリーおよびイベントの基礎となる論理

ロジック	説明
(a fby b)	a, ..., b,を検出します。...は任意のシーケンスです。
(a fby ((notoccur c) and b))	a, ..., b:を検出しますが、a と b の間に c は存在しません。
(a fby (not c)) fby (not (not b))	X が c になることができない場合、a, X, b:を検出します。
(((a fby b) fby (c fby d)) and (notoccur k))	a, ..., b, ..., c, ..., d :を検出しますが、k はシーケンス内のどこにも現れません。
a fby (notoccur(c) and b)	シーケンス内に c の出現がない、a FBY b を検出します。
is(a) fby is(b)	a と b の間に何も置かずに、a FBY b を直接検出します。
a fby (b fby ((notoccur c) and d))	a と b の間に c の出現がなく、a ... b ... d,を検出します。

ロジック	説明
(notoccur c) and (a fby (b fby d))	どこにも c の出現がない、a ... b ... d,を検出します。

パターンの制限

パターンウィンドウで定義するパターンには、次の制限があります。

- キーフィールドのデータ型は int64 でなければなりません。
- 関心のあるイベントは、演算子ツリーの 1 つの位置でのみ使用する必要があります。たとえば、次のコードは ERROR を返します。

  a fby (notoccur(a) and b)

- パターンウィンドウは挿入イベントでのみ機能します。

  更新または削除を生成する入力ウィンドウがある場合は、入力ウィンドウとパターンウィンドウの間にプロシジャウィンドウを配置する必要があります。プロシジャウィンドウは、データを挿入するために非挿入データをフィルタリングまたは変換します。

  パターンもまたインサートだけを生成します。パターンウィンドウによって生成されるイベントは、パターンが検出するように定義されたイベントのシーケンスを正常に検出したことを示すものです。パターンのスキーマは、パターン内の対象イベントから指定する非キーフィールドに加えて、単調に増加するパターン HIT カウントで構成されます。

  dfESPpattern::addOutputField() and dfESPpattern::addOutputExpression()

- 対象のパターンイベントに対して WHERE 句の式を定義する場合、バインディング変数は常に比較の左側になければならず(bindvar == field など)、操作できません。

  たとえば、次の C++コード内で、addEvent ステートメントは無効とフラグされます。

  ```
  e1 = consec->addEvent(readingsWstats, "e1",
      "((vmin < aveVMIN) and (rCNT==MeterReadingCnt) and (mID==meterID))");
  e2 = consec->addEvent(readingsWstats, "e2",
      "((mID==meterID) and (rCNT+1==MeterReadingCnt) and (vmin < aveVMIN))");
  op1 = consec->fby_op(e1, e2,2880000000l);
  ```

  e1 の WHERE 句を考慮してください。これらのイベントの間の演算子は後続のイベントであるため、これは最初にマッチするイベントです。イベントフィールド **vmin** がフィールド **aveVMIN** よりも小さいことを保証します。これが True であれば、変数 **rCNT** を現在のメーター読み取り回数にバインドし、変数 **mID** を **meterID** フィールドにバインドします。

  e2 について考えてみましょう。次の前提条件が満たされていることを確認してください。

  - **meterID** は両方のイベントで同じです。

- **meterReadingCnt** に基づいてメーターの測定値が連続しています。
- 2 番目のイベントの **vmin** は **aveVMIN** より小さいです。

この式の ERROR は、**rCNT** 変数を 1 ずつ増加させ、現在のメーターの示度と比較して、メーターの測定値が連続していたかどうかをチェックしたものです。変数は操作できません。代わりに、変数をきれいに保つために比較の右側に操作を限定します。

次のコードは、このチェックを行う正しい方法を示しています。メーターの測定値が連続していることを確認します(変数を増分するのではなく、現在のイベントのメーター読み取りフィールドを減らしていることを前提にします)。

```
e1 = consec->addEvent(readingsWstats, "e1",
    "((vmin < aveVMIN) and (rCNT==MeterReadingCnt) and (mID==meterID))");
e2 = consec->addEvent(readingsWstats, "e2",
    "((mID==meterID) and (rCNT==MeterReadingCnt-1) and (vmin < aveVMIN))");
op1 = consec->fby_op(e1, e2,2880000000l);
```

ステートレスパターンウィンドウの使用

パターンウィンドウは、入力ウィンドウと出力ウィンドウの両方に対して挿入専用です。パターンウィンドウの出力は、パターンウィンドウで見つかったパターンの数を表す単調に増加する整数 ID です。ID の後に、パターンの対象イベントのフィールドから組み立てられた任意の数の非キーフィールドが続きます。

パターンウィンドウの入力と出力は共に無制限で挿入専用なので、ステートレスウィンドウ(つまり、インデックスタイプ pi_EMPTY のウィンドウ)の自然な候補です。通常、保持ポリシーが設定されたコピーウィンドウには、挿入専用のウィンドウが表示されます。

パターンウィンドウは自動的に挿入専用としてマークされます。それらは挿入ではないレコードを拒否します。したがって、パターンウィンドウで pi_EMPTY のインデックスタイプを使用する場合は、問題は発生しません。ソースウィンドウがパターンウィンドウに供給している場合、dfESPwindow::setInsertOnly()呼び出しを使用して、挿入専用であることをソースウィンドウに明示的に伝える必要があります。これにより、ソースウィンドウは挿入以外の演算コードを持つイベントを拒否し、pi_EMPTY のインデックスタイプを使用できるようにします。

ステートレスウィンドウは、メモリ使用に関して効率的です。わずかなメモリ使用量(合計メモリが 500MB 未満)でも、10 億件を超えるパターン検出シナリオのイベントを実行します。

```
Source Window [insert only, pi_EMPTY index] --> PatternWindow[insert only,
    pi_EMPTY index]
```

ハートビート間隔の有効化

TIME アウトになるパターンは、システムによってハートビートに送信されます。何百万もの未完了のパターンが開いている場合、デフォルトの 1 秒のハートビート間

隔は短すぎます。この場合、システムは 1 秒ごとにすべてのパターンをタイムアウトさせようとするため、システムのパフォーマンスが低下する可能性があります。

この問題を解決するには、ハートビート間隔を調整します。

- XML では、project 要素で次の属性を使用します。heartbeat-interval='*number-of-seconds*'
- C++では、プロジェクトを開始する前に dfESPproject::setHeartbeatInterval(int *number-of-seconds*)を呼び出します。

実用的な*秒数*に設定してください。

インデックス生成関数の使用

インデックス生成関数は、パターンが適用される前にイベント内のフィールドを選択してソートします。たとえば、次の EOI とパターンロジックを考えます。

```
<event name='e1'>(sym == symbol) and (price > 100)</event>
<event name='e2'>(sym == symbol) and (price < 100)</event>
<logic>fby(e1,e2)</logic>
```

ここでは、同じシンボルを持つイベントを検出し、価格が最初に 100 を超え、次に 100 を下回ったときにそれらを追跡しています。sym は EOI の一部であるため、パターンウィンドウに流れ込むすべてのイベントは、このパターンのすべてのインスタンスに対してチェックされます。多くのイベントに適用されると、その評価はプロジェクトのパフォーマンスを低下させる可能性があります。

あるいは、パターンでインデックス sym を生成できます。

```
<window-pattern...index='sym'>
```

この場合、すべてのイベントはインデックスであるため、まず sym 別にソートされます。その後、パターンはイベントに適用されます。sym='IBM'の場合、インデックスに一致するイベントのみが評価されます。したがって、EOI を単純化できます。

```
<event name='e1'>price > 100</event>
<event name='e2'>price < 100</event>
```

パターンウィンドウには評価するイベントが少ないため、このプロセスによりパフォーマンスが向上します。

カウンターウィンドウの使用

カウンターウィンドウを使用して、モデルを通ってストリーミングされているイベントの数とそれらが処理されているレートを確認します。

カウンターウィンドウに関連付けられている XML エレメントについては、“window-counter” (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

カウンターウィンドウのスキーマは構成できません。次の形式でイベントを生成するようにハードコードされています。

"*input**:string,totalCount:int64,totalSeconds:double,totalRate:double,intervalCount:int64,intervalSeconds:double,intervalRate:double" 。

input の値は、イベントをカウンターウィンドウに送信したウィンドウの名前です。残りのフィールドは、パフォーマンス統計をレポートします。

表 4.4　出力スキーマフィールド

名前	説明
totalCount	プロジェクトの開始以降(またはカウントがクリアされて以降)ウィンドウに入ったイベントの数。
totalSeconds	プロジェクトの開始以降(またはカウントがクリアされて以降)ウィンドウがイベントを受信した秒数。
totalRate	totalCount / totalSeconds
intervalCount	指定された間隔でウィンドウに入ったイベントの数。
intervalSeconds	間隔の実際の秒数。この値は、指定された間隔に近いですが、まったく同じではありません。
intervalRate	intervalCount / intervalSeconds

count-interval を指定すると、カウンターウィンドウは、パフォーマンス統計をレポートするイベントをその間隔で定期的に生成します。最後のイベントが生成されてからカウンターウィンドウがイベントを受信していない場合、新しいイベントは生成されません。

```
<window-counter name='counter'
                count-interval='2 seconds'
                clear-interval='30 seconds'/>
```

```
<event opcode='upsert' window='trades/trades/counter'>
      <value name='input'>trades</value>
      <value name='intervalCount'>288215</value>
      <value name='intervalRate'>144108</value>
      <value name='intervalSeconds'>2</value>
      <value name='totalCount'>794312</value>
      <value name='totalRate'>132385</value>
      <value name='totalSeconds'>6</value>
   </event>
```

count-interval を指定しない場合、値はデフォルトで 1 秒に設定され、全体の値のみを含むイベントが生成されます。

```
<window-counter name='counter'/>
```

```
<event opcode='upsert' window='trades/trades/counter'>
      <value name='input'>trades</value>
      <value name='totalCount'>7815189</value>
      <value name='totalRate'>132461</value>
      <value name='totalSeconds'>59</value>
   </event>
```

生成されるイベントの演算コードは、カウンターウィンドウのインデックスに基づいています。インデックスが pi_EMPTY の場合、演算コードは挿入です。他のインデックス値の場合、演算コードはアップサートです。

カウンターウィンドウを使用するには、カウンターウィンドウをターゲットとしてエッジを追加し、モニターするウィンドウをソースとして追加します。複数のウィンドウを同じカウンターウィンドウに接続することができます。結果を表示するには、フォーマットされたイベントを標準出力に出力する<contquery>エレメントの TRACE 属性にカウンターウィンドウを追加することもできます。

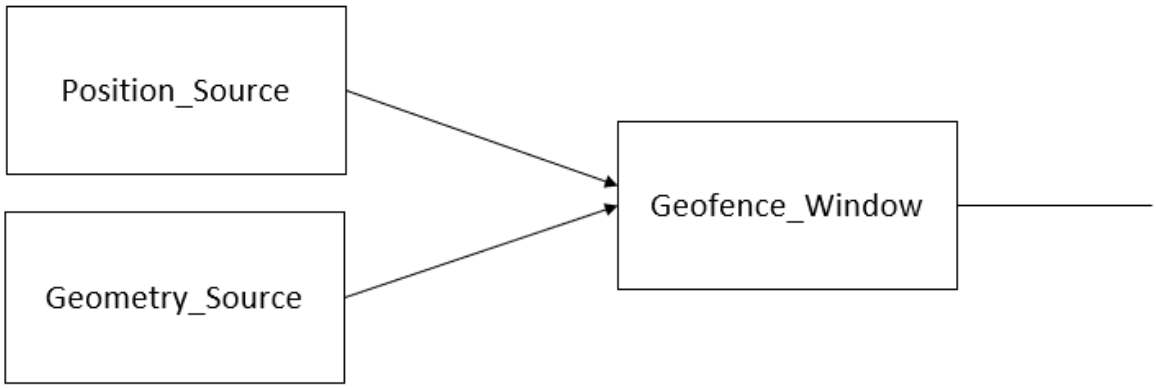
ジオフェンスウィンドウの使用

ジオフェンスウィンドウの概要

*ジオフェンス*は、実際の地理的領域の仮想境界です。特定の場所を中心とした半径としてジオフェンスを動的に生成することも、特定の境界線のセットとしてジオフェンスを作成することもできます。ジオフェンスウィンドウは、イベントストリームの場所が関心領域の内側にあるのか、関心領域に近いのかを判断します。場所の詳細を使ってイベントを補強することができます。

ジオフェンスウィンドウには 2 つの入力ウィンドウが必要です。1 つはストリーミングイベントをインジェクトするためのもの、もう 1 つはジオフェンス領域と場所をインジェクトするためのものです。デフォルトでは、イベントをインジェクトするウィンドウを、プロジェクトで指定する最初のエッジを使用してジオフェンスウィンドウに接続します。ジオフェンス領域と位置をインジェクトするウィンドウを 2 番目のエッジを使用してジオフェンスウィンドウに接続します。

```
<edges>
  <edge source='position_source' target='geofence_window'/>
  <edge source='geometry_source' target='geofence_window'/>
</edges>
```



したがって、ジオフェンスウィンドウは、外部結合ウィンドウまたはルックアップ操作のように動作します。イベントはストリーミング側にあり、ジオフェンス領域と場所はディメンション側にあります。

または、入力ウィンドウをジオフェンスウィンドウに接続するエッジ、**position** または **geometry** に役割を明示的に割り当てることができます。たとえば、

```
<edges>
  <edge source='geometry_source' target='geofence_window' role='geometry'/>
  <edge source='position_source' target='geofence_window' role='position'/>
</edges>
```

ジオフェンスウィンドウは、デカルト座標系または地理座標系をサポートするように設計されています。唯一の要件は、すべての座標が一貫していなければならず、同じ空間または投影を参照する必要があることです。地理座標の場合、座標は(X、Y)直交座標(経度、緯度)で指定する必要があります。すべての距離はメートルで定義され、算出されます。

入力位置メタデータに応じて、ジオフェンス検索を選択済みジオメトリに制限できます。たとえば、プロジェクトが複数の自動車の位置を識別するイベントを受け取った場合、特定のサブセットに関連するジオメトリジオフェンスのみを検索するよう選択できます。geometry 入力および position 入力の join-key-fieldname 属性を使用してこれを行います。属性値は、ジオフェンス検索が実行される前に比較および評価されます。

ジオフェンスウィンドウに関連付けられている XML エレメントについては、*“window-geofence” (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*および*“ジオフェンスウィンドウに関連する XML 言語要素” (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*を参照してください。

次の ESP サーバープロパティを使用して、ジオフェンスウィンドウの出力マップでジオメトリタイプのローカライズされた値を指定することができます。

表 4.5 ジオフェンスのプロパティ

プロパティ	デフォルト値
Geofence0042	**円**
Geofence0043	**多角形**

プロパティ	デフォルト値
Geofence0044	**ポリライン**
Geofence0045	**不明**

ジオメトリ

概要

関心領域および位置は、*ジオメトリ*として定義されます。ジオフェンスウィンドウでは、多角形、ポリライン、円のジオメトリがサポートされています。ジオメトリはイベントとしてパブリッシュされ、ジオメトリごとに 1 つのイベントがパブリッシュされます。

geometry エレメントの geotype-fieldname 属性を使用して、どのタイプのジオメトリを含めるかを指定します。この属性が指定されていない場合、ジオフェンスウィンドウはデータ特性に基づいてタイプを自動的に決定します。データが指定されたタイプと一致しない場合(たとえば、データが非閉環だが geotype-fieldname='polygon'を指定している)、ジオメトリは拒否されます。

ジオフェンスウィンドウでは、ジオメトリを動的に更新できる挿入、更新、および削除の演算コードがサポートされています。各ジオフェンスウィンドウのインスタンスは、多角形ジオメトリ、ポリラインジオメトリ、または円ジオメトリを実行でき、すべての 種類のジオフェンス解析を同時に実行できます。

ジオフェンスウィンドウは、ルックアップ結合のように動作することに注意してください。ジオメトリ入力ウィンドウから来るフィールドの全部または一部を使用して、その出力スキーマを構築する必要があります。出力スキーマの詳細については、“出力スキーマ”を参照してください。

多角形

*多角形*は、関心領域を表す平面形状です。ジオフェンスウィンドウは、多角形、マルチ多角形、穴または複数のリングを持つ多角形をサポートします。

多角形のリングを表す位置座標のリストとして多角形を定義します。*リング*とは、位置座標の閉じたリストです。閉じたとみなされるためには、リングリストの最後の点は最初の点と同じでなければなりません。たとえば、正方形のように 4 つの点で幾何学的に定義されたリングは、5 つの位置座標を宣言しなければならず、最後の座標は最初の座標と同じでなければなりません。

入力多角形ウィンドウスキーマには、少なくとも次の 2 つのフィールドが必要です。

- INT64 種類または文字列種類の単一のキーフィールド。このフィールドは、多角形の ID を定義します。
- array(dbl)型のデータフィールド。

Double 型の半径フィールドオプションを指定することもできます。このフィールドは、多角形の周囲の近接距離を表します。このフィールドが指定されていない場合は、パラメーター radius で定義されているデフォルト距離が使用されます。この値は、プロパティ proximity-analysis が **true** に設定されている場合にのみ使用されます。

多角形が閉じておらず、閉じたリングを含まない場合は、ポリラインとみなされます。

たとえば、次の文字列は 4 つの点で構成される多角形を表します。5 番目の数字のペアが最初のものと同じであることに注目してください。

5.281 9.455 3.607 7.112 6.268 6.181 8.414 7.705 5.281 9.455

複数のリングを持つ多角形の場合、定義される最初のリングは外部リングでなければならず、他のリングは内部リングまたは穴でなければなりません。

注: 多角形の定義に重複する穴はサポートされていません。

スキーマには、オプションの説明フィールドを含めることもできます。その他のフィールドはすべて 無視されます。

ジオフェンスウィンドウは、多角形で作業する場合、ストリーミングウィンドウから来る各イベント位置を分析し、この位置が内部にある多角形を返します。複数の一致するジオメトリがある場合(多角形が重なっている場合)、オプション output-multiple-results が true に設定されている場合、複数のイベントが生成されます(ジオメトリごとに 1 つ)。

さらに、プロパティ近接分析が **true** に設定されている場合、ジオフェンスウィンドウは、半径プロパティ値で定義された距離内にある多角形の ID を返します。距離は、位置から最も近い外側のリングセグメントまで測定されます(穴は無視されます)。

多角形が検出されると、出力 geodistance-fieldname は次を返します。

- polygon-compute-distance-to プロパティが**セグメント**に設定されている場合、位置から多角形までの正確な距離になります。この値は、位置が多角形内にあるときは負で、外側にあるときは正の値になります。近接解析を true に設定すると、パターンウィンドウを使用して符号を変更することで、多角形の交差を簡単に検出できます。
- polygon-compute-distance-to プロパティが **centroid** に設定されている場合、位置から多角形の重心までの正確な距離。

ポリライン

ポリラインは、境界線またはトリップワイヤを表す形状です。ポリラインのセグメントまたはセグメントを表す座標のリストとしてポリラインを定義します。セグメ

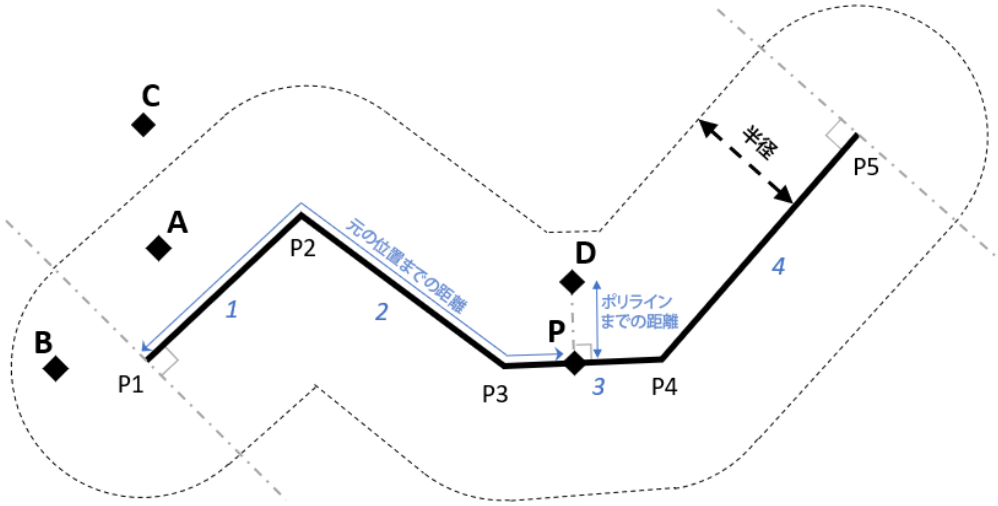
ント定義内のポイントのシーケンスは、ポリラインのベクトル方向とその左右の側を定義します。

ポリラインで作業する場合、ジオフェンスウィンドウは位置ソースウィンドウからの各イベント位置を分析します。radius プロパティ値で定義された距離内にあるポリラインの ID を返します。距離は、位置から最も近いセグメントまで測定されます。

ポリラインが検出されると、出力 geodistance-fieldname はイベント位置からポリラインまでの正確な距離を返します。この値は、位置がポリラインの左側にある場合は負の値になります。右側にあるときは正の値です。トリップウィングの交差点は、パターンウィンドウを使用した標識の変更によって簡単に検出できます。

次のダイアグラムを考えてみましょう。ポリラインは太い黒い線で描かれています。ポリラインの周囲の破線は、半径の範囲内の面積です。A、B、C、D は受信イベント位置です。

図 4.1　複数のイベント位置に関連するポリライン



イベント位置 A とポリラインの間の距離は半径未満です。その場合、ジオフェンスウィンドウはポリライン ID を返します。

strict-projection='false'の場合、イベント位置 B はポリラインに近くなります。その場合、ポリライン ID が返されます。strict-projection='true'の場合、B はポリライン近くから外れます。ポリライン ID は返されません。

イベント位置 C はポリライン近くから外れています。ポリライン ID は返されません。

イベントポジション D に関して:

- P は位置 D の投影点です。P の座標は、projection-fieldname パラメーターで指定されたフィールドによって返されます。
- D と P の間の距離は、geodistance-fieldname パラメーターで指定されたフィールドによって返されます。
- P から P3、P2 から P1 は元の位置までの距離です。その距離は、distance-to-origin-fieldname パラメーターで指定されたフィールドによって返されます。

- セグメント番号は 3 です。これは、segmentnumber-fieldname パラメーターで指定されたフィールドによって返されます。

projection-fieldname、geodistance-fieldname、distance-to-origin-fieldname、segmentnumber-fieldname パラメーターの詳細については、"output" (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

円

*円*は関心のある位置の位置を包含します。3 つの値で円を定義します。

- 円の中心を表す 2 つの座標 X と Y(経度と緯度)
- 中心からの半径距離

入力円ジオメトリウィンドウスキーマの次の 3 つのフィールドが必須です。

- INT64 種類または文字列種類の単一のキーフィールド。このフィールドは、円の ID を定義します。
- 次のいずれかの操作を行います。
  - 円の中心の X 座標と Y 座標を含む Double 型の 2 つの座標フィールド
  - スペースで区切られた、X、Y の順序で数値として定義された座標を持つ、Array(dbl)型、String 型、または RString 型のデータフィールド

スキーマには、次のオプションフィールドもあります。

- 中心点位置の周りの円形領域を表す半径フィールド(double)。このフィールドを指定しない場合は、パラメーター radius で定義されたデフォルト距離が使用されます。
- 説明フィールド。

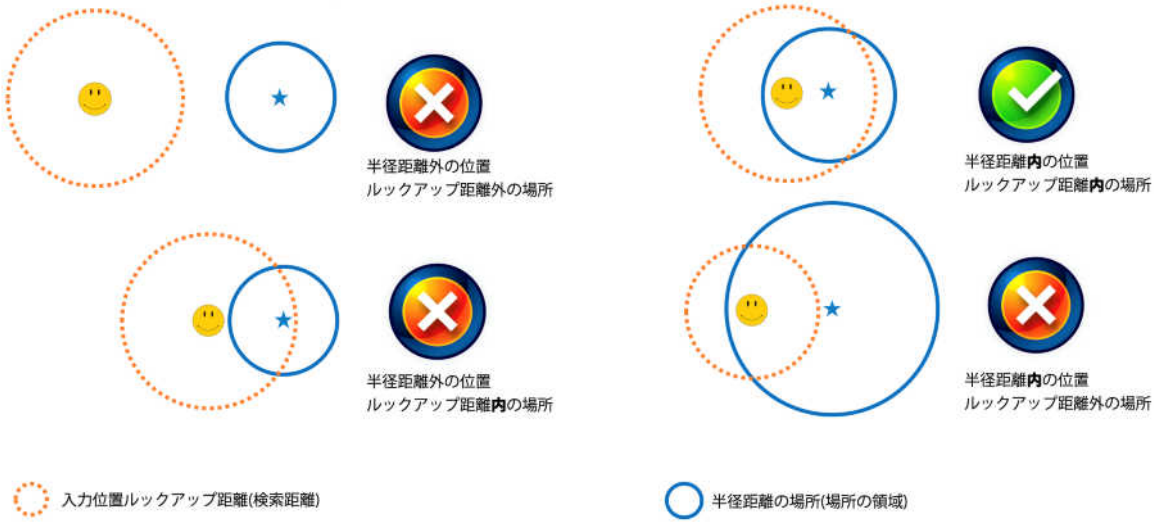
その他のフィールドはすべて無視されます。

円の作業をする場合、ジオフェンスウィンドウはストリーミングウィンドウからの各イベント位置を分析し、次の基準に一致する円 ID を返します。

- 位置ルックアップ距離を 0 に設定すると、位置は単純な点のように振る舞います。それは円の内外にあります。それが円の中にあれば、一致があります。
- 同様に、円のジオメトリの場合、円の半径を 0 に設定すると、円はベアポイントのように動作します。このポイントは、一致を返すために位置ルックアップ距離エリアになければなりません。
- 位置ルックアップ距離および円半径の他の値については、位置および円の中心が互いの距離内になければなりません。それ以外の場合、一致は返されません。位置は円内にあり、円の中心と位置の間の距離はルックアップ距離よりも小さくなっています。
- 位置ルックアップ距離と円半径の両方が 0 に等しい場合、それらは一致を有する正確な同じ点でなければなりません。

位置ルックアップ距離は、追加イベント入力フィールド値またはパラメーター lookupdistance によって定義されます。

図 4.2　円ジオメトリルックアップロジック



ルックアップ距離に複数の一致するジオメトリがあり、オプション output-multiple-results が true に設定されている場合、複数のイベントが生成されます(ジオメトリごとに 1 つ)。

位置ウィンドウスキーマ

位置入力ウィンドウのスキーマには、出力スキーマに伝播される任意の種類および数のフィールドを含めることができます。ジオフェンスウィンドウでは、次のスキーマフィールドが使用されます。

- X と Y または経度と緯度の座標を含む Double 種類の 2 つの必須座標フィールド。
- 現在の位置イベントのルックアップ距離を含む Double タイプのオプションフィールド。このフィールドは、円のジオメトリを使用するように設定されたジオフェンスウィンドウでのみ使用されます。

その他のフィールドはすべて 無視され、出力スキーマに伝播されます。

出力スキーマ

ジオフェンスウィンドウはルックアップ結合のように動作し、次のフィールドで次の順序で出力スキーマは自動的に定義されます。

- それぞれの順序で入力位置ウィンドウから来るすべてのフィールド。
- ジオメトリの ID を受け取る int64 型または string 型の必須フィールド。ジオメトリが見つからない場合、生成されたイベントではこのフィールドの値はヌルです。このフィールドは、パラメーター geoid-fieldname によって定義されます。

出力スキーマには、次のフィールドが追加されています。

- ジオメトリウィンドウのスキーマにジオメトリの説明がある場合に、ジオメトリの説明を受け取るフィールド。このフィールドの存在と名前は、パラメーター geodesc-fieldname によって定義されます。
- 位置からジオメトリまでの距離を受け取るフィールド(double)。このフィールドの値は、次のいずれかです。
  - 円の中心までの距離
  - ポリラインの最も近いセグメントまでの距離
  - 多角形の重心または最も近いセグメントまでの距離

  このフィールドの存在と名前は、パラメーター geodistance-fieldname によって定義されます。
- output-multiple-results が true に設定されている場合、一致するジオメトリのイベント番号を受け取るタイプ Int64 の必須キーフィールド。このフィールド名は、パラメーター eventnumber-fieldname によって定義されます。
- ジオメトリタイプを受け取る string 型のフィールド。このフィールドの存在と名前は、パラメーター geotype-fieldname によって定義されます。
- ポリラインの最も近いセグメント上の投影点の座標を受け取る 2 つのフィールド(倍精度)。これらのフィールドの値は、ポリライン以外のジオメトリタイプではヌルです。これらのフィールドの存在と名前は、パラメーター projection-fieldname によって定義されます。これらのフィールドの名前の前には、それぞれ_X と_Y が付きます。
- ポリラインの最も近いセグメント上の投影点のセグメント番号を受け取るフィールド(int32)。その値は、ポリライン以外のジオメトリタイプの場合ヌルです。このフィールドの存在と名前は、パラメーター segmentnumber-fieldname によって定義されます。
- セグメントのポリラインの長さを受け取るフィールド(array(double))。サイズは、地理座標の場合はメートル、デカルト座標の場合は単位です。その値は、ポリライン以外のジオメトリタイプの場合ヌルです。このフィールドの存在と名前は、パラメーター segmentsizes-fieldname によって定義されます。
- ポリラインに沿って投影点からポリラインの最初の点までの距離を受け取るフィールド(double)。その値は、ポリライン以外のジオメトリタイプの場合ヌルです。このフィールドの存在と名前は、パラメーター distance-to-origin-fieldname によって定義されます。
- ジオメトリ入力スキーマから伝播されたフィールドのリスト。これらのフィールドの存在と名前は、パラメーター include-geo-fields によって定義されます。

メッシュインデックス

ジオフェンスウィンドウは、高速かつ待機時間ルックアップ処理のために、空間データ構造を使用する最適化メッシュインデックスアルゴリズムを実装しています。この構造は、スペースを *セル*と呼ばれる格子形状のバケットに細分します。この構造は、使用している座標系とは無関係であり、任意の種類のデカルト座標、地理座標または投影座標空間をシームレスに使用することができます。

メッシュアルゴリズムは、空間細分のスケールを定義するパラメーター(*メッシュ率*)を使用します。このメッシュ率は、使用されている座標単位の 10 の累乗を表す[-5,5]範囲内の整数です。

たとえば、デフォルトの係数 0 は座標単位ごとに 1 つの細分を生成します。因子 1 は、10 ユニットあたりの細分を生成します。係数-1 は、1 単位あたり 10 の小区画を生成します。この係数は、X 軸と Y 軸の両方に、または各軸に対して独立して設定できます。

四角形の多角形を表す次の座標セットを考えてみましょう。多角形を閉じて、終わりに繰り返される最初の点に注意してください。

[(1001.12,9500.12) (1001.12,9510.12) (1010.12,9510.12) (1010.2,9500.12) (1001.12,9500.12)]

- メッシュ率が 1 の場合、ジオフェンスウィンドウは座標を 10^1 で割ります。この結果、[(100,950) (100,951) (101,950) (101,951)]となり、このジオメトリに対して 4 つのメッシュセル、つまり(101-100+1)*(951-950+1)=4 が作成されます。
- 因数が 2 の場合、(10-10+1)*(95-95+1)=1 メッシュのセルが作成されます。
- メッシュ率を-1 に設定すると、ウィンドウは 9,191 メッシュのセルを作成します。これにより、(10101-10011+1)*(95101-95001+1)=91*101=9191 という超過メッシュになります。

最適なパフォーマンスを得るには、メッシュ率を空間カバレッジおよびロードされたジオメトリの数に適合させる必要があります。ジオメトリあたりのメッシュセルの数が多すぎると、ジオメトリの取り込みが遅くなり、オーバーサイズのインデックスが生成されます。ジオメトリあたりのメッシュセルの数が少なすぎると、ルックアップ処理が遅くなり、ストリームのパフォーマンスと待機時間に影響します。

専用パラメーターの max-meshcells-per-geometry は、大きすぎるメッシュの作成を避けるためにジオメトリごとに作成される最大許容メッシュセルを設定します。これにより、無駄な集中算出が発生します。ジオメトリがこの制限を超えると、拒否されます。非常に広い領域をカバーする場合は、ジオメトリあたりより高いメッシュ係数を設定するか、より高い最大メッシュセル数を設定することを検討してください。

ジオフェンスウィンドウには、取り込まれた最初の 1,000 個のジオメトリを解析して適切なメッシュ率を自動的に計算して設定する内部アルゴリズムが用意されています。これはデフォルトで自動的にアクティブになります。メッシュ率を手動で設定するには、パラメーター autosize-mesh を false に設定します。

例

```
<window-geofence name="geofence_win" index="pi_HASH">

 <geofence
   coordinate-type="geographic"
   log-invalid-geometry="true"
   output-multiple-results="true"
   output-sorted-results="false"
   max-meshcells-per-geometry="100"
   autosize-mesh="true"
 />
```

```
<geometry
  data-fieldname="GEO_data"
  desc-fieldname="GEO_desc"
  x-fieldname="GEO_x"
  y-fieldname="GEO_y"
  radius-fieldname="GEO_radius"
  radius="0"
  data-separator=" "
/>

<position
  x-fieldname="GPS_longitude"
  y-fieldname="GPS_latitude"
  lookupdistance="110"
/>

<output
  geoid-fieldname="GEO_id_out"
  geodesc-fieldname="GEO_desc_out"
  eventnumber-fieldname="event_nb"
  geodistance-fieldname="GEO_dist"
/>

</window-geofence>
```

プロシジャウィンドウの使用

概要

プロシジャウィンドウを使用して、入力ウィンドウの任意の数や各入力ウィンドウの入力ハンドラー関数を指定します。プロシジャウィンドウの入力ハンドラーは、共有ライブラリの呼び出し可能な C++関数を使って定義することができます。この種類の入力ハンドラーを定義するには、cxx-pluginXML エレメントを使用します。

注: プロシジャウィンドウは、SAS Micro Analytic Service モジュールとストアの使用をサポートしていません。そのサポートは、計算ウィンドウで提供されています。

入力イベントが到着すると、一致するウィンドウに登録された入力ハンドラーが呼び出されます。次に、その入力ハンドラー関数によって生成されたイベントが生成されます。

プロシジャウィンドウの状態をハンドラー間で共有するために、インスタンス固有のコンテキストオブジェクト(dfESPpcontext)がハンドラー関数に渡されます。各ハンドラーは、コンテキストオブジェクトにあるものにフルアクセスできます。ハンドラーは、他のハンドラーが使用するために、または将来の呼び出し時に使用するために、このコンテキストにデータを格納できます。

プロシジャウィンドウに関連付けられている XML エレメントについては、"window-procedural" (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)および"プロシジャウィンドウに関連する XML 言語要素" (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

C++ウィンドウハンドラーの使用

次は、C++で書かれたプロシジャウィンドウハンドラーのシグネチャの例です。

```
typedef bool (*dfESPevent_func)(dfESPpcontext *pc,
          dfESPschema *is, dfESPeventPtr nep,
           dfESPeventPtr oep, dfESPschema *os,
          dfESPptrVect<dfESPeventPtr>&oe);
```

プロシジャコンテキストはハンドラーに渡されます。入力スキーマ、新規作成イベント、および古いイベント(更新の場合)は、呼び出されるとハンドラーに渡されます。最終的なパラメーターは、出力イベント(プロシジャウィンドウが生成するイベントの構造)のスキーマと、出力イベントのベクトルへの参照です。ハンドラーが計算されたイベントをプッシュする必要があるのは、このベクトルです。

1 つの入力ウィンドウしか定義されていないため、1 つのハンドラー関数を定義し、レコードが到着したときに呼び出されます。

```
// This handler functions simple counts inserts, updates,
//     and deletes.
// It generates events of the form "1,#inserts,#updates,
//     #deletes"
//
bool opcodeCount(dfESPpcontext *mc, dfESPschema *is,
                 dfESPeventPtr nep, dfESPeventPtr oep,
                 dfESPschema *os, dfESPptrVect
                 <dfESPeventPtr>& oe) {

   derivedContext *ctx = (derivedContext *)mc;
   // Update the counts in the past context.
   switch (nep->getOpcode()) {
      case dfESPeventcodes::eo_INSERT:
        ctx->numInserts++;
        break;
      case dfESPeventcodes::eo_UPDATEBLOCK:
        ctx->numUpdates++;
        break;
      case dfESPeventcodes::eo_DELETE:
        ctx->numDeletes++;
        break;
    }

   // Build a vector of datavars, one per item in our output
   //     schema, which looks like: "ID*:int32,insertCount:
   //     int32,updateCount:int32,deleteCount:int32"

dfESPptrVect<dfESPdatavarPtr> vect;
os->buildEventDatavarVect(vect);
```

```
// Set the fields of the record that we are going to produce.

vect[0]->setI32(1); // We have a key of only 1, we keep updating one record.
vect[1]->setI32(ctx->numInserts);
vect[2]->setI32(ctx->numUpdates);
vect[3]->setI32(ctx->numDeletes);

// Build the output Event, and push it to the list of output
//     events.

dfESPeventPtr ev = new dfESPevent();
ev->buildEvent(os, vect, dfESPeventcodes::eo_UPSERT,
            dfESPeventcodes::ef_NORMAL);
oe.push_back(ev);

// Free space used in constructing output record.
vect.free();
return true;
```

次の例は、これがプロシジャウィンドウウィンドウにどのように収まるかを示しています。

```
<project name='project' pubsub='auto' threads='1'>
  <contqueries>
   <contquery name='contQuery' trace='proceduralWindow'>
    <windows>
     <window-source name='sourceWindow'>
      <schema>
       <fields>
        <field name='ID' type='int32' key='true'/>
        <field name='symbol' type='string'/>
        <field name='quantity' type='int32'/>
        <field name='price' type='double'/>
       </fields>
      </schema>
      <connectors>
       <connector class='fs' name='publisher'>
        <properties>
         <property name='type'>pub</property>
         <property name='fstype'>csv</property>
         <property name='fsname'>input.csv</property>
         <property name='transactional'>true</property>
         <property name='blocksize'>1</property>
        </properties>
       </connector>
      </connectors>
     </window-source>
     <window-procedural name='proceduralWindow'>
      <schema>
       <fields>
        <field name='ID' type='int32' key='true'/>
        <field name='insertCount' type='int32'/>
        <field name='updateCount' type='int32'/>
        <field name='deleteCount' type='int32'/>
       </fields>
      </schema>
```

```
    <cxx-plugin-context name='plugin' function='get_derived_context'/>
    <cxx-plugin source='sourceWindow' name='plugin' function='countOpcodes'/>
   </window-procedural>
  </windows>
  <edges>
   <edge source='sourceWindow' target='proceduralWindow'/>
  </edges>
 </contquery>
</contqueries>
</project>
```

注: registerMethod クラスは次の関数で構成されています。

- registerMethod_SO
- registerMethod_DSEXT

これらの関数の詳細については、**$DFESP_HOME/doc/html/index.html** にある完全なクラスとメソッドのドキュメントを参照してください。

プロシジャウィンドウがソースウィンドウ(sw)からイベントを見るたびに、コンテキスト mc を使用してハンドラー opcodeCount を呼び出し、出力イベントを生成します。

アプリケーションは、プロシジャウィンドウの dfESPengine::logBadEvent()メンバー関数を使用して、無効であると判断したイベントをログに記録できます。たとえば、この関数を使用して、モデルがデータ品質チェックを実行し、合格しないイベントを記録することを許可することができます。イベントを拒否する一般的な理由は 2 つあります。

- イベントには、キーフィールドの 1 つにヌル値が含まれています。
- 指定された演算コードが既存のウィンドウインデックス(たとえば、同じキーの 2 つの挿入キー、または存在しないキーの削除)と競合します。

SAS Micro Analytics Service モードで実行する DS2 テーブルサーバーコードの変換

以前のバージョンの SAS Event Stream Processing では、テーブルサーバーモードで実行される DS2 コードを使用したプロシジャウィンドウ入力ハンドラーがサポートされていました。この機能のサポートはリリース 5.2 で廃止されました。モデルで DS2 コードを実行するには、プロジェクトレベルで SAS Micro Analytic Service モジュールを定義し、算出ウィンドウで SAS Micro Analytic Service マップを定義する必要があります。

DS2 コードのテーブルサーバーモードでの実行と SAS Micro Analytic Service マップでの実行の主な違いは次のとおりです。

- テーブルサーバーでは、シンボルに遅延バインディングが使用されました。すべての入力フィールドは、常に DS2 メソッドに渡されました。DS2 メソッドの宣言変数はすべて、プロシジャウィンドウのスキーマ内のフィールド名と一致していれば、派生イベントにエクスポートされました。

- SAS Micro Analytic Service モジュールでは、関数呼び出しと同様のインターフェイスが使用されます。ここでは、DS2 メソッドのシグネチャで宣言した入力フィールドのみがメソッドに渡されます。in_out として宣言されたメソッドパラメーターのみが DS2 メソッドからエクスポートされ、対応する名前付き出力フィールドに割り当てられます。名前とタイプの出力フィールドと一致する入力フィールドは、算出ウィンドウにエコーされます。一致したフィールドを明示的に DS2 メソッドに渡す必要はありません。
- SAS Micro Analytic Service で実行される DS2 メソッドでは、SAS Micro Analytic Service のスレッドとパッケージで共有できる読み取り/書き込み共有ベクトルがサポートされています。これにより、複数のスレッドを使用する場合、高速の同時データ共有が可能になります。

次の例は、テーブルサーバーモードで実行されているコードを、SAS Micro Analytics Service モジュールで実行できるコードに変換する方法を示しています。次のソースウィンドウスキーマがあるとします。

```
id*:int64,sensorName:string,sensorValue:double
```

次の出力ウィンドウスキーマがあるとします。

```
id*:int64,sensorName:string,sensorValue:double,absValue:double,sqrtValue:double
```

この DS2 コードがテーブルサーバーモードにあるとします。

```
ds2_options cdump;
  data esp.out;
  dcl double absValue;
  dcl double sqrtValue;
  method run();
    set esp.in;
      absValue  = abs(sensorValue);
      sqrtValue = sqrt(sensorValue);
    end;
  enddata;
```

SAS Micro Analytics Service モジュールでは、次の DS2 コードを使用します。

```
ds2_options sas;
  package p1/overwrite=yes;
    method compute(double sensorValue, in_out double absValue, in_out
double_sqrtValue);
      absValue  = abs(sensorValue);
      sqrtValue = sqrt(sensorValue);
    end;
  endpackage;
```

DS2 コードで演算コードが処理されるときは、演算コード値を整数から文字列に変換してから SAS Micro Analytic Service モジュールで使用する必要があります。

演算コード	テーブルサーバーモード(整数)	SAS Micro Analytic Service モジュール(文字列)
Insert	1	insert
Update	2	更新

演算コード	テーブルサーバーモード(整数)	SAS Micro Analytic Service モジュール(文字列)
Delete	3	delete
Upsert	4	upsert
Safe Delete	5	safedelete

イベントフラグも変換する必要があります。

フラグ	テーブルサーバーモード(整数)	SAS Micro Analytic Service モジュール(文字列)
標準	1	N
保持	4	R

たとえば、次の DS2 コードがテーブルサーバーモードで実行されているとします。

```
<ds2-tableserver source="Aggregate1">
  <code>
  <![CDATA[ds2_options cdump;
        data esp.out;
          method run();
            set esp.in;
            if _opcode = 2 or _opcode = 3 then
              _opcode = 1;
          end;
        enddata;]]>
  </code>
</ds2-tableserver>
```

次のコードでは、同じ評価が実行され、同じ結果が生成されます。

```
<mas-module module="opcode_module" language="ds2" func-names="convert_opcode">
 <code>
 <![CDATA[ds2_options sas;
 package opcode_module/overwrite=yes;
   method convert_opcode(varchar(16) _inOpcode, in_out varchar _outOpcode);
     if (_inOpcode = 'delete' or _inOpcode = 'update') then
        _outOpcode = 'insert';
     else
        _outOpcode = _inOpcode;
     end;
 endpackage;]]>
 </code>
</mas-module>
```

関数呼び出しベースですが、SAS Micro Analytic Service モジュールで使用される DS2 コードでは、入力イベントごとに、0、1 つ、または複数の出力イベントを生成できます。

オブジェクトトラッカーウィンドウの使用

オブジェクトトラッカーウィンドウの概要

オブジェクトトラッカーウィンドウを使用して、リアルタイムでマルチオブジェクトトラッキング(MOT)を実行します。*マルチオブジェクトトラッキング*は、受信オブザベーションに基づいたモデルを使用して、時間の経過とともに複数のオブジェクトの同一性と位置を正確に推定するプロセスです。これらのオブザベーションは、オブジェクト検出モデルの出力です。

マルチオブジェクトトラッキングの課題には、シーンの雑音、ターゲットのダイナミクス、クラス内およびクラス間の変動、測定ノイズ、フレームレートなどがあります。オブジェクトトラッカーウィンドウは、tracking-by-detection と呼ばれるパラダイムでトラッカーとディテクターを 結合することで、これらの課題に対処します。オブジェクトトラッカーウィンドウでは、トラッキング検出の 2 つの手法である IOU(IOU) 法と ByteTrack メソッドのどちらかを選択することができます。

注: 多くの場合、IOU 法よりも効率的であるため、ByteTrack 法の使用をお勧めします。

オブジェクトトラッカーウィンドウに関連付けられている XML 要素については、window-object-tracker および"Object-Tracker ウィンドウに関連する XML 言語要素" (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

ONNX 例を使用した Computer Vision と ONNX 例を使用した Pose Estimation は、Object Tracker ウィンドウを使用する完全なプロジェクトです。詳細については、"Install Example Projects" (*SAS Event Stream Processing: Using SAS Event Stream Processing Studio*)を参照してください。

検出による追跡の交差オーバーユニオン法の概要

交差オーバーユニオン(IOU)法は、検出データのみを使用してオブジェクトを追跡します。基本的に、IOU は 2 つの境界ボックス間の重なりの程度を定量化する数値です。IOU 法を使用すると、増分およびイベント指向のトラッキングが可能になります。マルチターゲットトラッキングのフレームワークを提供する MOTChallenge や、現実世界の交通現場からキャプチャした挑戦的なビデオシーケンスで構成される UA-DETRAC ベンチマークで優れた結果を生み出しています。IOU 法の詳細については、Bocinski, Eiselein, and Sikora (2017)を参照してください。

オブジェクトトラッカーウィンドウで実装されている IOU 法は、次の機能を実現します。

- オブジェクトの位置の変化率を表す速度ベクトルを使用して、次のフレームでトラッキング対象のオブジェクトが失われるたびに、その次の位置を予測します。tracker 要素の velocity-vector-frames のパラメーターは、速度ベクトルの算出に使用するフレーム数を定義します。
- 既存のトラックとの最初の暫定的な一致と検出が行われた後(IOU>σIOU)、関連付けられていない状態の検出は次のように照合されます。不一致のトラックがある IOU > σIOU2 を持つ検出は、IOU がより大きなどの不一致トラックとも一致します。
- 残りの不一致検出は、既存のトラックの IOU が tracker 要素の iou-sigma-dup パラメーターの値より小さい場合にのみトラックを作成します。この機能により、二重検出の再割り当てが回避されます。iou-sigma-dup パラメーターをデフォルト値 **0.0** に設定すると、この機能は無効になります。
- トラックは複数のライフから始まります。トラックがフレーム上の検出と一致しないままになると、寿命が失われます。検出と一致しないまま数フレームが経過すると、トラックは消滅します。tracker 要素の max-track-lives パラメーターは、検出されないトラックの寿命を設定します。

以下は、以前に参照したパラメータを強調表示するオブジェクトトラッカーウィンドウのサンプル XML コードです:

```
<window-object-tracker pubsub="true" name="tracking">
    <tracker method="iou" score-sigma-low="0.3" score-sigma-high="0.7" iou-sigma="0.6"
iou-sigma2="0.3" iou-sigma-dup="0" velocity-vector-frames="11" max-track-lives="60" min-
track-length="0" track-retention="0"/>
    <output mode="wide" tracks="65" velocity-vector="true"/>
    <input count="objCount" score="obj_%_score" label="obj_%_label" coord-type="rect"
x="obj_%_x" y="obj_%_y" width="obj_%_w" height="obj_%_h"/>
</window-object-tracker>
```

Tracking-by-Detection の ByteTrack 法の概要

オブジェクトトラッカーウィンドウの ByteTrack 法は、一般的な Tracking-by-Detection アルゴリズムである ByteTrack アルゴリズムの実装です。ByteTrack の詳細については、Zhang et al(2022)を参照してください。

オブジェクトトラッカーウィンドウの IOU 法とオブジェクトトラッカーウィンドウの ByteTrack 法は、両方とも IOU 計算を実行します。ByteTrack アルゴリズムのオブジェクトトラッカーウィンドウの実装では、最初の関連付けステップと 2 番目の関連付けステップの両方に IOU 計算を使用します。

XML コードによる例を示します。

```
<window-object-tracker pubsub="true" name="tracking">
        <tracker method="bytetrack" track-thresh="0.5" high-thresh="0.6" match-
thresh="0.8" max-track-lives="60" velocity-vector-frames="30" />
        <output mode="array" velocity-vector="true"  />
        <input group-by="cam_id"  coord-type="rect" count="objCount" x="obj_x" y="obj_y"
width="obj_w" height="obj_h" score="obj_score" label="obj_label"/>
    </window-object-tracker>
```

例のいくつかのパラメーターに関する情報は、次のとおりです。

- 速度ベクトルは、オブジェクトの位置の変化率を表します。ベクトルは、トラッキング対象のオブジェクトが次のフレームで見つからない場合、そのオブジェクトの次の位置を予測します。tracker 要素の velocity-vector-frames パラメーターは、速度ベクトルの算出に使用するフレーム数を定義します。
- tracker 要素の track-thresh パラメーターは、トラックが低スコアトラックであるか、または高スコアトラックであるかを定義する検出スコアのしきい値を指定します。この例に示されている 0.5 の値は、このパラメーターのデフォルト値です。
- tracker 要素の high-thresh パラメーターは、新しい検出に対して新しいトラックが作成されるしきい値を指定します。この例に示されている 0.6 の値は、このパラメーターのデフォルト値です。
- tracker 要素の match-thresh パラメーターは、ハイスコアトラックの類似度割り当てのマッチングしきい値を指定します。このパラメーターの値は IOU 距離であるため、値が大きいほど制限は少なくなります。この例に示されている 0.8 の値は、このパラメーターのデフォルト値です。
- トラックは複数の lives から始まります。トラックがフレーム上の検出と一致しないままになると、寿命が失われます。検出と一致しないまま数フレームが経過すると、トラックは消滅します。tracker 要素の max-track-lives パラメーターは、検出されないトラックの寿命を設定します。
- input 要素の group-by パラメーターは、cam_id というフィールドを使用して、受信イベントをグループ化することを指定します。group-by パラメーターはオプションで、単一のオブジェクトトラッカーウィンドウが複数のカメラからイベントを受信する場合に便利です。オブジェクトトラッカーウィンドウでは、カメラ ID ごとにトラッカーがインスタンス化されます。

オブジェクトトラッカーウィンドウの入力スキーマ

オブジェクトトラッカーウィンドウはマルチオブジェクトオブジェクトトラッキングを実行するために検出データを使用します。検出データは、オブジェクト検出モデル(YOLO など)を視覚画像のストリームに適用することで得られます。

受信した検出データを処理するソースウィンドウは、次のフィールドで構成される入力スキーマを使用します。

- 入力画像のフレーム ID。
- (オプション)。画像の blob 表現。
- 検出対象オブジェクトはイベントにとって重要なものです。
- 検出されたオブジェクトごとに、入力スキーマは次のフィールドで構成されます。
  - 座標のオリジンに対応する、境界ボックスの中心または下隅の x 座標。オブジェクト検出出力では、この座標はしばしば左上隅を指します。
  - 座標のオリジンに対応する、境界ボックスの中心または下隅の y 座標。オブジェクト検出出力では、この座標はしばしば左上隅を指します。

- 座標のオリジンに対応する、境界ボックスの幅または高い方の境界ボックス隅の x 座標。オブジェクト検出出力では、この座標はしばしば右下隅を指します。
- 座標のオリジンに対応する、境界ボックスの幅または高い方の境界ボックス隅の x 座標。オブジェクト検出出力では、この座標はしばしば右下隅を指します。
- (オプション)。検出スコア。デフォルト値は **1** です。
- 検出対象オブジェクトのラベル。
- オブジェクトの属性。
- キーポイントを指定する場合、配列 n*k*(x)を使用します。ここで n は検出されたオブジェクトの数、k は位置にあるキーポイントの数であり、x は x 座標です。
- キーポイントを指定する場合、配列 n*k*(y)を使用します。ここで n は検出されたオブジェクトの数、k は位置にあるキーポイントの数であり、y y 座標です。
- キーポイントを指定する場合、配列 n*k*(s)を使用します。ここで n は検出されたオブジェクトの数、k は位置にあるキーポイントの数であり、s 指定されたスコアです。
- キーポイントを指定する場合、配列 n*k*(label_id)を使用します。ここで n は検出されたオブジェクトの数、k は位置にあるキーポイントの数であり、label_id 指定されたラベルです。

- (オプション)。タイムスタンプ。
- イベントキーフィールドを含め、他のすべてのフィールドは出力イベントに伝播されます。

*キーポイント*は、フレーム間を移動するオブジェクトを追跡するために使用される、画像内の特定の検出可能な特徴を識別します。通常、キーポイントは、さまざまな視点から見て一貫して識別できる明確なポイント、コーナー、またはエッジです。キーポイントが指定されると、それを使用して、周囲に対するオブジェクトの位置と方向を推定します。たとえば、カテゴリ レベルのオブジェクトポーズ追跡システムでは、キーポイントはオブジェクトを囲む境界ボックスの頂点を表すことができます。

次の入力スキーマは配列入力を処理します。

```
frame_id*:int64,image:blob,objCount:int32,obj_label:string,obj_score:array(dbl),
obj_x:array(dbl),obj_y:array(dbl),obj_w:array(dbl),obj_h:array(dbl),
obj_kpts_count:array(i32),obj_kpts_x:array(dbl),obj_kpts_y:array(dbl),obj_kpts_score:array(dbl),obj_kpts_label_id:array(i32),
time:stamp
```

次のオブジェクトトラッカーウィンドウは、その入力スキーマを使用するソースウィンドウからの入力を受け入れます。

```
<window-object-tracker pubsub="true" name="tracker">
        <tracker method="iou" score-sigma-low="0.08" score-sigma-high="0.08" iou-sigma="0.15" iou-sigma2="0.1" iou-sigma-dup="0.1" velocity-vector-frames="5" max-track-lives="40" track-retention="10"/>
        <output mode="array" tracks="30" velocity-vector="true" keypoints="true" scale-x="220" scale-y="640"/>
```

```
        <input count="objCount" score="obj_score" label="obj_label" coord-type="rect"
x="obj_x" y="obj_y" width="obj_w" height="obj_h"
                kpts-count="obj_kpts_count" kpts-label-id="obj_kpts_label_id" kpts-
score="obj_kpts_score" kpts-x="obj_kpts_x" kpts-y="obj_kpts_y"/>
</window-object-tracker>
```

次の入力スキーマは、検出された 3 つのオブジェクトを処理します。

```
frame_id*:int64,image:blob,objCount:int32,obj_0_x:double,obj_0_y:double,obj_0_h:double,
obj_0_w:double,obj_0_score:double,obj_0_label:string,obj_1_x:double,obj_1_y:double,obj_1_
h:double,
obj_1_w:double,obj_1_score:double,obj_1_label:string,obj_2_x:double,obj_2_y:double,obj_2_
h:double,
obj_2_w:double,obj_2_score:double,obj_2_label:string,obj_3_x:double,obj_3_y:double,obj_3_
h:double,
obj_3_w:double,obj_3_score:double,obj_3_label:string,time:stamp
```

オブジェクトトラッカーウィンドウの入力要素を使用して、オブジェクトの入力フィールドの命名規則を定義します。配列以外の入力を処理する場合は、オブジェクト番号のプレースホルダーとして**%**文字を使用します。

たとえば、<input>要素の'x'属性の値が**'obj _%_ x'**の場合、ウィンドウはオブジェクトの x 座標が obj_0_x, obj_1_x, obj_2_x, ...と想定します。

```
<window-object-tracker pubsub="true" name="tracking">
    <tracker method="iou" score-sigma-low="0.3" score-sigma-high="0.7" iou-sigma="0.6"
iou-sigma2="0.3" iou-sigma-dup="0" velocity-vector-frames="11" max-track-lives="60" min-
track-length="0" track-retention="0"/>
    <output mode="wide" tracks="65" velocity-vector="true"/>
    <input count="objCount" score="obj_%_score" label="obj_%_label" coord-type="rect"
x="obj_%_x" y="obj_%_y" width="obj_%_w" height="obj_%_h"/>
</window-object-tracker>
```

coord-type=rect は、幅と高さの値とともに左上隅の x 座標と y 座標を使用して、境界ボックスを指定します。coord-type=yolo は、中心の x および y 座標と幅および高さの値を使用して、境界ボックスを指定します。coord-type=coco は、左上隅の x-min および y-min 座標と、右下隅の x-max および y-max 座標を使用して、境界ボックスを指定します。この形式は、COCO(コンテキスト内の共通オブジェクト)として知られるデータセットから名前を取ります。

<input>要素が使用されていない場合は、標準命名規則の出力フィールド分析ストアが使用されます。

オブジェクトトラッカーウィンドウの出力スキーマ

概要

オブジェクトトラッカーウィンドウの出力は、出力要素の mode 属性の値に応じて、幅スキーマ出力は長さスキーマに準拠します。

すべて出力スキーマには、入力ウィンドウのオブジェクト以外のすべてのフィールドが含まれます。入力ウィンドウのキー フィールドは、出力スキーマの一部ではありません。

幅スキーマ

幅スキーマには、次のフィールドが含まれています。

*prefix*_density type="int32" 1
*prefixN*_id type="int64" 2 3
*prefixN*_label type="string"
*prefixN*_score type="double"
*prefixN*_x type="double"
*prefixN*_y type="double"
*prefixN*_w type="double"
*prefixN*_h type="double"
*prefixN*_center_x type="double" 4
*prefixN*_center_y type="double"
*prefixN*_track_x type="array(dbl)" 5
*prefixN*_track_y type="array(dbl)"
*prefixN*_track_count type="int32"
*prefixN*_track_attributes type="string"

1 <output>要素の prefix 属性を使用して、*prefix* の値を定義します。prefix のデフォルト値は**'Object'**です。

2 *N* 値は、オブジェクト番号のゼから始まるインデックスです。<output>要素の tracks 属性を使用して、出力オブジェクトの数を定義します。

3 <..>_id フィールドはオブジェクトのトラックの識別子を含みます。

4 <..>_center_x フィールドと<..>_center_y フィールドは、オブジェクトの現在位置の境界ボックスの中心の x 座標と y 座標を含みます。

5 <..>_track_x および<..>_track_y フィールドは、境界ボックスの中心の x 座標と y 座標の履歴をフレームごとに含む配列フィールドです。この情報は、オブジェクトの完全なトラックを提供します。つまり、これらの点を結ぶ線がオブジェクトの動きをトレースします。詳細については、"配列スキーマ"を参照してください。

velocity-vector="true"の場合、次の追加フィールドがスキーマに表示されます。

*prefixN*_vvect_x type="double" 1
*prefixN*_vvect_y type="double"

1 <..>_vvect_x フィールドおよび<..>_vvect_y フィールドは、オブジェクトの最後の速度ベクトルの座標が含まれます。

たとえば、prefix='Object'、tracks=4、および velocity-vector="true"と入力スキーマは、"オブジェクトトラッカーウィンドウの入力スキーマ"で指定されたものであり、出力スキーマは次のようにようになります:

frame_id*:int64,image:blob,objCount:int32,time:stamp,Object_density:int32,Object0_id:int64,Object0_label:string,
Object0_score:double,Object0_x:double,Object0_y:double,Object0_w:double,Object0_h:double,
Object0_center_x:double,Object0_center_y:double,Object0_track_x:array(dbl),Object0_track_y:array(dbl),
Object0_vvect_x:double,Object0_vvect_y:double,Object1_id:int64,Object1_label:string,Object1_score:double,
Object1_x:double,Object1_y:double,Object1_w:double,Object1_h:double,Object1_center_x:double,
Object1_center_y:double,Object1_track_x:array(dbl),Object1_track_y:array(dbl),Object1_vvect_x:double,
Object1_vvect_y:double,Object2_id:int64,Object2_label:string,Object2_score:double,Object2_x:double,
Object2_y:double,Object2_w:double,Object2_h:double,Object2_center_x:double,Object2_center_y:double,
Object2_track_x:array(dbl),Object2_track_y:array(dbl),Object2_vvect_x:double,Object2_vvect_y:double,

Object3_id:int64,Object3_label:string,Object3_score:double,Object3_x:double,Object3_y:double,
Object3_w:double,Object3_h:double,Object3_center_x:double,Object3_center_y:double,
Object3_track_x:array(dbl),Object3_track_y:array(dbl),Object3_vvect_x:double,Object3_vvect_y:double

keypoints="true"の場合、次の追加フィールドがスキーマに表示されます。

*prefixN*_track_kpts_count type="array(i32)" 1
*prefixN*_track_kpts_x type="array(dbl)" 2
*prefixN*_track_kpts_y type="array(dbl)" 3
*prefixN*_track_kpts_score type="array(dbl)" 4
*prefixN*_track_kpts_label_id type="array(i32)" 5

1 n*t*(k)配列は、各トラック位置のキーポイントの数を指定します。

2 n*t*k*(x)配列は、X 座標の各トラック位置のキーポイントを指定します。

3 n*t*k*(y)配列は、X 座標の各トラック位置のキーポイントを指定します。

4 n*t*k*(s)配列は、各トラック位置のキーポイントのスコアを指定します。

5 n*t*k*(label_id)配列は、各トラック位置のキーポイントの数を指定します。

長いスキーマ

長いスキーマには、次のフィールドが含まれています。

*prefix*_density 1 type="int32"
*prefix*_id type="int64"
*prefix*_label type="string"
*prefix*_score type="double"
*prefix*_x type="double"
*prefix*_y type="double"
*prefix*_w type="double"
*prefix*_h type="double"
*prefix*_center_x type="double" 2
*prefix*_center_y type="double"
*prefix*_track_x type="array(dbl)" 3
*prefix*_track_y type="array(dbl)"
*prefix*_track_count type="int32"
*prefix*_track_attributes type="string"

1 <output>要素の prefix 属性を使用して、*prefix* の値を定義します。prefix のデフォルト値は**'Object'**です。

2 <..>_center_x フィールドと<..>_center_y フィールドは、オブジェクトの現在位置の境界ボックスの中心の x 座標と y 座標を含みます。

3 <..>_track_x および<..>_track_y フィールドは、境界ボックスの中心の x 座標と y 座標の履歴をフレームごとに含む配列フィールドです。この情報は、オブジェクトの完全なトラックを提供します。つまり、これらの点を結ぶ線がオブジェクトの動きをトレースします。詳細については、"配列スキーマ"を参照してください。

velocity-vector="true"の場合、スキーマには次の追加フィールドが含まれます。

*prefix*_vvect_x type="double"
*prefix*_vvect_y type="double"

たとえば、prefix='Object'、velocity-vector="true"、および入力スキーマが"オブジェクトトラッカーウィンドウの入力スキーマ"で指定されているものであるとします。出力スキーマは次のようになります。

```
frame_id*:int64,image:blob,objCount:int32,time:stamp,Object_density:int32,Object_id*:int64,Object_label:string,
Object_score:double,Object_x:double,Object_y:double,Object_w:double,Object_h:double,
Object_center_x:double,Object_center_y:double,Object_track_x:array(dbl),Object_track_y:array(dbl),
Object_vvect_x:double,Object_vvect_y:double
```

keypoints="true"の場合、次の追加フィールドが出力スキーマに表示されます。

```
prefix_track_kpts_count    type="array(i32)" 1
prefix_track_kpts_x    type="array(dbl)" 2
prefix_track_kpts_y    type="array(dbl)" 3
prefix_track_kpts_score    type="array(dbl)" 4
prefix_track_kpts_label_id    type="array(dbl)" 5
```

1 t*(k)配列は、各トラック位置のキーポイントの数を指定します。

2 t*k*(x)配列は、X 座標の各トラック位置のキーポイントを指定します。

3 t*k*(y)配列は、X 座標の各トラック位置のキーポイントを指定します。

4 t*k*(s)配列は、各トラック位置のキーポイントのスコアを指定します。

5 t*k*(label_id)配列は、各トラック位置のキーポイントのラベル ID を指定します。

配列スキーマ

n が追跡されるオブジェクトの数に等しく、t がトラック内のポジションの数と同じです。配列スキーマには、次のフィールドが含まれています。

```
prefix_density    type="int32" 1
prefix_id         type="array(i64)" 2
prefix_label      type="string" 3
prefix_score      type="array(dbl)" 4
prefix_x          type="array(dbl)" 5
prefix_y          type="array(dbl)" 6
prefix_w          type="array(dbl)" 7
prefix_h          type="array(dbl)" 8
prefix_center_x    type="array(dbl)" 9
prefix_center_y    type="array(dbl)" 10
prefix_track_x     type="array(dbl)" 11
prefix_track_y     type="array(dbl)" 12
prefix_track_count type="array(i32)" 13
prefix_attributes  type="string" 14
```

1 n は、オブジェクトの数を指定します。

2 n*(id)の配列。

3 n*(label)は、オブジェクトごとに 1 つのラベルの CSV 文字列です。

4 n*(score)の配列。

5 n*(x)の配列。

6 n*(y)の配列。

7 n*(w)の配列。

8 n*(h)の配列。

9 n*(c_x)の配列。

10 n*(c_y)の配列。

11 n*t*(x)の配列。

12 n*t*(y)の配列。

13 n*(t)は、各オブジェクトのトラック内の位置の数です。

14 n*(attrList)は、区切り文字として','を使用した文字列または rstring としてのCSV(ラベル区切り属性)です。

velocity-vector="true"の場合、出力スキーマには次の追加フィールドが含まれます。

*prefix*_vvect_x type="double"
*prefix*_vvect_y type="double

keypoints="true"の場合、次の追加フィールドが出力スキーマに表示されます。

*prefix*_track_kpts_count type="array(i32)" 1
*prefix*_track_kpts_x type="array(dbl)" 2
*prefix*_track_kpts_y type="array(dbl)" 3
*prefix*_track_kpts_score type="array(dbl)" 4
*prefix*_track_kpts_label_id type="array(i32)" 5

1 t*(k)配列は、各トラック位置のキーポイントの数を指定します。

2 n*t*k*(x)配列は、X 座標の各トラック位置のキーポイントを指定します。

3 n*t*k*(y)配列は、X 座標の各トラック位置のキーポイントを指定します。

4 t*k*(s)配列は、各トラック位置のキーポイントのスコアを指定します。

5 t*k*(label_id)配列は、各トラック位置のキーポイントの数を含みます。

Pose Estimation with ONNX の例では、w_object_tracker_array ウィンドウで配列スキーマを使用します。このウィンドウの定義には、keypoints="true"および velocity-vector="true"が含まれます。そのウィンドウの配列スキーマ出力は次のとおりです:

id:int64,_nObjects_:int64,image:blob,Object_density:int64,Object_id:array(i64),Object_label(string),
Object_score:array(i64),Object_x:array(i64),Object_y:array(i64),Object_w:array(i64),Object_h:array(i64),
Object_center_x:array(i64),Object_center_y:array(i64),Object_track_x:array(i64),Object_track_y:array(i64),
Object_track_count:array(i64),Object_attibutes:string,Object_vvect_x:array(i64),Object_vvect_y:array(i64),
Object_track_kpts_count:array(i64),Object_track_kpts_x:array(i64),Object_track_kpts_y:array(i64),
Object_track_kpts_score:array(i64),Object_track_kpts_label_id:array(i64)

ONNX 例を使用した Pose Estimation のイベントの出力の一部を次に示します。1つのオブジェクトの x 座標と y 座標が 10 回トラッキングされています。

Object_track_x: [1,057;1,056;1,056;1,056;1,056;1,056;1,056;1,057;1,057;1,057]
Object_track_y: [329;329;329;329;330;330;330;330;331;331]
Object_track_count: [10]

ONNX 例を使用した Pose Estimation の別のイベントを次に示します。次の 2 つのオブジェクトがあります。このイベントには、最初のオブジェクトの位置が 10 個、2 番目のオブジェクトの位置が 10 個含まれます。

Object_track_x: [1,057;1,057;1,058;1,058;1,058;1,057;1,057;1,057;1,057;1,055;39;55;68;69;80;84;85;87;86;86]
Object_track_y: [331;332;332;332;332;333;333;333;332;332;399;399;402;399;389;394;379;360;359;350]
Object_track_count: [10;10]

Object_track_x では、最初の 10 個の位置 (1,057;1,057;1,058;1,058;1,058;1,057;1,057;1,057;1,057;1,055)が最初のオブジェクトに関連します。位置は、最も古いものから新しいものの順に並べられます。1,057(最初の数字)は最も古い位置で、1,055 が最新です。他の 10 個の位置は、2 番目のオブジェクトに関連します。

オブジェクトトラッカーウィンドウの履歴を管理する

track-retention パラメーターを使用トラックがメモリ内の位置の履歴に保持するフレームの数を指定します。デフォルトでは、このパラメーターの値は **0** です。同じオブジェクトが数時間または数日経過しても画像に存在する場合、このパラメーターを **0** の値のままにすると、トラック履歴が無期限に増加する可能性があります。この増加はプロジェクトをクラッシュさせる可能性があります。

以降の追跡を新しい状態で続行できるように、どのような条件でオブジェクトトラッカーウィンドウの履歴をクリアするかを指定できます。

frame-skip-field パラメータを使用して、履歴をクリアするために使用するフレームインデックスまたは時間を含む入力イベントのフィールドの名前を指定します。frame-skip-amount パラメーターを使用して、履歴がクリアされてオブジェクト追跡が再開されるまでに、見逃されるフレーム数または経過時間を秒単位で指定します。

たとえば、次のフィールドと内容を含む入力イベントのストリームがあるとします。

表 4.6　サンプル入力イベントのフィールドと内容

フレーム番号	画像
1	**image1**
2	**image2**
3	**image3**
10	**image10**

"frame-skip-field"="frame_number"に設定したと仮定します。さらに、frame_number フィールドにはフレームインデックスが含まれており、"frame-skip-amount"=5 であると仮定します。

"frame_number"=3 および"frame_number"=10 のギャップが、指定されたフレームスキップ量 **5** より大きい場合、前の画像(**image1**、 **image2** 、および **image3**)で追跡されていたオブジェクトはクリアされます。オブジェクト追跡は **image10** から再開されます。

ML 構文関連については、*“Object-Tracker ウィンドウに関連する XML 言語要素” (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*を参照してください。

参照

Bockinski, Erik, Volker Eiselein, and Thomas Sikora.2017.“High-Speed Tracking-by-Detection Without Using Image Information.”In *IEEE International Conference on Advanced Video and Signal-based Surveillance*.Lecce, Italy.IEEE Computer Society.http://elvera.nue.tu-berlin.de/files/1517Bochinski2017.pdf

Zhang, Yifu, Peize Sun, Yi Jiang, Dongdong Yu, Fucheng Weng, Zehuan Yuan, Ping Luo, Wenyu Liu, Xinggang Wang.2022."ByteTrack: Multi-Object Tracking by Associating Every Detection Box."https://arxiv.org/pdf/2110.06864

StateDB ウィンドウの使用

概要

SAS Event Stream Processing プロジェクトでは、結合操作、集計、ローリング集計を非常に高いスループットと非常に低い待機時間で実行できます。通常、これらのタスクを実行するために必要なデータは RAM に格納されます。このデータは、それらのタスクを実行するウィンドウの"状態"を表します。

この"状態"データの量が、システム機能に負担をかける場合があります。たとえば、結合が大量のデータに対して実行される場合、または集計期間が数日または数か月にわたる場合、処理パフォーマンスが低下する可能性があります。例:

- 必要な RAM の量は、システム能力を超える場合があります。
- データをメモリにロードする時間は、許容できないほど遅くなる可能性があります。
- 通常、大量のデータはフェイルオーバーで再処理されます。再処理を行うと、システムの障害からの回復速度が耐えられないほど遅くなる可能性があります。
- ステートフルモデルでは、各ノードに必要な状態データをロードする必要があるため、オートスケーリングが遅くなる可能性があります。この遅いオートスケーリングにより、データが不要に重複し、メモリ内のフットプリントが増加する可能性があります。

StateDB ライターウィンドウと StateDB リーダーウィンドウを使用すると、外部に配置された高性能データストアまたはデータベースを使用して、結合と集計に必要な"ステート(状態)"を保存および維持できます。外部ストアを使用すると、RAM を使用する必要性が減少します。次にいくつかの利点を示します。

- これらの外部ストアは、SAS Event Stream Processing よりも効率的に大量のデータを長期間にわたって管理できます。
- データが外部に保存されているため、プロジェクトは可能な限りステートレスです。
- ポッドとプロジェクト間でデータを共有できるため、スケーラビリティが向上します。
- プロジェクトが開始されるたびにデータをプリロードする必要はありません。
- フェールオーバーの場合、データはすぐに利用でき、失われることはないため、データをリロードする必要はありません。

> 重要　StateDB ウィンドウを使用するには、Redis をダウンロードしてインストールする、SingleStore をインストールする必要があります。詳細については、Redis ウェブサイトまたは SingleStore ウェブサイトを参照してください。

> 重要　StateDB Reader ウィンドウが Singlestore テーブルから読み取る場合、そのテーブルには次の 2 つのフィールドが含まれている必要があります。
>
> - esp_meta_timestamp
> - esp_meta_ttd
>
> これらのフィールドは、SAS Event Stream Processing によって、Singlestore テーブルからのフィールドの削除を管理するために使用されます。これらは、StateDB Writer ウィンドウによってテーブルが作成されるときに自動的に作成されます。別の方法で作成されたテーブルを使用する場合は、これらの 2 つのフィールドを BIGINT 型として手動で追加する必要があります。

StateDB ウィンドウに関連付けられている XML エレメントについては、“StateDB ウィンドウに関連する XML 言語要素” (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

StateDB ライターウィンドウの使用

StateDB ライターウィンドウを使用して、次のアクションを完了します。

- 着信イベントを定義済みの外部データベースに書き込む
- イベントに必要なインデックスを作成する
- 入力イベントを変更せずにプロジェクトの次のウィンドウに渡す

ポッド内では、プロジェクト内で操作を適切に順序付けするために、同期的に書き込まれます。指定されたキーフィールドは、データベース内のプライマリインデックスとして使用されます。入力マッピングから選択された他のすべてのフィールドは、指定された名前を使用して書き込まれます。

StateDB ライターウィンドウは、データ(イベント)をキーと値のペアとして格納します。パフォーマンスを低下させずに非キーフィールドのクエリを実行できるようにするために、StateDB ライターを使用すると、これらのフィールドにセカンダリインデックスを作成できます。これらのセカンダリインデックスは、1 対多の結合または集計を実行するために必須です。

StateDB リーダーウィンドウの使用

StateDB リーダーウィンドウを使用して、受信イベントと指定された条件に基づいて、定義された外部データベースからデータをクエリまたはフェッチします。これらの条件の一部は、StateDB ライターウィンドウで作成したインデックスに基づいています。

データベースからゼロ、1 つ、または複数のレコードを取得できます。これらのレコードを集約するようにウィンドウプロパティを構成できます。ウィンドウは、出力として 1 つまたは複数のイベントを生成します。

注: StateDB リーダーウィンドウは、StateDB リーダークエリが、セカンダリインデックスまたはそれらのフィールドに存在するプライマリインデックスを持たないフィールドを指定するたびに、エラーを生成します。

StateDB リーダーウィンドウは、次の機能をサポートしています。

表 4.7 サポートされる機能

機能	説明
ルックアップ操作	この機能は、通常、結合ウィンドウによって実行される 1 対 1 の結合に似ています。 StateDB リーダーウィンドウの受信イベントごとに、プライマリインデックスに基づいてデータベース内に一致するレコードが 0 個または 1 個あります。 一致するレコードがない場合、StateDB リーダーウィンドウは null フィールドを書き込みます。一致するレコードがある場合、StateDB リーダーウィンドウはフィールド自体の値を書き込みます。
複数のイベントの取得	この機能は、通常、結合ウィンドウによって実行される 1 対多の結合に似ています。 データベースから取得する追加のキー(source=query)を指定する必要があり、取得されたイベントのグループに対して一意である必要があります。1 つの着信イベントごとの発信イベントごとに一意のキーを作成するために、データベースから取得されたフィールドから 1 つ以上のキーを追加することをお勧めします。

機能	説明
	StateDB リーダーウィンドウの受信イベントごとに、セカンダリインデックスに基づいてデータベース内に一致するレコードが 0 個以上あります。 一致するレコードがない場合、StateDB リーダーウィンドウは空白のフィールドを書き込みます。一致するレコードがある場合、StateDB リーダーウィンドウは、データベースからフェッチされたイベントのフィールドを持つ複数のイベントを含むトランザクションブロックを書き込みます。
イベントの集計	この機能は、集計ウィンドウで実行される集計に似ています。 着信イベントごとに、1 つの出力イベントがあります。出力イベントには、集計関数によって定義されたフィールドが含まれています。 StateDB リーダーウィンドウでは、次の集計関数を使用できます。 ■ ESP_aAve(*value*) ■ ESP_aCount() ■ ESP_aFirst(*value*) ■ ESP_aLastNonNull(*value*) ■ ESP_aMax(*value*) ■ ESP_aMin(*value*) ■ ESP_aLag(*value, lag*) ■ ESP_aLast(*value*) ■ ESP_aCat(*value, stringConstant*) ■ ESP_aSum(*value*) これらの関数は、SAS Event Stream Processing Studio の StateDB リーダーウィンドウの出力スキーマで構成します。 集計関数の詳細については、“集計ウィンドウフィールド計算式の集計関数”を参照してください。

Redis が外部データストアである場合の StateDB ウィンドウでのトランスポート層セキュリティ(TLS)の使用

トランスポート層セキュリティ(TLS)を使用して、Redis データストアと SAS Event Stream Processing の間の接続を保護できます。デフォルトでは、接続は安全ではありません。

SAS Event Stream Processing には、Redis サーバーの認証に使用される OpenSSL ソフトウェアが付属しています。OpenSSL ソフトウェアは、特定のロジックを使用して証明書およびその他の関連成果物を検索します。SAS Event Stream Processing は、認証の目的でこれらの証明書と成果物を使用します。

クライアント側のキーと証明書は、相互認証が必要な場合にのみ必要です。

接続を保護するには:

1 プロジェクト XML コードの statedb 要素で、use-ssl="true"を指定します。または、SAS Event Stream Processing Studio でパラメーターを設定します。

2 必要に応じて、StateDB ウィンドウの次のプロパティの値を設定します。

属性	説明
[ca-cert-filename=*path*]	ロードして検証に使用する CA 証明書またはバンドルファイルの名前を指定します。バンドルファイルは、既知の CA 証明書のセットを指定します。
[ca-path=*path*]	信頼された CA 証明書ファイルが OpenSSL 互換の構造で保存されるディレクトリを指定します。 **注:** パスを指定すると、TLS はそこにある認証の成果物をすべて使用します。CA 証明書またはバンドルファイルの名前を指定する必要はありません。
[cert-filename=*name*]	クライアント側の証明書の名前を指定します。指定すると、private-key-filename も指定する必要があります。
[private-key-filename=*name*]	認証に使用する秘密キーファイルの名前を指定します。指定すると、cert-filename も指定する必要があります。
[server-name=*name*]	Server Name Indication(SNI)は、クライアントが接続しようとしているホスト名を示すことができるようにする Transport Layer Security(TLS)プロトコルの拡張機能です。この属性は、SNI TLS 拡張を指定します。

Redis が外部データストアである場合の StateDB ウィンドウでのセカンダリインデックスの使用

Redis はセカンダリインデックスのアウトオブボックスサポートを提供しないため、StateDB ライターウィンドウを使用して実装する必要があります。このセカンダリインデックスの仕組みを理解するには、Redis がプライマリインデックスを実装する方法を理解する必要があります。

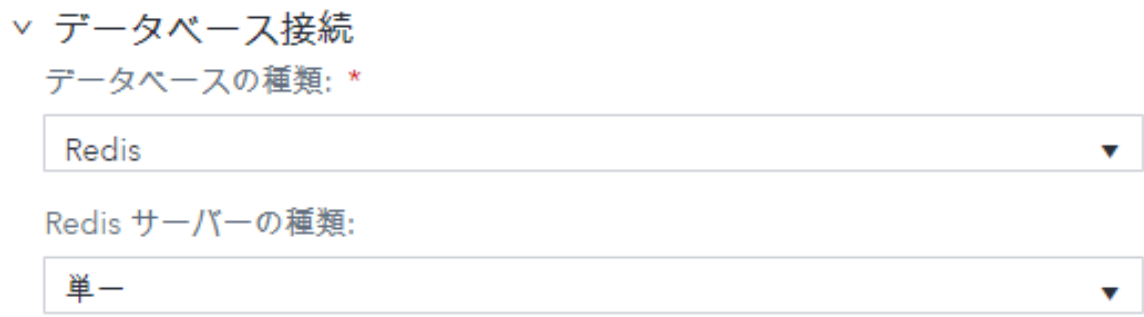
次のイベント スキーマを考えてみましょう。

assetid*:string, tagid*:string, timestamp*:stamp, value:double

assetid、tagid、および timestamp がキーフィールドであることに注意してください。

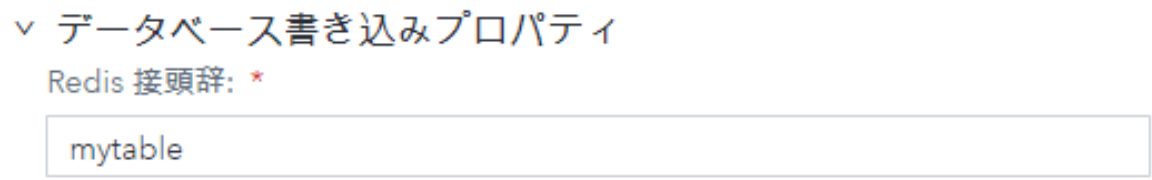
セカンダリインデックスの使用を開始するには、StateDB ライターウィンドウで外部データストアとして Redis の使用を指定します。ここでは、単一のデータストアが使用されます。

図 4.3　データベース接続を Redis として指定する



Redis テーブル接頭辞の値を mytable に設定するとします。

図 4.4　Redis 接頭辞を指定する



ここで、次の 2 つのイベントを考えてみましょう。

asset2, tagId46122, 1617765990190749, 0.461220
asset2, tagId186548, 1617859302543312, 0.461333

Redis は、イベントを Redis ハッシュ構造のキー値レコードとして保存します。プライマリインデックスキーは次のように格納されます。

prefix_name:*key_value*[,*key_value*,...].

したがって、StateDB ライターウィンドウが指定されたイベントを処理するとき、次のようなプライマリインデックスキーがあります。

mytable:asset2:tagId46122:1617765990190749
mytable:asset2:tagId168548:1617859302543312

Redis は、セカンダリインデックスをキー値レコードとして Redis Sorted Set 構造に保存します。このセカンダリインデックスのキーは、次のように格納されます。*prefix_name*:SK:*sec_key_value*[,*sec_key_value*,...]。値は、このグループの関連するプライマリインデックスです。

StateDB ライターウィンドウで、次のように mytable 接頭辞にセカンダリインデックスを定義するとします。

図 4.5　セカンダリインデックスの指定

Redis は、このセカンダリインデックスキーを次のように格納します。

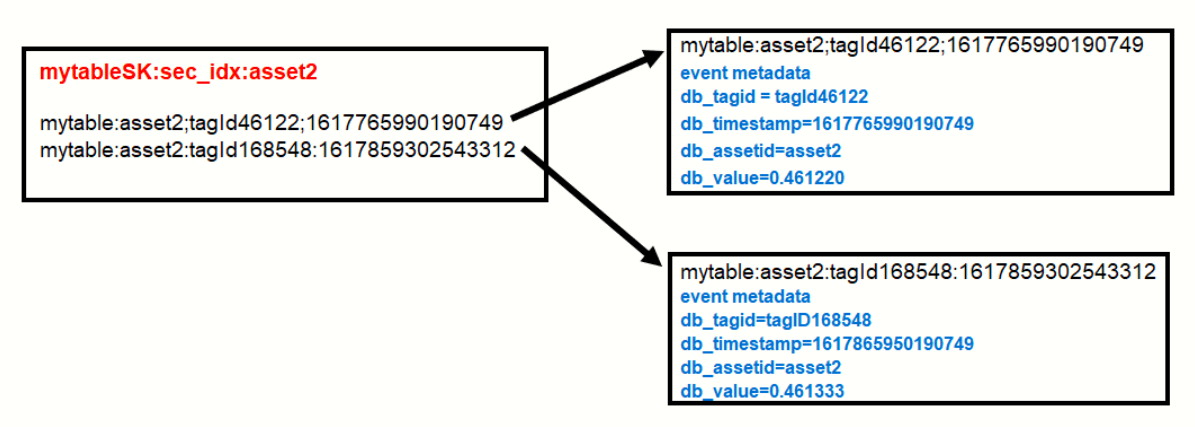
mytable:SK:sec_idx:asset2

StateDB ライターウィンドウの出力を次のように指定するとします。

図 4.6　StateDB ライターウィンドウの出力を指定する

その後、各 Redis エントリは、対応するイベントのデータを格納するキー値ストアを指します。

図 4.7　イベントへのインデックスの対応



イベントを StateDB リーダー ウィンドウに渡すように StateDB ライターウィンドウを設定したとします。

StateDB リーダーウィンドウによって実行されるクエリが、StateDB ライターウィンドウ(**mytable**)に対して定義したものと同じテーブル接頭辞を使用するように指定します。次に、実際のクエリを次のように定義します。

- **入力フィールド**に **assetid** を指定します。この値は、セカンダリインデックス **sec_idx** に指定した値と一致します。
- StateDB ライターウィンドウで**書き込みフィールド**として指定したものと一致する **Redis フィールド**で **db_assetid** を指定します。
- データベースの種類が Redis の場合、使用可能な演算子は==のみです。

図 4.8　データベースクエリの指定

このクエリを使用して、StateDB リーダーウィンドウはセカンダリインデックスのプライマリインデックスのリストを取得します。次に、StateDB リーダーウィンドウは、入力イベントのセカンダリインデックスの **assetid** に一致するすべてのイベントをフェッチします。

受信イベントのセカンダリインデックスフィールドの値は **asset2** であるため、次のアクションが発生します。

1 StateDB リーダーウィンドウは、キーを **test:SK:sec_Idx:asset2** とした Sorted Set を取得します。これにより、ウィンドウに 2 つのキーのリストが表示されます。

mytable:asset2:tagId46122:1617765990190749

mytable:asset2: tagId186548:1617859302543312

2 StateDB リーダーウィンドウは、Redis から前の 2 つのキーのハッシュデータ構造に格納されている対応するイベントを取得し、次の値を返します。

db_assetid=asset2, db_tagid=tagId46122, db_timestamp=1617765990190749, db_value=0.461220

db_assetid=asset2, db_tagid=tagId186548, db_timestamp=1617859302543312, db_value=0.461333

StateDB リーダーウィンドウで、次のように出力スキーマを作成します。

- 指定された最初の 4 つのフィールドに対して、**Input** する**ソース**を指定します。フィールドは、ソースウィンドウによって処理された入力イベントから直接取得されます。
- 残りのフィールドでは、**ソース**を **Query** に指定します。入力値は、Redis データストアから取得した値と照合されます。
- **集計関数**で集計関数を選択して、取得した **ソースフィールド**の値に適用します。結果は、指定された**フィールド名**に書き込まれます。

図 4.9 出力スキーマの作成

出力スキーマ

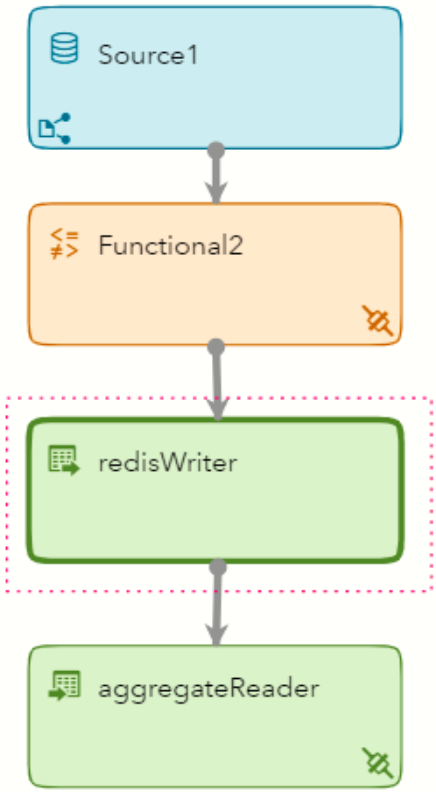
キー	フィールド名	種類	ソース	ソースフィールド	集計関数	パラメータ
☑	tagid	string	入力			
☑	timestamp	int64	入力			
☑	value	double	入力			
☐	sum_value	double	クエリ	db_value	ESP_aSum	(関数にパラメータがありません)
☐	avg_value	double	クエリ	db_value	ESP_aAve	(関数にパラメータがありません)
☐	first_value	double	クエリ	db_value	ESP_aFirst	(関数にパラメータがありません)
☐	last_value	double	クエリ	db_value	ESP_aLast	(関数にパラメータがありません)
☐	min_value	double	クエリ	db_value	ESP_aMin	(関数にパラメータがありません)

1 -

例: StateDB ライターおよび StateDB リーダーウィンドウを介したイベントのフロー

次のプロジェクトを考えてみましょう。これは、1 つの StateDB ライターウィンドウと 1 つの StateDB リーダーウィンドウを使用して、イベントがプロジェクトをどのように通過するかについての簡単なシナリオを実装しています。

図 4.10　サンプル プロジェクト



ソースウィンドウが産業アセットからデータを受け取り、次の入力スキーマが含まれているとします。

```
<schema>
  <fields>
    <field name="id" type="int64" key="true"/>
    <field name="assetid" type="string"/>
    <field name="tagid" type="string"/>
    <field name="timestamp" type="int64"/>
    <field name="value" type="double"/>
  </fields>
</schema>
```

図に示すように、関数ウィンドウはソースウィンドウからイベントを受信し、2 つの新しい変数を生成します: time-to-live(ttl)および timestamp(currentTimestamp)。

```
<window-functional index="pi_EMPTY" name="Functional2" pubsub="true">
        <schema>
         <fields>
          <field name="assetid" type="string" key="true"/>
          <field name="tagid" type="string" key="true"/>
          <field name="timestamp" type="int64" key="true"/>
          <field name="value" type="double"/>
          <field name="ttl" type="int64"/>
          <field name="currentTimestamp" type="int64"/>
         </fields>
        </schema>
        <function-context>
         <functions>
          <function name="currentTimestamp"><![CDATA[systemMicro()]]></function><!-- 1 --
>
          <function name="ttl"><![CDATA[86400]]></function>
         </functions>
        </function-context>
    <connectors>
        <!-- 2 -->...
    </connectors>
</window-functional>
```

1 systemMicro 関数の詳細については、21 章, “関数ウィンドウと通知ウィンドウの機能”を参照してください。

2 ファイルおよびソケットコネクタを使用して、出力を CSV ファイルに書き込んで検証します。(実際のコードは示していません。)

StateDB ライターウィンドウは、関数ウィンドウからイベントを受信します。

これは、Redis を使用した StateDB ライターウィンドウです。

```
<window-statedb-writer name="redisWriter" pubsub="true" >
  <statedb type="redis" hostname="generic.localhost" port="6379" cluster="false"
username="username" password="password" max-reconnect-retries="20" reconnect-retry-
delay="1000"/> <!-- 1 -->
    <write prefix="mytable" ttl="ttl" ttl-offset="60" del-dead-sec-keys="true"> <!-- 2 -->
     <secondary-indexes>
      <secondary-index name="sec_idx"> <!-- 3 -->
       <fields>
        <field name="assetid"/>
       </fields>
      </secondary-index>
     </secondary-indexes>
     <output> <!-- 4 -->
      <field-selection name="assetid"    db-name="db_assetid" />
      <field-selection name="tagid"     db-name="db_tagid" />
      <field-selection name="timestamp"  db-name="db_timestamp" />
      <field-selection name="value"     db-name="db_value" />
     </output>
    </write>
</window-statedb-writer>
```

1 ウィンドウは、ユーザー名 **username** とパスワード **password** を使用して、ポート 6379 で generic.localhost の Redis データベースにアクセスします。プロジェクトを作成する前に、特定のデータベースに関する同様の情報を収集してください。

2 prefix 値の **mytable** は、データのグループ化に使用されます。Redis にはテーブルオブジェクトがないため、この接頭辞がテーブル名として機能します。

3 セカンダリインデックス sec_idx が Redis データベース上に作成されます。StateDB ライターウィンドウがセカンダリインデックスを作成すると、インデックスはキーと値のペアの構造を構築します。セカンダリインデックスフィールドの値がキーとして機能し、実際のイベントのプライマリキーのリストが値として機能します。ここには、assetid と呼ばれる単一のセカンダリキーがありますが、必要に応じて複数のセカンダリキーを作成できます。

4 選択したイベントのフィールドは、db-name 属性値で指定された名前でデータベースに書き込まれます。例えば、assetid は、db_assetid と書かれます。もし db-name 属性が指定されていない場合、StateDB ライターウィンドウは選択されたフィールド名を使用します。

注: Redis にはネイティブセカンダリインデックスのサポートがありません。セカンダリインデックス構造を作成するには、StateDB ライターウィンドウを使用する必要があります。

これは、SingleStore を使用する StateDB ライターウィンドウです。

```
<window-statedb-writer name="SSWriter" pubsub="true" >
  <statedb type="singlestore" hostname="generic.localhost" port="3306"
username="username" password="password" database="memsql_example"/> <!-- 1 -->
    <write table="mytable" ttl="ttl" ttl-offset="60" del-dead-sec-keys="true"> <!-- 2 -->
      <secondary-indexes>
        <secondary-index name="sec_idx">
          <fields>
            <field name="assetid"/>
          </fields>
        </secondary-index>
      </secondary-indexes>
      <output>
        <field-selection name="assetid"    db-name="db_assetid" />
        <field-selection name="tagid"      db-name="db_tagid" />
        <field-selection name="timestamp"  db-name="db_timestamp" />
        <field-selection name="value"      db-name="db_value" />
      </output>
    </write>
</window-statedb-writer>
```

1 ウィンドウは、ユーザー名 username とパスワード password を使用して、ポート 6379 で generic.localhost の"memsql_example"という SingleStore データベースにアクセスします。プロジェクトを作成する前に、特定のデータベースに関する同様の情報を収集してください。

2 データは mytable という名前のテーブルに書き込まれます。

StateDB リーダーウィンドウは、セカンダリインデックス sec_idx に基づいて (StateDB ライターウィンドウの出力からの) 入力値を集計します。

これは、Redis を使用した StateDB リーダーウィンドウです。

```
<window-statedb-reader name="aggregateReader" pubsub="true" >
  <statedb type="redis" hostname="generic.localhost" port="6379" cluster="false"
username="username" password="password" max-reconnect-retries="20" reconnect-retry-
delay="1000"/>
```

```
  <query prefix="mytable" query="assetid==db_assetid"/> <!-- 1 -->
   <schema>
    <fields>
     <field name="assetid" type="string" key="true"/>
     <field name="tagid" type="string" key="true"/>
     <field name="timestamp" type="int64" key="true"/>
     <field name="value" type="double"/>
     <field name="sum_value" type="double"/>
     <field name="avg_value" type="double"/>
     <field name="first_value" type="double"/>
     <field name="last_value" type="double"/>
     <field name="min_value" type="double"/>
     <field name="max_value" type="double"/>
     <field name="lag1_value" type="double"/>
    </fields>
   </schema>
   <output>
    <field-selection name="assetid" source="input"/> <!-- 2 -->
    <field-selection name="tagid" source="input"/>
    <field-selection name="timestamp" source="input"/>
    <field-selection name="value" source="input"/>
    <field-selection name="db_value" source="query" aggregate="ESP_aSum"/> <!-- 3 -->
    <field-selection name="db_value" source="query" aggregate="ESP_aAve"/>
    <field-selection name="db_value" source="query" aggregate="ESP_aFirst"/>
    <field-selection name="db_value" source="query" aggregate="ESP_aLast"/>
    <field-selection name="db_value" source="query" aggregate="ESP_aMin"/>
    <field-selection name="db_value" source="query" aggregate="ESP_aMax"/>
    <field-selection name="db_value" source="query" aggregate="ESP_aLag,1"/>
   </output>
   <connectors>
    ...<!-- 4 -->
   </connectors>
</window-statedb-reader>
```

1 クエリは、query タグで query="assetid==db_assetid"として指定されます。ウィンドウは assetid の値と Redis の"db_assetid"から取得した値を照合します。

2 このフィールドと次の 3 つの field-selection タグは、source=input を指定します。これら 4 つのフィールドの値は、ソースウィンドウによって処理された入力イベントから直接取得されます。

3 このフィールドと次の 6 つの field-selection タグは、source=query を指定します。これらの 6 つのフィールドの値は、データベースから取得されます。この場合、6 つのフィールドすべてがデータベースフィールド **db_value** を取得し、指定された集計関数を使用して取得した値を集計します。

4 ファイルおよびソケットコネクタを使用して、出力を CSV ファイルに書き込んで検証します。(実際のコードは示していません。)

これは、SingleStore を使用した StateDB リーダーウィンドウです。これは、Redis の対応するものと実質的に同じです。

```
<window-statedb-reader name="aggregateReader" pubsub="true" >
 <statedb type="singlestore" hostname="generic.localhost" port="3306"
username="username" password="password" database="memsql_example"/>
  <query table="mytable" query="assetid==db_assetid"/>
   <schema>
    <fields>
```

```
      <field name="assetid" type="string" key="true"/>
      <field name="tagid" type="string" key="true"/>
      <field name="timestamp" type="int64" key="true"/>
      <field name="value" type="double"/>
      <field name="sum_value" type="double"/>
      <field name="avg_value" type="double"/>
      <field name="first_value" type="double"/>
      <field name="last_value" type="double"/>
      <field name="min_value" type="double"/>
      <field name="max_value" type="double"/>
      <field name="lag1_value" type="double"/>
     </fields>
    </schema>
    <output>
     <field-selection name="assetid" source="input"/>
     <field-selection name="tagid" source="input"/>
     <field-selection name="timestamp" source="input"/>
     <field-selection name="value" source="input"/>
     <field-selection name="db_value" source="query" aggregate="ESP_aSum"/>
     <field-selection name="db_value" source="query" aggregate="ESP_aAve"/>
     <field-selection name="db_value" source="query" aggregate="ESP_aFirst"/>
     <field-selection name="db_value" source="query" aggregate="ESP_aLast"/>
     <field-selection name="db_value" source="query" aggregate="ESP_aMin"/>
     <field-selection name="db_value" source="query" aggregate="ESP_aMax"/>
     <field-selection name="db_value" source="query" aggregate="ESP_aLag,1"/>
    </output>
    <connectors>
     ...
    </connectors>
</window-statedb-reader>
```

StateDB ウィンドウを使用するその他のシナリオを次に示します。

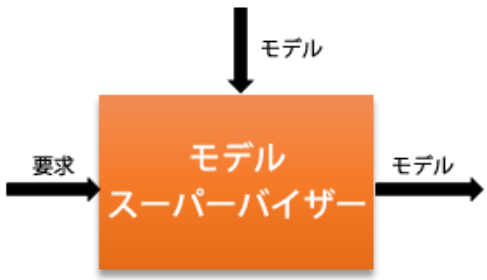
- 1 つのプロジェクトには、1 つの StateDB ライターウィンドウと複数の StateDB リーダーウィンドウを含めることができます。
- 1 つのプロジェクトに 1 つの StateDB ライターウィンドウが存在し、1 つ以上の StateDB リーダーウィンドウを含む他のプロジェクトと通信できます。
- 別の方法を使用して、Redis に状態データを入力できます(Redis CLI など)。その後、1 つ以上のセカンダリインデックスがデータベースに存在する 1 つ以上のプロジェクトを作成すると、Redis プライマリキーに対してのみクエリを実行できます。検索操作のみがサポートされています。 1 対多の結合と集計はサポートされていません。

5

分析

モデルスーパーバイザウィンドウの使用

*モデルスーパーバイザウィンドウ*は、モデルイベントのフローを管理します。入力要求イベントを使用すると、どのモデルを配置するか、いつ、どこにそれを配置するかを制御できます。モデルイベントはスコアウィンドウにパブリッシュされます。



モデルスーパーバイザウィンドウでは、任意の数のモデルイベントを受け取ることができます。ストリーミング分析プロジェクトでは、モデルイベントは、通常、学習ウィンドウまたはモデルリーダーウィンドウから送信されます。モデルイベントを受信すると、モデルスーパーバイザウィンドウは、モデルスーパーバイザウィンドウの配置モードとユーザーの要求に基づいて、他のストリーミング分析ウィンドウにイベントを処理し、パブリッシュします。

表 5.1　window-model-supervisor 要素のプロパティ

プロパティ	必須/オプション	説明
name	必須	ウィンドウ名を指定します。それは、次の文字のいずれかで始まる必要があります: _、AZ、AZ。名前の残りの部分には次の文字を含めることができます: _、az、AZ、0〜9 です。
deployment-policy	オプション	オフラインモデルの配置ポリシーを指定します。有効なオプションは、アイテムの受信直後にモデルイベントを受信ウィンドウに送信する **immediate**、コマンドラインで指定された要求にしたがってモデルイベントを送信する **on-demand** です。
capacity	オプション	許可するオフラインモデルの最大数を指定します。最大容量になった後、旧モデルのイベントは破棄されます。
pubsub	オプション	ウィンドウをパブリッシュやサブスクライブするかどうかを指定します。pubsub のプロジェクトレベルの値が **manual** の場合、**true** を指定するとパブリッシングとサブスクライブが有効になり、**false** を指定すると無効になります。

モデルスーパーバイザウィンドウ要素の一般的な例を次に示します。

```
<window-model-supervisor name='w_supervisor' deployment-policy='on-demand'
capacity='1000'>
   <connectors>
   ...
   </connectors>
</window-model-supervisor>
```

詳細については、*SAS Event Stream Processing: ストリーミング分析の適用*を参照してください。

モデルリーダーウィンドウの使用

ほとんどの場合、Model Reader ウィンドウは、オフラインモデルの場所と種類を参照する要求イベントを受け取ります。オフラインモデルは、ESP サーバーとは別に指定、開発、学習、および格納されます。

また、Model Reader ウィンドウは、Score ウィンドウまたは Model Supervisor ウィンドウにモデルメタデータを含むモデルイベントをパブリッシュします。

レコメンダーシステムの場合は、Model Reader ウィンドウ自体内でオフラインモデルのプロパティ値も指定します。詳細については、“レコメンダーシステムの使用” (*SAS Event Stream Processing: ストリーミング分析の適用*)を参照してください。

ONYX モデルの場合は、要求イベントを受信せずに、モデルリーダーウィンドウの中でオフラインモデルプロパティ値を指定します。

ONNX モデルの場合、Model Reader ウィンドウ内で前処理およびウォームアップパラメータを指定することもできます。

詳細については、“ONNX モデルの取り扱い” (*SAS Event Stream Processing: ストリーミング分析の適用*)を参照してください。

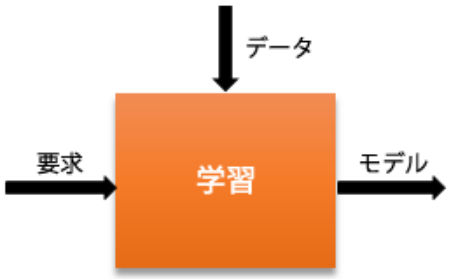
表 5.2　window-model-reader 要素のプロパティ

プロパティ	必須/オプション	説明
name	必須	ウィンドウ名を指定します。それは、次の文字のいずれかで始まる必要があります: _、a～z、A～Z。名前の残りの部分には次の文字を含めることができます: _、a～z、A～Z、0～9。
model-type	オプション	読み取ってスコアウィンドウに渡すモデルの種類を指定します。有効なオプションは、それぞれ分析ストアモデルの場合は **astore**、レコメンダーシステムのオフラインモデルの場合は **recommender** です。
pubsub	オプション	ウィンドウをパブリッシュやサブスクライブするかどうかを指定します。pubsub のプロジェクトレベルの値が **manual** の場合、**true** を指定するとパブリッシングとサブスクライブが有効になり、**false** を指定すると無効になります。

学習ウィンドウの使用

*学習ウィンドウ*はデータイベントを受け取り、モデルイベントをスコアウィンドウへパブリッシュします。それらは受信データイベントを使用して、リアルタイムでモデルパラメーターを開発して調整します。多くの場合、このデータはパターンを学習するための履歴データです。受信データには、予測しようとしている結果と関連する変数の両方が含まれている必要があります。

学習ウィンドウは要求イベントを受け取ることもできます。これらのイベントは、イベントが引き続きストリーミングされている間に学習アルゴリズムを調整できます。



学習ウィンドウがアルゴリズムを調整した後、調整されたモデルはモデルイベントを通じてスコアウィンドウまたはモデルスーパーバイザウィンドウに書き込まれます。

表 5.3　window-train 要素のプロパティ

プロパティ	必須/オプション	説明
name	必須	ウィンドウ名を指定します。それは、次の文字のいずれかで始まる必要があります。_、AZ、AZ。名前の残りの部分には次の文字を含めることができます。_、az、AZ、0〜9 です。
algorithm	必須	ウィンドウのアルゴリズムを指定します。プロジェクトがオンラインの場合、アルゴリズムを短い名前で指定する必要があります。
pubsub	オプション	ウィンドウをパブリッシュやサブスクライブするかどうかを指定します。pubsub のプロジェクトレベルの値が **manual** の場合、**true** を指定するとパブリッシングとサブスクライブが有効になり、**false** を指定すると無効になります。

学習ウィンドウ要素の一般的な例を次に示します。

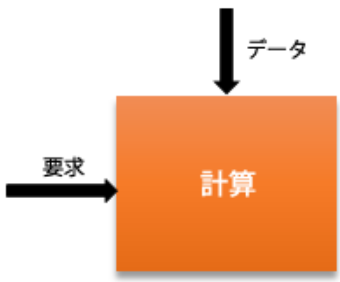
```
<window-train name='w_train' algorithm='algorithm_short_name'>
      <parameters>
       <properties>
        ...
       </properties>
      </parameters>
      <input-map>
       ...
      </input-map>
     </window-train>
```

詳細については、*SAS Event Stream Processing: ストリーミング分析の適用*を参照してください。

算出ウィンドウの使用

計算ウィンドウの概要

*計算ウィンドウ*は、確立された分析手法に基づきリアルタイムに統計情報を作成します。それらはデータイベントを受け取り、新しく変換されたスコアデータを出力イベントにパブリッシュします。(送信エッジには役割が割り当てられていないため、ダイアグラムには表示されません。)計算ウィンドウは要求イベントを受け取ることもできます。



計算ウィンドウは、データの正規化と変換方法の他、学習とスコアリングをまとめる学習モデル用に設計されています。

表 5.4 window-calculate エレメントのプロパティ

プロパティ	必須/オプション	説明
name	必須	ウィンドウ名を指定します。名前は、_、a-z、A-Z のいずれかの文字で始める必要があります。名前の 2 文字目以降には、_、a-z、

プロパティ	必須/オプション	説明
		A-Z、0-9 の文字を含めることができます。
algorithm	必須	ウィンドウのアルゴリズムを指定します。プロジェクトがオンラインの場合、アルゴリズムを短い名前で指定する必要があります。
collapse-updates	オプション	**true** の場合、複数の更新ブロックが 1 つの更新ブロックに集約されます。
exp-max-string	オプション	式エンジンがウィンドウに対して使用する文字列の最大サイズを指定します。デフォルト値は **1024** です。
index	オプション	ウィンドウのインデックスの種類を指定します。有効なオプションは、**'pi_RBTREE'**、**'pi_HASH'**、**'pi_LN_HASH'**、**'pi_CL_HASH'**、**'pi_FW_HASH'**、**'pi_EMPTY'**、**'pi_HLEVELDB'**、または **'pi_HLEVELDB_NC'**です。
output-insert-only	オプション	**true** の場合、ウィンドウが非挿入イベントを他のウィンドウに渡さないようにします。
pubsub	オプション	ウィンドウをパブリッシュやサブスクライブするかどうかを指定します。pubsub のプロジェクトレベルの値が **manual** の場合、**true** を指定するとパブリッシングとサブスクライブが有効になり、**false** を指定すると無効になります。
pubsub-index	オプション	パブリッシュ/サブスクライブ インデックス値を指定します。有効なオプションは、**'pi_RBTREE'**、**'pi_HASH'**、**'pi_LN_HASH'**、**'pi_CL_HASH'**、**'pi_FW_HASH'**、**'pi_EMPTY'**、

プロパティ	必須/オプション	説明
		'pi_HLEVELDB'、または **'pi_HLEVELDB_NC'**です。
pulse-interval='interval (unit)'	オプション	更新の正規バッチを書き込む間隔を指定します。interval の整数を渡します。unit で有効なオプションは、**microseconds**、**millseconds**、**seconds**、**minutes**、**hours**、**days** です。デフォルトは **milliseconds** です。

次は計算ウィンドウエレメントの一般的な例です。

```
<window-calculate name='w_calculate' algorithm='algorithm_short_name'>
        <schema>
          <fields>
            ...
          </fields>
        </schema>
        <parameters>
          <properties>
            ...
          </properties>
        </parameters>
        <input-map>
         ...
        </input-map>
        <output-map>
          ...
        </output-map>
        <connectors>
        ...
        </connectors>
      </window-calculate>
```

SAS Micro Analytic Service モジュールを定義して、計算ウィンドウ内で実行するコードブロックを指定できます。コードのブロックは、1 つ以上の関数を含むことができます。Python や DS2 でコードのブロックを書くことができます。Python でのコードについては、を参照してください。DS2 でのコードについては、"DS2 Programming for SAS Micro Analytic Service" (*SAS Micro Analytic Service: Programming and Administration Guide*)を参照してください。

関数を入力データを受け取るソースウィンドウにバインドするには、計算ウィンドウ内で SAS Micro Analytic Service マップを指定する必要があります。

SAS Micro Analytic Service モジュールの操作

概要

SAS Micro Analytic Service は、メモリ常駐のハイパフォーマンスなプログラム実行サービスです。SAS Micro Analytic Service モジュールは、基本的に、SAS Event Stream Processing プロジェクト内で実行するコードの名前付きブロックです。このコードブロックには、Python または DS2 で記述できる 1 つまたは複数の関数を含めることができます。モジュールはプロジェクトレベルで作成します。

SAS Micro Analytic モジュール内での Python の使用

環境変数

SAS Micro Analytic モジュール内で Python 関数を使用するためには、2 つの環境変数を設定します。

- MAS_M2PATH は、必要な Python プログラム、mas2py.py へのフルパス名を指定します。SAS Event Stream Processing を Kubernetes 環境で運用しているか、オンプレミスで運用しているかにかかわらず、この環境変数の値を変更する必要はありません。
- MAS_PYPATH は、Python 実行可能ファイルへのフルパス名を指定します。SAS Event Stream Processing が Kubernetes 環境で動作する場合、この名前は Python の埋め込みバージョンを指します。本製品をオンプレミスで運用している場合は、Python のインストールされたバージョン(3.x)の場所に設定してください。たとえば、setMAS_PYPATH=/usr/bin/python です。

SAS Micro Analytic モジュール内では、Python コードが標準出力(stdout)および標準誤差(stderr)に書き込むメッセージは保存されません。これらのメッセージを外部ファイルに保存するには、MAS_PYOUT_FILE 環境変数を設定する必要があります。この変数の値はフルパスとファイル名として指定します。

デフォルトでは、このファイルはプロジェクトが再起動されるたびに上書きされます。ファイルに追加するには、MAS_PYOUT_FILE に指定する値に+を追加します(たとえば、MAS_PYOUT_FILE="+/mnt/data/project/output/python.log")。

Kubernetes 環境では、 **設定**ページの**ユーザーセッション**セクションで MAS_PYOUT_FILE の値を設定します。次に例を示します:

注: 前の図は、プロジェクトパッケージの使用を前提としています。詳細については、"Project Package" (*SAS Event Stream Processing: Using SAS Event Stream Processing Studio*)を参照してください。

エッジ環境では、システムコンソールで MAS_PYOUT_FILE の値を設定します。

```
export MAS_PYOUT_FILE=/directory/filename
```

Python へのパッケージの追加

Python コードを実行する SAS Micro Analytic モジュールに追加の Python パッケージが必要な場合は、SAS Configurator for Open Source (sas-pyconfig)ユーティリティを使用して、カスタムインストールをダウンロード、構成、ビルドします。詳細については、"Configure Open Source Integration Points" (*SAS Viya Platform: Deployment Guide*)を参照してください。

このユーティリティを実行すると、Persistent Volume Claim (PVC)内に新しい Python がインストールされます。この新しいインストールを使用するには、espconfig-sas-pyconfig.yaml の ESPConfig 要素のテンプレートに sas-pyconfigPVC ボリュームを追加します。

次に例を示します。

```
apiVersion: builtin
kind: PatchTransformer
metadata:
  name: patch-transformer-esp-pyconfig
patch: |-
  - op: add
    path: /spec/projectTemplate/deployment/spec/template/spec/volumes/-
    value:
      name: python-volume
```

```
      persistentVolumeClaim:
        claimName: sas-pyconfig
  - op: add
    path: /spec/projectTemplate/deployment/spec/template/spec/containers/0/
volumeMounts/-
    value:
      name: python-volume
      mountPath: /opt/sas/viya/home/sas-pyconfig
      readOnly: true
target:
  group: iot.sas.com
  kind: ESPConfig
  name: sas-esp-operator
  version: v1
```

このパッチを site-config に配置し、transformers セクションの kustomization.yaml に行を追加します。

```
site-config/espconfig-sas-pyconfig.yaml
```

この Python インスタンスを指すように、SAS Event Stream Processing Studio および SAS Event Steam Manager の MAS_PYPATH 変数を更新します。前述の例に基づくと、値**は/opt/sas/viya/home/sas-pyconfig/default_py/bin/python3** である必要があります。

SAS Event Stream Processing Studio の MAS_PYPATH を更新する方法については、“Specify Environment Variables and ESP Server Properties for All Users” (*SAS Event Stream Processing: Using SAS Event Stream Processing Studio*)を参照してください。

SAS Event Steam Manager でのアップデート方法については、“Global Settings” (*SAS Event Stream Processing: Using SAS Event Stream Manager*)を参照してください。

後で Python ライブラリを追加するには、sas-pyconfig を再実行します。追加のプロファイルを追加することもできます。実行する Python インストールのプロファイルを指すように MAS_PYPATH を更新する必要があります。

SAS Micro Analytic Service マップの定義

関数をソースウィンドウにバインドするには、算出ウィンドウで SAS Micro Analytic Service マップを定義します。このバインドは、ソースウィンドウの入力ハンドラーとして機能します。

次の例で、株式取引のトータルコストの算出について考えてみましょう。

```
<project name="MASPython" threads="1" pubsub="auto" heartbeat-interval="1"
luaroot="@ESP_PROJECT_OUTPUT@/luaroot">
  <mas-modules>
    <mas-module module="module1" language="python" func-names="compTotal"> <!-- 1 --
>
      <code><![CDATA[def compTotal(quantity, price):<!-- 2 -->
    "Output: total"
    total= quantity * price
    return total]]></code>
    </mas-module>
```

```
 </mas-modules>
 <contqueries>
  <contquery name="cq1">
   <windows>
    <window-source name="src_win">
     <schema>
      <fields>
       <field name="ID" type="string" key="true"/>
       <field name="security" type="string"/>
       <field name="quantity" type="double"/>
       <field name="price" type="double"/>
      </fields>
     </schema>
     <connectors>
      <connector name="New_Publisher_1" class="fs">
       <properties>
        <property name="type"><![CDATA[pub]]></property>
        <property name="fsname"><![CDATA[@ESP_PROJECT_HOME@/files/
python_trades.csv]]></property>
        <property name="fstype"><![CDATA[csv]]></property>
       </properties>
      </connector>
     </connectors>
    </window-source>
    <window-calculate name="pw1" algorithm="MAS">
     <schema> <!-- 3 -->
      <fields>
       <field name="ID" type="string" key="true"/>
       <field name="security" type="string" key="false"/>
       <field name="quantity" type="double" key="false"/>
       <field name="price" type="double" key="false"/>
       <field name="total" type="double" key="false"/>
      </fields>
     </schema>
     <mas-map>
      <window-map module="module1" function="compTotal" revision="0"
source="src_win"/><!-- 4 -->
     </mas-map>
     <connectors>
      <connector name="New_Subscriber_1" class="fs">
       <properties>
        <property name="type"><![CDATA[sub]]></property>
        <property name="snapshot"><![CDATA[false]]></property>
        <property name="fsname"><![CDATA[@ESP_PROJECT_OUTPUT@/output.csv]]></
property>
        <property name="fstype"><![CDATA[csv]]></property>
       </properties>
      </connector>
     </connectors>
    </window-calculate>
   </windows>
   <edges>
    <edge source="src_win" target="pw1" role="data"/><!-- 5 -->
   </edges>
  </contquery>
 </contqueries>
```

```
</project>
```

1 module1 という名前の SAS Micro Analytic Service モジュールは、MASPython プロジェクトのプロジェクトレベルで定義されています。これには Python で書かれた関数 compTotal が含まれています。

2 compTotal の Python コードは<code>エレメント内に指定されています。この関数は、src_win ソースウィンドウから pw1 算出ウィンドウに渡されるすべてのイベントの入力ハンドラーとして機能します。

3 スキーマはで、受信イベントの構造を定義するフィールドを指定します。取引されている証券に関連するデータ、株式数、および現在の価格が算出ウィンドウにストリーミングされます。スキーマは、compTotal 関数によって算出される合計も指定します。

4 算出ウィンドウのマップレベルで、ウィンドウマップは module1 の compTotal 関数をソースウィンドウにバインディングします。

5 ソースウィンドウと算出ウィンドウの間のエッジには、役割 data が必要です。

注: 前のコードブロックは、プロジェクトパッケージのディレクトリ内の入力ファイルと出力ファイルを参照します。詳細については、“Project Package” (*SAS Event Stream Processing: Using SAS Event Stream Processing Studio*)を参照してください。

算出ウィンドウには、SAS Micro Analytic Service モジュールマップを 1 つしか配置できません。このマップには、1 つまたは複数のウィンドウマップを含めることができます。

たとえば、次のコードでは、DS2 関数を含む 2 つの SAS Micro Analytic Service モジュールがプロジェクトレベルで定義されています。

```
<mas-modules>
  <mas-module language="ds2" module="module_1" func-names='compute_volume'>
    <code>
     <![CDATA[
         ds2_options sas;
         package myModule_1/overwrite=yes;
         method compute_volume(int quantity, double price, in_out int volume);
         volume = quantity * price;
         end;
         endpackage;
     ]]>
    </code>
  </mas-module>
  <mas-module language="ds2" module="module_2" func-names='compute_price_sqr'>
    <code>
     <![CDATA[
         ds2_options sas;
         package myModule_2/overwrite=yes;
         method compute_price_sqr(int quantity, double price, in_out double price_sqr);
         price_sqr = price * price;
         end;
         endpackage;
     ]]>
    </code>
  </mas-module>
</mas-modules>
```

ESP サーバーは、XML コードに表示される順序でモジュールをロードします。

SAS Micro Analytic Service マップ内の 2 つのウィンドウマップは、DS2 関数を算出ウィンドウにバインドします:

```
...
<mas-map>
  <window-map module="module_1" revision="0" source="Trades1"
function="compute_volume"/>
  <window-map module="module_2" revision="0" source="Trades2"
function="compute_price_sqr"/>
</mas-map>
...
```

ウィンドウマップには 4 つの属性があります。

- module は、以前に定義されたモジュールの名前を参照します
- revision は、0 に設定する必要があります
- source は、ハンドラーを指定するソースウィンドウの名前です
- function は、モジュールに定義されている関数を指定します

SAS Micro Analytics Service モジュールを定義するために使用する XML エレメントの詳細については、*“ストリーミング分析ウィンドウに関連する XML 言語要素” (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*を参照してください。

複数の派生イベントの生成

SAS Micro Analytic Service モジュールのメソッドでは、1 つ以上の配列を書き込むことができます。少なくとも 1 つの出力配列名が派生イベントフィールド名と一致する場合は、単一の入力イベントから複数の派生イベントを生成できます。

生成されるイベントの数は、実行時に一致する配列内のエレメントの数によって決定されます。

SAS Event Stream Processing では、すべての挿入イベントに一意のキーが必要です。したがって、ソースイベントのキーは、複数の生成された派生イベントで重複させることはできません。キーが重複すると、SAS Event Stream Processing は処理を停止し、エラーを返します。

SAS Micro Analytics Service モジュール機能を算出ウィンドウにマップするときは、派生した各イベントに一意のキー値が設定されていることを確認してください。

次の入力イベントについて考えてみましょう。

```
i,n,1,field1,field2,field3
```

プロジェクトのソースウィンドウでは、次のような入力スキーマを使用しているとします。

```
<schema>
  <fields>
    <field name="ID" type="int32" key="true"/>
    <field name="field1" type="string"/>
    <field name="field2" type="string"/>
    <field name="field3" type="string"/>
```

```
  </fields>
</schema>
```

また、算出ウィンドウの出力スキーマは次のとおりです。

```
<schema>
  <fields>
    <field name="ID" type="int32" key="true"/>
    <field name="_masRowNum" type="int32" key="true"/>
    <field name="stringD" type="string"/>
  </fields>
</schema>
```

算出ウィンドウで次の SAS Micro Analytic Service マップを使用しているとします。

```
<mas-map>
    <window-map module="masds2" revision="0" source="Src" function="transpose"/>
 </mas-map>
```

次の SAS Micro Analytic Servic モジュールを使用して、入力イベントを転置することができます。

```
<mas-modules>
 <mas-module language="ds2" module="masds2" func-names='transpose'>
  <code>
   <![CDATA[
        ds2_options sas;
        package test_package;
          method transpose(nchar(50) field1, nchar(50) field2,
                           nchar(50) field3, in_out nchar(50)
                           stringD[3]);
            stringD[1] = trim(field1);
            stringD[2] = trim(field2);
            stringD[3] = trim(field3);
          end;
        endpackage;
      ]]>
  </code>
 </mas-module>
</mas-modules>
```

出力イベントは次のようになります。

```
i,n,1,1,field1
i,n,1,2,field2
i,n,1,3,field3
```

ソースウィンドウの ID と算出ウィンドウの_masRowNum の組み合わせが、算出ウィンドウの出力スキーマのキーとなります。また、算出ウィンドウの出力スキーマの stringD というフィールドが、必要に応じて DS2 コードの stringD[]という配列名と一致していることにも注意してください。

詳細については、“Unique Keys” (*SAS Micro Analytic Service: Programming and Administration Guide*)を参照してください。

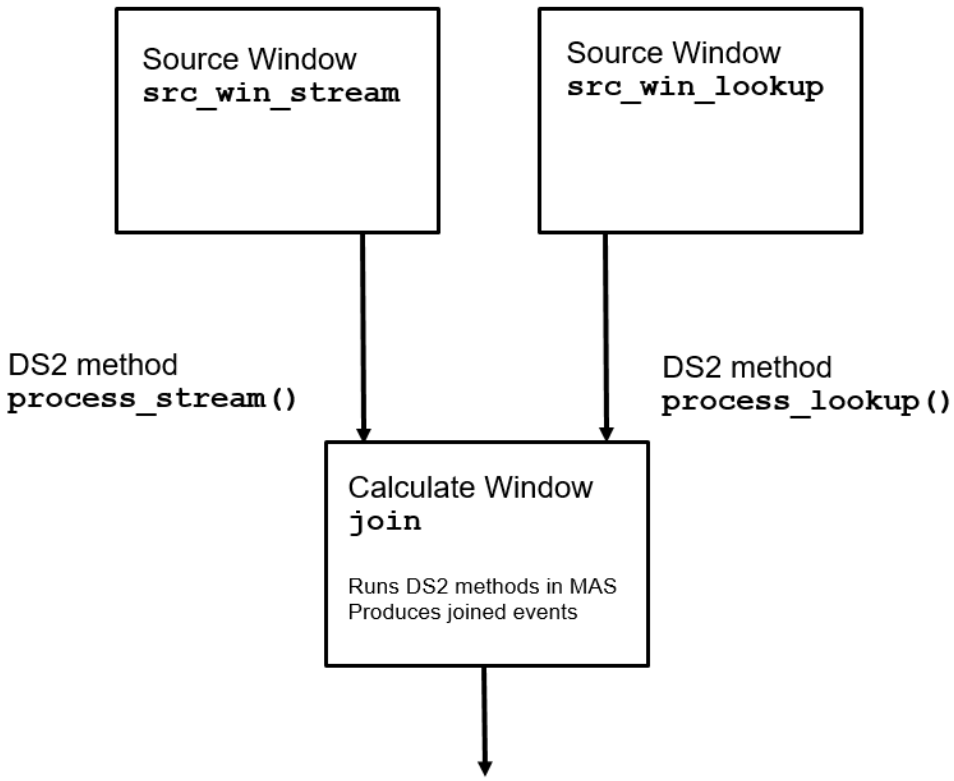
共有ベクトルと共有ハッシュテーブルのデータ構造の使用

SAS Micro Analytics Service では、ユーザーコンテキスト内の実行モジュール間でデータを共有する 2 つの方法として、共有ベクトルと共有ハッシュテーブルが提供されます。*共有ベクトル*は、データ値のコレクションです。ベクトル内の値には、複数のデータ型を混在させることができます。*共有ハッシュテーブル*はベクトルのコンテナであり、そこにベクトルを格納し、キーを使用してアクセスします。これらの構造はどちらもスレッドセーフでロックフリーです。それらを使用して、SAS Event Stream Processing プロジェクト内の DS2 メソッド間でデータを共有できます。

これらのデータ構造の詳細については、"State Sharing between Modules" (*SAS Micro Analytic Service: Programming and Administration Guide*)を参照してください。

次のクエリでは、2 つのソースウィンドウによる計算ウィンドウへのイベントのストーミングが示されます。共有ハッシュテーブルを使用する 2 つの DS2 メソッドが、SAS Micro Analytic Service モジュール内で実行されます。モジュールでは、非再生成内部結合がシミュレートされます。計算ウィンドウ内でモジュールが実行され、結合されたイベントが生成されます。

図 5.1　ハッシュテーブルを共有する SAS Micro Analytic Service モジュールによる連続クエリ



SAS Micro Analytic Service モジュールのコードは次のとおりです。

```
<mas-modules>
  <mas-module language="ds2" module="module_1"
        func-names='process_lookup,process_stream'>
   <code>
    <![CDATA[
      ds2_options sas;
       package module_1/overwrite=yes;

       dcl package masstate /* 1 */ st();
       dcl package logger logr('DF.ESP');

       method process_stream /* 2 */(varchar(16) _inOpcode, bigint lookupID,
           in_out varchar matchString,
           in_out varchar matchDescription,
           in_out varchar _outOpcode) ;
         dcl int rc;
         dcl varchar(32) key;

         key = lookupID;
         _outOpcode='noop'; /* 3 */

         /* logr.log('d', 'key=$s', key); */
         if (0 eq st.get(key)) then do;
           matchString=st.getString(key,0);
           matchDescription=st.getString(key,1);
           _outOpcode=_inOpcode; /* 4 */
         end;
         /* logr.log('d', 'inOpcode=$s, outOpcode=$s', _inOpcode, _outOpcode); *//* 5 */
       end;

       method process_lookup /* 6 */(bigint ID, varchar(32) value,
           varchar(4096) description,
           in_out varchar _outOpcode);
         dcl int rc;
         dcl varchar(32) key;

         _outOpcode='noop';
         key = ID;
         rc = st.createVector(key,2);  /* 7 */
         rc = st.setString(key, 0, value); /* 8 */
         rc = st.setString(key, 1, description);
         rc = st.put(key); /* 9 */

         rc = st.deleteVector(key); /* 10 */
       end;
      endpackage;
      ]]>
     </code>
  </mas-module>
</mas-modules>
```

1 共有ハッシュテーブルを使用する場合は、パッケージ MASSTATE を使用して、データの作成、共有、取得、削除を行います。

2 メソッド process_stream は、受け取ったイベントを共有ハッシュテーブルに挿入するように設計されています。また、_inOpcode を参照することで更新イベントと削除イベントを処理することもできます。

3 デフォルトは **noop** です。つまり、イベントは生成されません。

4 一致時に Opcode を設定し、イベントを生成します。

5 デバッグメッセージをログに書き込むには、この行のコメントを解除します。

6 メソッド process_lookup は、共有ハッシュテーブルのルックアップ ID を通じて文字列を検索するように設計されています。一致するものが見つからない場合、_outOpcode は **noop** なので、イベントは生成されません。

提示されたメソッドで挿入イベントが処理されます。これを変更して_inOpcode を調べ、共有ハッシュテーブルのエントリを追加、更新または削除することができます。

7 新しいベクトルを作成します。

8 イベントデータを挿入します。

9 ベクトルをハッシュに入れます。

10 ベクトルを削除します。

2 つのソースウィンドウの XML コードは次のとおりです。

```
<window-source name='src_win_lookup' index='pi_RBTREE'>
    <schema-string>ID*:int64,value:string,description:string</schema-string>
    <connectors>
      <connector class='fs' name='pub'>
        <properties>
          <property name='type'>pub</property>
          <property name='fstype'>csv</property>
          <property name='fsname'>lookup.csv</property>
        </properties>
      </connector>
    </connectors>
</window-source>

<window-source name='src_win_stream' index='pi_RBTREE'>
  <schema-string>ID*:int64, lookupID:int64, price:double, quant:int64</schema-string>
  <connectors>
    <connector class='fs' name='pub'>
      <properties>
        <property name='type'>pub</property>
        <property name='fstype'>csv</property>
        <property name='fsname'>stream.csv</property>
      </properties>
    </connector>
  </connectors>
</window-source>
```

メソッドの SAS Micro Analytic Service マップを含む計算ウィンドウの XML コードは、次のとおりです。

```
<window-calculate name='join' algorithm='MAS'>
  <schema-string>ID*:int64,matchID:int64,matchString:string,matchDescription:string,
            price:double,quant:int64</schema-string>
  <mas-map>
```

```
    <window-map module="module_1" revision="0" source="src_win_lookup"
function="process_lookup"/>
    <window-map module="module_1" revision="0" source="src_win_stream"
function="process_stream"/>
  </mas-map>
  <connectors>
   <connector class='fs' name='sub'>
    <properties>
     <property name='type'>sub</property>
     <property name='fstype'>csv</property>
     <property name='fsname'>result.csv</property>
     <property name='snapshot'>true</property>
    </properties>
   </connector>
  </connectors>
</window-calculate>
```

ソースウィンドウと計算ウィンドウの間のエッジの XML コードは次のとおりです。

```
<edges>
  <edge source='src_win_lookup' target='join' role='data'/>
  <edge source='src_win_stream' target='join' role='data'/>
</edges>
```

コネクタグループの XML コードは次のとおりです。ソースウィンドウと計算ウィンドウを含む連続クエリの外にこのコードを配置します。

```
<project-connectors>
    <connector-groups>
      <connector-group name="group1"> <!-- 1 -->
        <connector-entry connector='cq_01/join/sub' state='running'/>
        <connector-entry connector='cq_01/src_win_lookup/pub' state='finished'/>
      </connector-group>
      <connector-group name="group2">
        <connector-entry connector='cq_01/src_win_stream/pub' state='finished'/>
      </connector-group>
    </connector-groups>
    <edges>
      <edge source='group1' target='group2'/>
    </edges>
  </project-connectors>
```

1 *コネクタのグループ*は、connector-entry エレメントのコンテナです。コネクタエントリでは、コネクタの状態を指定できます。

ここで、ソースウィンドウ src_win_lookup に次のイベントをストリーミングするとします。

```
i,n,10001,sunw,"Workstation manufacturer"
i,n,20001,ibm,"From typewriters to mainframes"
```

次に、ソースウィンドウ src_win_stream に次のイベントをストリーミングします。

```
i,n,1,10001,101.45,100
i,n,2,20001,23.5,1000
i,n,3,20001,10.1,80
i,n,4,10001,10.2,85
```

XML の結合データは次のとおりです。

```
<event opcode='insert' window='project_01/cq_01/join'>
   <value name='ID'>1</value>
   <value name='matchDescription'>Workstation manufacturer</value>
   <value name='matchString'>sunw</value>
   <value name='price'>101.450000</value>
   <value name='quant'>100</value>
 </event>

 <event opcode='insert' window='project_01/cq_01/join'>
   <value name='ID'>2</value>
   <value name='matchDescription'>From typewriters to mainframes</value>
   <value name='matchString'>ibm</value>
   <value name='price'>23.500000</value>
   <value name='quant'>1000</value>
 </event>

 <event opcode='insert' window='project_01/cq_01/join'>
   <value name='ID'>3</value>
   <value name='matchDescription'>From typewriters to mainframes</value>
   <value name='matchString'>ibm</value>
   <value name='price'>10.100000</value>
   <value name='quant'>80</value>
 </event>

 <event opcode='insert' window='project_01/cq_01/join'>
   <value name='ID'>4</value>
   <value name='matchDescription'>Workstation manufacturer</value>
   <value name='matchString'>sunw</value>
   <value name='price'>10.200000</value>
   <value name='quant'>85</value>
 </event>
```

スコアウィンドウの使用

*スコアウィンドウ*はモデルイベントを受け取り、受信データイベントの予測を行います。それらは得点データを生成します。このスコアデータを使用して、学習したモデルに基づいて予測を生成することができます。(出力エッジには役割が割り当てられていないため、図には表示されません)。

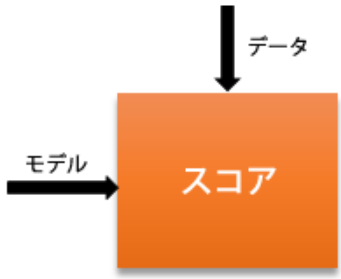


表 5.5　window-score 要素のプロパティ

プロパティ	必須/オプション	説明
name	必須	ウィンドウ名を指定します。それは、次の文字のいずれかで始まる必要があります。_、AZ、AZ。名前の残りの部分には次の文字を含めることができます。_、az、AZ、0〜9 です。
model-type	オプション	読み取ってスコアウィンドウに渡すモデルの種類を指定します。有効なオプションは、それぞれ分析ストアモデルの場合は **astore**、レコメンダーシステムのオフラインモデルの場合は **recommender** です。
pubsub	オプション	ウィンドウをパブリッシュやサブスクライブするかどうかを指定します。pubsub のプロジェクトレベルの値が **manual** の場合、**true** を指定するとパブリッシングとサブスクライブが有効になり、**false** を指定すると無効になります。

スコアウィンドウ要素の一般的な例を次に示します。

```
<window-score name='w_score'>
        <schema>
          <fields>
           ...
          </fields>
        </schema>
        <models>
         ...
        </models>
      </window-score>
```

詳細については、*SAS Event Stream Processing: ストリーミング分析の適用*を参照してください。

6

カスタムウィンドウ

カスタムウィンドウ操作

概要

カスタムウィンドウを使用すると、エンドユーザーはプロジェクト間で外部コードを再利用できます。まず、開発者は、Python コードまたは Lua コードを使用してカスタムウィンドウを作成する構成ファイルを作成します。次に、誰かが SAS Event Stream Processing Studio を使用して、エンドユーザーが利用できるようにします。

構成ファイルは、カスタムウィンドウの構成と、受信イベントに対してウィンドウによって実行されるアクションの両方を指定します。ファイル内のコードは、Python と Lua の両方で使用できるすべての共通サーバー機能にアクセスできます。create 関数は、出力イベントを生成するコード内で指定する必要があります。

入力ウィンドウからのイベントストリームは、カスタムウィンドウで指定されたプロパティにマップされます。カスタムウィンドウは、create 関数を呼び出すときにこれらのプロパティの値を使用します。コードは 1 つ以上のオブジェクトをカスタムウィンドウに返し、オブジェクトからのフィールドを出力イベントのイベントフィールドにマップします。

このトピックでは、開発者がカスタム ウィンドウのコードを作成する方法について説明します。開発者がカスタムウィンドウを管理する方法とユーザーがそれらを使用する方法の詳細については、“Working with Custom Windows” (*SAS Event Stream Processing: Using SAS Event Stream Processing Studio*)を参照してください。

カスタムウィンドウの例と、SAS Event Stream Processing Studio に追加できるカスタムウィンドウの取得については、SAS Event Stream Processing Studio Custom Windows GitHub を参照してください。

構成ファイルの作成

概要

受信イベントに対してカスタムウィンドウが実行するアクションを指定するコードは、ESP サーバーで使用される 4 つの関数を参照します。

- create- 1 つ以上のイベントを生成するために呼び出される関数。

  > 重要　この関数は必須です。コードにこの関数が含まれていない場合、ESP サーバーはエラーを返します。

- init- プログラム値を初期化するオプションの関数。ウィンドウが作成された後に呼び出されます。
- heartbeat- プロジェクトハートビート間隔で呼び出されるオプションの関数。
- destroy- ウィンドウが破棄されたときに呼び出されるオプションの関数。

コードには、ウィンドウ構成を定義する_espconfig_というグローバル変数が含まれている必要があります。_espconfig_グローバル変数は、Python ディクショナリまたは Lua テーブルとして指定されています。

受信および発信イベントの処理

このコードは、イベントストリーミングプロセス中に適切なタイミングで呼び出される関数のエントリポイントを指定します。唯一必要な関数は、create 関数です。

ウィンドウが作成されると、init 関数が呼び出されます。構成ファイルで定義された初期化データが渡されます。関数のシグネチャは次のようになります(Python および Lua):

```
def init(settings):
function init(settings)
```

ウィンドウが入力イベントを受信すると、create 関数が呼び出されます。この関数のシグネチャは、パラメーターが展開されているかどうかによって 2 種類の形式のいずれかになります。

- expand-parms の値が **false** の場合、シグネチャは次のようになります。

  ```
  def create(data,context):
  function create(data,context)
  ```

ここで data は、構成ファイルの inputVariables セクションで定義されたフィールドを含むオブジェクトです。context パラメーターは、カスタムウィンドウの名前と入力ウィンドウの名前を含むオブジェクトです。

- expand-parms の値が **true** の場合、シグネチャには構成ファイルの inputVariables セクションの各エントリのパラメーターが含まれます。

次の入力変数を処理するとします:

```
"inputVariables": {
  "desc" : "Optional description of the input variables",
  "fields" : [
    {
      "name": "trade_symbol",
      "desc": "Stock symbol"
    },
    {
      "name": "trade_price",
      "desc": "Trade price"
    },
    {
      "name": "trade_quantity",
      "desc": "Trade quantity"
    }
  ],
}
```

その場合、関数のシグネチャは次のようになります:

```
def create(symbol,price,quant):
function create(symbol,price,quant)
```

シーケンスは常に、構成内で入力が定義されている順序になります。名前が構成内の名前と一致する必要がないことに注意してください。

create 関数は、構成ファイルの outputVariables セクションにフィールド名を含む 0 個以上のオブジェクトを返します。

この例は、株式取引データを扱う出力変数を示しています。

```
"outputVariables": {
  "desc" : "Optional description of the output variables",
  "fields" : [
    {
      "name": "maxp",
      "desc": "Maximum price"
    },
    {
      "name": "maxq",
      "desc": "Maximum quantity"
    }
  ],
}


def create(symbol,price,quant):

  ...

  event = {}
```

```
event["symbol"] = symbol
event["maxp"] = maxprice
event["maxq"] = maxquant

return event
```

create 関数から null 値を返すと、イベントは生成されません。オブジェクトのリストを返す場合、リスト内の各アイテムに対してイベントが生成されます。

カスタムウィンドウは、構成ファイルの outputVariables セクションで定義された任意の値を<plugin>要素で指定されたイベント フィールドにマップします。さらに、カスタムウィンドウは、入力ウィンドウと共通するフィールドをすべて新しいイベントにコピーします。

heartbeat カスタムウィンドウがサーバーからハートビートを受信したときに呼び出されます。これはプロジェクトで定義された間隔で発生します。

シグネチャは次のとおりです:

```
def heartbeat(context):
function heartbeat(context)
```

context パラメーターは、カスタムウィンドウの名前を含むオブジェクトです。

destroy 関数は、カスタムウィンドウがサーバーで破棄されたときに呼び出されます。

シグネチャは次のとおりです:

```
def destroy():
function destroy()
```

Lua または Python コードは、例外が発生する可能性があります。init 関数からの例外により、プロジェクトの読み込みに失敗します。create 関数、heartbeat 関数、およびスニペット(スレッド化された場合)からの例外は、on-script-exception 属性に基づいて処理されます。詳細については、“カスタムウィンドウを定義する”を参照してください。

ウィンドウ構成オブジェクトの指定

ウィンドウ構成オブジェクトを _espconfig_という名前の変数として指定します。Python ディクショナリまたは Lua テーブルのいずれかを使用します。このオブジェクトに使用する言語は、ウィンドウで実行されるコードの言語と同じである必要があります。

_espconfig_は、次のエントリを含んでいます。

- settings- カスタムウィンドウの操作設定を含むオブジェクトです。サポートされているプロパティは次のとおりです。
  - desc - 設定のオプションの説明。
  - expand-parms - create 関数が、inputVariable セクションの各エントリに対して個別のパラメータを受け取るか、入力変数の名前にマップされるフィールドを含むオブジェクトを受け取るかを決定します。このプロパティのデフォルトは **false** です。

次に、settings の例を示します。プロパティが false のエントリ:

```
"settings" : {
   "expand-parms":false
}
```

入力変数の名前が a、b、および c であると仮定すると、create 関数のシグネチャは次のようになります:

```
def create(data,context):
   aVal = data["a"]
   bVal = data["b"]
   cVal = data["c"]
   ...
```

プロパティが true の settings エントリの例を次に示します:

```
"settings" : {
   "expand-parms":true
}
```

create 関数のシグネチャは次のようになります:

```
def create(aVal,bVal,cVal):
   ...
```

- process-blocks - create 関数が複数のイベントを含むブロックでイベントを受信するか一度に 1 つのイベントを受信するかを決定します。このプロパティのデフォルトは **false** です。

  注: process-blocks が **true** および expand-parms が **true** の場合、エラーが返されます。

- encode-binary - create 関数に渡されるバイナリデータが Base64 でエンコードされているかどうかを決定します。プロパティのデフォルトは **false** です。

- inputVariables - 受信ウィンドウからの入力データをカスタムウィンドウの create 関数にマップするオブジェクトを含む配列。各オブジェクトは、次のフィールドが含まれています。
  - fields - には、マップ対象のフィールドが含まれています。
  - name - create 関数に渡されるオブジェクトを設定するために使用される名前です(パラメーターが展開されていない場合)。
  - desc - フィールドのオプションの説明。
  - esp_type- カスタムウィンドウのユーザーが入力変数へのマップを許可するフィールドのデータ型を指定します。
  - optional - フィールドがオプションかどうかを示すオプションのブール値。**true** の場合、入力フィールドのマッピングエントリはプロジェクトに必要ありません。

  次の例は、株式取引データを処理する ESP プロジェクトの inputVariables 配列を示しています。カスタムウィンドウは、シンボルごとに取引データを処理しています:

```
"inputVariables" : {
      "desc" : "Variables from the input event",
      "fields" : [
```

```
        {
          "name": "trade_symbol",
          "desc": "Stock symbol",
          "esp_type": "string",
          "optional": False
        },
        {
          "name": "trade_price",
          "desc": "Trade price",
          "esp_type": "int64",
          "optional": False
        },
        {
          "name": "trade_quantity",
          "desc": "Trade quantity",
          "esp_type": "int64",
          "optional": False
        }
      ]
    }
```

- outputVariables - create 関数からの出力データを出力のイベントフィールドにマップするオブジェクトを含む配列。各オブジェクトには次のフィールドがあります:
  - fields - には、マップ対象のフィールドが含まれています。
  - name - create 関数に渡されるオブジェクトを設定するために使用される名前です(パラメーターが展開されていない場合)。
  - desc - フィールドのオプションの説明。
  - esp_type- カスタムウィンドウのユーザーが出力変数へのマップを許可されているフィールドのデータ型を指定します。

  次の例は、シンボルごとに取引の最大価格と数量を追跡するカスタムウィンドウの outputVariables 配列を示しています:

```
"outputVariables" : {
    "desc" : "Variables created for the output event",
    "fields" : [
      {
        "name": "maxp",
        "desc": "Maximum price",
        "esp_type": "int64"
      },
      {
        "name": "maxq",
        "desc": "Maximum quantity",
        "esp_type": "int64"
      }
    ]
  }
```

  create 関数は、maxp および maxq フィールドを持つ 1 つ以上のオブジェクトを返します。カスタムウィンドウは、<output-map>を使用してこれらのフィールドを適切な出力イベントフィールドにマップします。

```
<output-map>
 <properties>
```

```
  <property name="maxp" esp_type="int64">max_price</property>
  <property name="maxq" esp_type="int64">max_quant</property>
 </properties></output-map>
```

この場合、カスタムウィンドウは max_price の値を maxp に、および max_quant の値を maxq に設定し、データ型が int64 である必要があることを指定します。入力ウィンドウとカスタムウィンドウに共通する他のフィールドは、すべてそのままコピーされます。

- initialization- init 関数に渡されるオブジェクトを含む配列。カスタムウィンドウは値を単一のデータオブジェクトに配置し、関数に渡します。この配列が構成に存在する場合は、init 関数でないとエラーが発生します。

  各オブジェクトには次のフィールドがあります:

  - fields - には、マップ対象のフィールドが含まれています。
  - name - プロパティの名前。
  - desc - プロパティのオプションの説明。
  - default - オプションのプロパティのデフォルト値。この値は、モデルが <plugin>要素に値を提供しない場合に使用されます。デフォルト値が指定されておらず、<plugin>要素にエントリが含まれていない場合、プロパティは init 関数に送信されません。
  - input_type- 初期化値がカスタムウィンドウのユーザーにドロップダウンリストとして表示されることを指定します。input_type = "dropdown"が存在する場合、ドロップダウンリストが使用されます。input_type が存在しない場合、テキストボックスが使用されます。
  - values- カスタムウィンドウのユーザーが選択できる初期化値のカンマ区切りのリストを指定します。

  次の例は、シンボルごとに取引の最大価格と数量を追跡するカスタムウィンドウの initialization 配列を示しています:

```
"initialization" : {
      "desc" : "Some initialization values",
      "fields" : [
        {
          "name": "symbols",
          "desc": "",
          "default": "AAPL",
          "input_type": "dropdown",
          "values": ["AAPL","IBM","T"]
        }
      ]
    }
```

サンプル_espconfig_は、Python で記述されています:

```
_espconfig_ = {
   "settings" : {
      "desc" : "Some settings for the custom window",
      "expand_parms" : True,
      "process_blocks" : False,
      "encode_binary" : False
   },
   "inputVariables" : {
      "desc" : "Variables from the input event",
```

```
      "fields" : [
        {
          "name": "trade_symbol",
          "desc": "Stock symbol",
          "esp_type": "string",
          "optional": False
        },
        {
          "name": "trade_price",
          "desc": "Trade price",
          "esp_type": "int64",
          "optional": False
        },
        {
          "name": "trade_quantity",
          "desc": "Trade quantity",
          "esp_type": "int64",
          "optional": False
        }
      ]
    },
    "outputVariables" : {
      "desc" : "Variables created for the output event",
      "fields" : [
        {
          "name": "maxp",
          "desc": "Maximum price",
          "esp_type": "int64"
        },
        {
          "name": "maxq",
          "desc": "Maximum quantity",
          "esp_type": "int64"
        }
      ]
    },
    "initialization" : {
      "desc" : "Some initialization values",
      "fields" : [
        {
          "name": "symbols",
          "desc": "",
          "default": "AAPL",
          "input_type": "dropdown",
          "values": ["AAPL","IBM","T"]
        }
      ]
    }
  }
```

サンプル_espconfig_は、Lua で記述されています:

```
  espconfig = {
    settings = {
      desc = "Some settings for the custom window",
      expand_parms = true,
      process_blocks = false,
```

```
    encode_binary = false
  },
  inputVariables = {
    desc = "Variables from the input event",
    fields = {
      {
        name = "trade_symbol",
        desc = "Stock symbol",
        esp_type = "string",
        optional = false
      },
      {
        name = "trade_price",
        desc = "Trade price",
        esp_type = "int64",
        optional = false
      },
      {
        name = "trade_quantity",
        desc = "Trade quantity",
        esp_type = "int64",
        optional = false
      }
    }
  },
  outputVariables = {
    desc = "Variables created for the output event",
    fields = {
      {
        name = "maxp",
        desc = "Maximum price",
        esp_type = "int64"
      },
      {
        name = "maxq",
        desc = "Maximum quantity",
        esp_type = "int64"
      }
    }
  },
  initialization = {
    desc = "Some initialization values",
    fields = {
      {
        name = "symbols",
        desc = "",
        default = "AAPL",
        input_type = "dropdown",
        values = ["AAPL", "IBM", "T"]
      }
    }
  }
}
```

カスタムウィンドウを定義する

次の XML コードは、カスタムウィンドウの一般的な構造を示しています。

```
<window-custom name="" index="">
 <schema>
  <fields>
   <field name="" type="" key=""/>
   ...
  </fields>
 </schema>
 <plugin code="" on-script-exception="log-and-continue | stop-and-remove">
  <input-map>
   <properties>
    <property name="" esp_type="">...</property>
    ...
   </properties>
  </input-map>
  <output-map>
   <properties>
    <property name="" esp_type="">...</property>
    ...
   </properties>
  </output-map>
  <initialization>
   <properties>
    <property name="" values="">...</property>
    ...
   </properties>
  </initialization>
 </plugin></window-custom>
```

このコードの最上位レベル要素は、<window-custom>カスタムウィンドウとして識別されます。割り当てた name で、SAS Event Stream Processing Studio にカスタムウィンドウが表示されます。カスタムウィンドウの schema を指定する必要があります。

plugin 要素には、コードの場所と入力値と出力値のマッピングに関する情報が含まれています。

- code 属性は、カスタムウィンドウをサポートするために使用されるコードの場所を指定します。
- on-script-exception 属性は、ウィンドウコードの実行中に発生する例外の処理方法を指定するオプション属性です。log-and-continue に設定すると、エラーがログに記録され、実行が続行されます。stop-and-remove に設定すると、致命的なエラーがログに記録され、プロジェクトが停止し削除されます。この値が指定されていない場合、ウィンドウは含まれるプロジェクトで定義されたスクリプト処理のビヘイビアーを継承します。
- input-map 要素は、入力イベントからの ESP イベント変数を create 関数に渡されるデータにマップします。name 属性には、構成データの inputVariables セクションの名前が含まれ、要素のコンテンツには入力フィールドの名前が含まれます。

内容が指定されていない場合、name 値が使用されます。esp_type 属性は、ウィンドウのユーザーが入力変数へのマップを許可するフィールドのデータ型を指定します。

- output-map 要素は、create 関数から返されたデータの値を出力イベントの ESP イベント値にマップします。name 属性には、構成データの outputVariables セクションの名前が含まれ、要素のコンテンツには出力フィールドの名前が含まれます。内容が指定されていない場合、name 値が使用されます。esp_type 属性は、カスタムウィンドウのユーザーが出力変数へのマップを許可されているフィールドのデータ型を指定します。
- オプションの initialization 要素には、コード内の init 関数に渡される名前と値のペアのセットが含まれています。プロパティの名前は、構成データの初期化セクションのエントリと一致する必要があります。values 属性は、ウィンドウのユーザーが選択できる初期化値のカンマ区切りのリストを指定します。

注: initialization エントリを指定する場合は、コードに init 関数を含める必要があります。そうしないと、エラーが生成されます。

カスタムウィンドウの例

次のプロジェクトでは、株式取引データを処理します。特定の株式記号を検索し、個々の取引の最大数量と価格を計算するカスタムウィンドウを使用します。初期化エントリは、対象のティカーシンボルを設定するために使用されます。

Python で書かれた構成ファイルを次に示します。

```
symboldata = {}
symbols = None

def init(settings):
   global symbols
   if "symbols" in settings:
      symbols = {}
      for s in settings["symbols"].split(","):
         symbols[s] = True

def create(symbol,price,quant):
   global symboldata

   event = None
   if symbols == None or symbol in symbols:
      if (symbol in symboldata) == False:
         data = {"max_price":price,"max_quant":quant}
         symboldata[symbol] = data
         event = {}
         event["symbol"] = symbol
         event["maxp"] = price
         event["maxq"] = quant
      else:
```

```
            data = symboldata[symbol]

            if (price > data["max_price"] or quant > data["max_quant"]):
               if (price > data["max_price"]):
                  data["max_price"] = price
               if (quant > data["max_quant"]):
                  data["max_quant"] = quant
               event = {}
               event["symbol"] = symbol
               event["maxp"] = data["max_price"]
               event["maxq"] = data["max_quant"]
               event["_opcode"] = "upsert"

      return event

_espconfig_ = {
   "settings" : {
      "desc" : "Some settings for the custom window",
      "expand_parms" : True,
      "process_blocks" : False,
      "encode_binary" : False
   },
   "inputVariables" : {
      "desc" : "Variables from the input event",
      "fields" : [
         {
            "name": "trade_symbol",
            "desc": "Stock symbol",
            "esp_type": "string",
            "optional": False
         },
         {
            "name": "trade_price",
            "desc": "Trade price",
            "esp_type": "int64",
            "optional": False
         },
         {
            "name": "trade_quantity",
            "desc": "Trade quantity",
            "esp_type": "int64",
            "optional": False
         }
      ]
   },
   "outputVariables" : {
      "desc" : "Variables created for the output event",
      "fields" : [
         {
            "name": "maxp",
            "desc": "Maximum price",
            "esp_type": "int64"
         },
         {
            "name": "maxq",
            "desc": "Maximum quantity",
```

```
            "esp_type": "int64"
          }
        ]
      },
      "initialization" : {
        "desc" : "Some initialization values",
        "fields" : [
          {
            "name": "symbols",
            "desc": "",
            "default": "AAPL",
            "input_type": "dropdown",
            "values": ["AAPL","IBM","T"]
          }
        ]
      }
    }
```

プロジェクトの XML は次のとおりです:

```
<project name="p" index="pi_EMPTY" pubsub="none" threads="4">
 <contqueries>
  <contquery name="cq" trace="plugin">
   <windows>
    <window-source name="trades">
     <schema>
      <fields>
       <field name="id" key="true" type="int64"/>
       <field name="symbol" type="string"/>
       <field name="currency" type="int32"/>
       <field name="time" type="int64"/>
       <field name="msecs" type="int32"/>
       <field name="price" type="double"/>
       <field name="quant" type="int32"/>
       <field name="venue" type="int32"/>
       <field name="broker" type="int32"/>
       <field name="buyer" type="int32"/>
       <field name="seller" type="int32"/>
       <field name="buysellflg" type="int32"/>
      </fields>
     </schema>
     <connectors>
      <connector class="fs" active="true">
       <properties>
        <property name="type">pub</property>
        <property name="fstype">csv</property>
        <property name="fsname">input.csv</property>
       </properties>
      </connector>
     </connectors>
    </window-source>
    <window-custom name="plugin">
     <schema>
      <fields>
       <field name="symbol" type="string" key="true"/>
       <field name="max_price" type="double"/>
       <field name="max_quant" type="int32"/>
```

```
        </fields>
       </schema>
       <plugin code="file://symbols.py">
        <input-map>
         <properties>
          <property name="trade_symbol" esp_type="string">symbol
          </property>
          <property name="trade_price" esp_type="double">price
          </property>
          <property name="trade_quantity" esp_type="int32">quant
          </property>
         </properties>
        </input-map>
        <output-map>
         <properties>
          <property name="maxp" esp_type="double">max_price
          </property>
          <property name="maxq" esp_type="int32">max_quant
          </property>
         </properties>
        </output-map>
        <initialization>
         <properties>
          <property name="symbols" values="AAPL,IBM,T">AAPL
          </property>
         </properties>
        </initialization>
       </plugin>
      </window-custom>
     </windows>
     <edges>
      <edge source="trades" target="plugin" />
     </edges>
    </contquery>
</contqueries></project>
```

7

高度なトピック

生成されたイベントを出力スロット間で分割

概要

ほとんどのウィンドウタイプは、スプリッター関数または式を登録して、新しく生成されたイベントに使用する出力スロットを決定することができます。この機能を使用すると、生成されたイベントを一連の出力スロットに送信することができます。デフォルトでは、ウィンドウはイベントを出力スロット 0 から 0 個以上の下流のウィンドウに送信します。

スプリッター関数のあるウィンドウとダウンストリームウィンドウの間にエッジを を追加する場合は、ダウンストリームウィンドウがその入力イベントを受け取る親ウィンドウのスロット番号を指定します。

ウィンドウスプリッターを使用する方が、2 つ以上の後続のフィルターウィンドウを使用するより効率的です。これは、フィルタリングが、各フィルターに対して複数回ではなく、ウィンドウスプリッターで 1 回実行されるためです。その結果、データの移動と処理が少なくなります。

Lua でのスプリッター関数の使用

出力イベントのスロット番号を指定するには、スロット番号に対応する整数を返す Lua 関数を作成します。次のコード例を考えてみましょう。

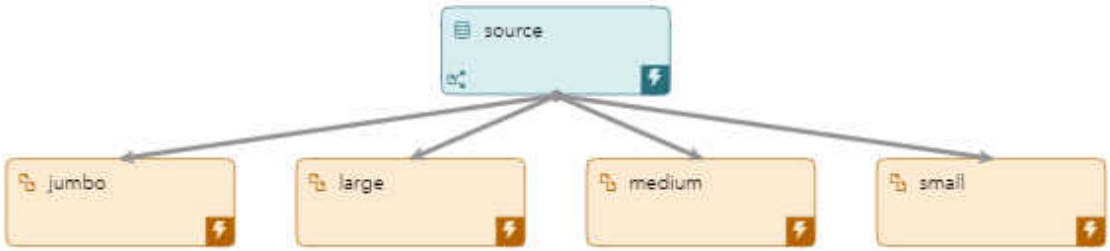
```
<window-...>
  <splitter-lua function="function">
    <use>...</use>
    <code><![CDATA[
      ...
      Lua function that returns an integer = slot number
      ...
    ]]></code>
  </splitter-lua>
</window-...>
```

use 要素が指定されている場合、ESP サーバーは入力イベント全体またはイベントフィールドのサブセットを指定された関数に渡します。

パフォーマンスと効率性を実現するため、use パフォーマンスを指定します。たとえば、100 個のフィールドを持つ入力イベントがあり、Lua コードでそのうちの 2 つのフィールドしか使用しない場合、それら 2 つのフィールドを use 要素内にリストします。

注: Lua コードに init という名前の関数がある場合、モデルがスプリッターの初期化を開始する前にこの関数が呼び出されます。

次のプロジェクトを考えてみましょう:



データストリームをソースウィンドウに交換します。ソース ウィンドウは、GetSlot という Lua 関数で定義されたスロットを使用して、データを 4 つのコピーウィンドウのいずれかに送信します。スロットは、取引のサイズによって決定されます(quant フィールド)。コピーウィンドウには、取引のサイズによって small、medium、large、と jumbo という名前が付けられています。

コードでは、入力ファイルを格納するためにプロジェクトパッケージが使用されていることに注意してください。

```
<project name="lua_splitter" index="pi_HASH" pubsub="auto" threads="4">
 <contqueries>
  <contquery name="cq">
   <windows>
    <window-source name="source" insert-only="true">
     <schema>
      <fields>
       <field name="id" type="int64" key="true"/>
       <field name="symbol" type="string" key="false"/>
       <field name="price" type="double" key="false"/>
       <field name="quant" type="int32" key="false"/>
      </fields>
     </schema>
     <connectors>
      <connector class="fs" name="fs">
       <properties>
        <property name="type"><![CDATA[pub]]></property>
        <property name="fsname"><![CDATA[@ESP_PROJECT_HOME@/test_files/
trades.csv]]></property>
        <property name="fstype"><![CDATA[csv]]></property>
       </properties>
      </connector>
     </connectors>
     <splitter-lua function="getSlot">
      <use>quant</use>
      <code><![CDATA[
       function getSlot(data)
        if (data.quant < 100) then
         return 0
```

```
        elseif (data.quant < 1000) then
          return 1
        elseif (data.quant < 10000) then
          return 2
        else
          return 3
        end
      end
    ]]></code>
  </splitter-lua>
 </window-source>
 <window-copy name="small">
  <retention type="bytime_sliding">10 seconds</retention>
 </window-copy>
 <window-copy name="medium">
  <retention type="bytime_sliding">10 seconds</retention>
 </window-copy>
 <window-copy name="large">
  <retention type="bytime_sliding">10 seconds</retention>
 </window-copy>
 <window-copy name="jumbo">
  <retention type="bytime_sliding">10 seconds</retention>
 </window-copy>
</windows>
<edges>
 <edge source="source" target="small" slot="0"/>
 <edge source="source" target="medium" slot="1"/>
 <edge source="source" target="large" slot="2"/>
 <edge source="source" target="jumbo" slot="3"/>
</edges>
</contquery>
</contqueries>
</project>
```

Lua のスプリッター関数で使用できる組み込み関数については、8 章, “共通の Lua 機能”を参照してください。

Python でのスプリッター関数の使用

出力イベントのスロット番号を指定するには、スロット番号に対応する整数を返す Python 関数を作成します。 次のコード例を考えてみましょう:

```
<window-...>
  <splitter-python function="function">
    <use>...</use>
    <code><![CDATA[
      ...
      Python function that returns an integer = slot number
      ...
    ]]></code>
  </splitter-python>
</window-...>
```

use 要素が指定されている場合、ESP サーバーは入力イベント全体またはフィールドのサブセットを指定された関数に渡します。

パフォーマンスと効率性を実現するため、use エレメントを指定します。たとえば、100 個のフィールドを持つ入力イベントがあり、Python コードでのみ使用する場合、それらのフィールドを use 要素内にリストします。

注: Python コードに init という名前の関数がある場合、モデルがスプリッターの初期化を開始する前にこの関数が呼び出されます。

次のプロジェクトでは、取引データを取得し、スロットを使用して 4 つのコピーウィンドウのいずれかにデータを送信します。スロットは、取引のサイズによって決定されます(quant フィールド)。コピーウィンドウには、取引のサイズによって small、medium、large、と jumbo という名前が付けられています。この例では、入力ファイルを含む project package の使用方法も示します。

```
<project name="python_splitter" index="pi_HASH" pubsub="auto" threads="4">
 <contqueries>
  <contquery name="cq">
   <windows>
    <window-source name="source" insert-only="true">
     <schema>
       <fields>
         <field name="id" type="int64" key="true"/>
         <field name="symbol" type="string" key="false"/>
         <field name="price" type="double" key="false"/>
         <field name="quant" type="int32" key="false"/>
       </fields>
     </schema>
     <connectors>
       <connector class="fs" name="fs">
         <properties>
           <property name="type"><![CDATA[pub]]></property>
           <property name="fsname"><![CDATA[@ESP_PROJECT_HOME@/test_files/
trades.csv]]></property>
           <property name="fstype"><![CDATA[csv]]></property>
         </properties>
       </connector>
    </connectors>
     <splitter-python function="getSlot">
       <use>quant</use>
       <code><![CDATA[
def getSlot(data):
    if (data["quant"] < 100):
        return(0)
    elif (data["quant"] < 1000):
        return(1)
    elif (data["quant"] < 10000):
        return(2)
    else:
        return(3)
       ]]></code>
     </splitter-python>
    </window-source>
    <window-copy name="small">
```

```
    <retention type="bytime_sliding">10 seconds</retention>
   </window-copy>
   <window-copy name="medium">
    <retention type="bytime_sliding">10 seconds</retention>
   </window-copy>
   <window-copy name="large">
    <retention type="bytime_sliding">10 seconds</retention>
   </window-copy>
   <window-copy name="jumbo">
    <retention type="bytime_sliding">10 seconds</retention>
   </window-copy>
  </windows>
  <edges>
   <edge source="source" target="small" slot="0"/>
   <edge source="source" target="medium" slot="1"/>
   <edge source="source" target="large" slot="2"/>
   <edge source="source" target="jumbo" slot="3"/>
  </edges>
 </contquery>
</contqueries>
</project>
```

次のプロジェクトは、Python connector からのデータを処理する点を除いて、前のプロジェクトと同じです。

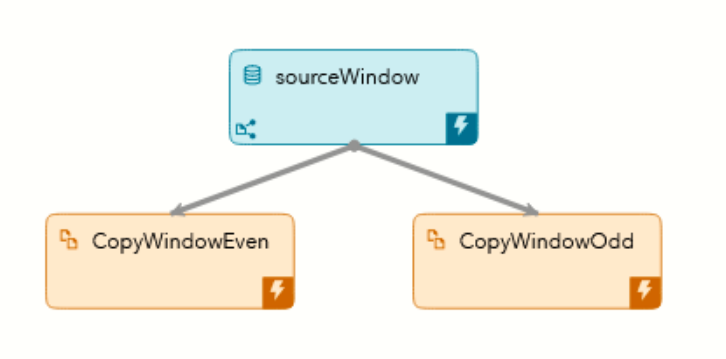
```
<project name="python_splitter2" index="pi_HASH" pubsub="auto" threads="4">
 <contqueries>
  <contquery name="cq">
   <windows>
    <window-source name="source" insert-only="true">
     <schema>
       <fields>
         <field name="id" type="int64" key="true"/>
         <field name="symbol" type="string" key="false"/>
         <field name="price" type="double" key="false"/>
         <field name="quant" type="int32" key="false"/>
       </fields>
     </schema>
     <connectors>
       <connector class="python" name="Trades">
         <properties>
           <property name="type">pub</property>
           <property name="code"><![CDATA[data = [
   {"id": "991", "symbol": "ibm", "price": "840.68", "quant": "36099"},
   {"id": "992", "symbol": "goog", "price": "171.69", "quant": "364"},
   {"id": "993", "symbol": "nvda", "price": "460.77", "quant": "44992"},
   {"id": "994", "symbol": "msft", "price": "929.5", "quant": "5"},
   {"id": "995", "symbol": "amzn", "price": "381.3", "quant": "18358"},
   {"id": "996", "symbol": "ibm", "price": "496.77", "quant": "47064"},
   {"id": "997", "symbol": "goog", "price": "480.13", "quant": "700"},
   {"id": "998", "symbol": "nvda", "price": "570.87", "quant": "33048"},
   {"id": "999", "symbol": "msft", "price": "785.11", "quant": "3129"},
   {"id": "1000", "symbol": "amzn", "price": "740.77", "quant": "48678"},
   {"id": "1001", "symbol": "ibm", "price": "191.07", "quant": "5790"},
   {"id": "1002", "symbol": "goog", "price": "74.7", "quant": "36007"},
   {"id": "1003", "symbol": "nvda", "price": "178.28", "quant": "25722"},
   {"id": "1004", "symbol": "msft", "price": "482.54", "quant": "18697"},
```

```
    {"id": "1005", "symbol": "amzn", "price": "888.2", "quant": "13810"},
    {"id": "1006", "symbol": "ibm", "price": "715.83", "quant": "39065"},
    {"id": "1007", "symbol": "goog", "price": "208.4", "quant": "8644"},
    {"id": "1008", "symbol": "nvda", "price": "607.88", "quant": "27985"},
    {"id": "1009", "symbol": "msft", "price": "259.78", "quant": "23"},
    {"id": "1010", "symbol": "amzn", "price": "144.64", "quant": "2383"}
]
def publish():
    return {"events": data, "done": True}
]]></property>
</properties>
      </connector>
    </connectors>
     <splitter-python function="getSlot">
       <use>quant</use>
       <code><![CDATA[
def getSlot(data):
    if (data["quant"] < 100):
        return(0)
    elif (data["quant"] < 1000):
        return(1)
    elif (data["quant"] < 10000):
        return(2)
    else:
        return(3)
       ]]></code>
     </splitter-python>
    </window-source>
    <window-copy name="small">
     <retention type="bytime_sliding">10 seconds</retention>
    </window-copy>
    <window-copy name="medium">
     <retention type="bytime_sliding">10 seconds</retention>
    </window-copy>
    <window-copy name="large">
     <retention type="bytime_sliding">10 seconds</retention>
    </window-copy>
    <window-copy name="jumbo">
     <retention type="bytime_sliding">10 seconds</retention>
    </window-copy>
   </windows>
   <edges>
    <edge source="source" target="small" slot="0"/>
    <edge source="source" target="medium" slot="1"/>
    <edge source="source" target="large" slot="2"/>
    <edge source="source" target="jumbo" slot="3"/>
   </edges>
  </contquery>
 </contqueries>
</project>
```

式エンジン言語でのスプリッター関数の使用

次のプロジェクトは、1 つのソースウィンドウと 2 つのコピーウィンドウで構成されています。ソースウィンドウには、イベントを 2 つのスロットのどちらに送信するのかを決定するスプリッターとしてユーザー定義関数が含まれています。各スロットは異なるコピーウィンドウに移動します。

図 7.1　スプリッターを使用したモデル



ソースウィンドウは次のとおりです:

```
<window-source index="pi_RBTREE" pubsub="true" name="sourceWindow">
        <splitter-expr>
         <expr-initialize>
          <udfs>
           <udf name="udf1" type="int32"><![CDATA[return ID%2]]></udf>
           <udf name="udf2" type="string"><![CDATA[return upper(symbol)]]></udf>
           <udf name="udf3" type="double"><![CDATA[return price]]></udf>
          </udfs>
         </expr-initialize>
         <expression><![CDATA[udf1() and (match_string(udf2(), "IBM")) and (udf3() >
100.0)]]></expression>
        </splitter-expr>
        <schema>
         <fields>
          <field name="ID" type="int32" key="true"/>
          <field name="symbol" type="string"/>
          <field name="price" type="double"/>
         </fields>
        </schema>
        <connectors>
         <connector class="fs" name="input_splitter">
          <properties>
           <property name="type"><![CDATA[pub]]></property>
           <property name="fsname"><![CDATA[@ESP_PROJECT_HOME@/test_files/
input_slots.csv]]></property>
           <property name="fstype"><![CDATA[csv]]></property>
          </properties>
         </connector>
        </connectors>
```

```
</window-source>
```

ソースウィンドウは、ファイルおよびソケットコネクタを使用して、**input.csv** という名前の CSV ファイルから入力イベントを受信します。イベントには、ID、会社のティッカーシンボル、販売価格が含まれます。

```
i, n, 1, ibm, 99.3
i, n, 2, amzn, 1626.03
u, n, 2, amzn, 1629.99
i, n, 3, appl, 197
p, n, 3, appl, 198
i, n, 5, goog, 1094.58
d, n, 1, ibm, 95.5
i, n, 11, fb, 138.68
u, n, 11, fb, 137.5
i, n, 4, amzn, 1633.2
i, n, 9, ibm, 123.46
u, n, 9, ibm, 126.44
p, n, 3, appl, 197.2
i, n, 6, nflx, 253.9
p, n, 5, goog, 1096.8
p, n, 11, fb, 139.5
i, n, 7, tsla, 180.8
```

ソースウィンドウにはスプリッタが含まれています。*式エンジン言語*で記述された 3 つのユーザー定義関数は、どのスロットを使用するかを評価するために使用される式を初期化します。

表 7.1 ユーザー定義関数の結果

ユーザー定義関数	説明
udf1	ID に対してモジュロ演算を実行します。ID は INT32 なので、整数を返します。ID mod 2 の結果は、ID が偶数の場合は 0、ID が奇数の場合は 1 になります。したがって、udf1 は、ID が奇数(余り 1)の場合はブール値 **true** を返し、ID が偶数(余り 0)の場合は **false** を返します。
udf2	match_string 関数を使用して、返される文字列にサブ文字列「IBM」が含まれているかどうかを確認します。
udf3	関数の数値出力を比較して、それが 100.0 より大きいかどうかを判断します。

各イベントがストリーミングされると、次の式によって結果が取得され、そのターゲットスロットが決定されます。

```
udf1() and (match_string(udf2(), "IBM")) and (udf3() > 100.0)
```

したがって、IBM の株価評価が 100.00 を超えると、イベントはスロット 1 にルーティングされます。

ソースウィンドウとコピーウィンドウ間のエッジによってスロットが指定されます。

```
<edge source="sourceWindow" target="CopyWindowEven" slot="0"/>
<edge source="sourceWindow" target="CopyWindowOdd" slot="1"/>
```

コピーウィンドウは次のとおりです:

```
<window-copy index="pi_RBTREE" pubsub="true" name="CopyWindowEven">
    <retention type="bycount_sliding"><![CDATA[1000]]></retention>
</window-copy>

<window-copy index="pi_HASH" pubsub="true" name="CopyWindowOdd">
    <retention type="bycount_sliding"><![CDATA[1000]]></retention>
</window-copy>
```

受信イベントをストリーミングした後、CopyWindowEven で処理されるイベントは次のとおりです:

```
Insert    7    tsla    180.8
Delete    11    fb    137.5
Update block    11    fb    139.5
Delete    5    goog    1,094.58
Update block    5    goog    1,096.80
Insert    6    nflx    253.9
Delete    3    appl    198
Update block    3    appl    197.2
Insert    4    amzn    1,633.20
Delete    11    fb    138.68
Update block    11    fb    137.5
Insert    11    fb    138.68
Delete    1    ibm    99.3
Insert    5    goog    1,094.58
Delete    3    appl    197
Update block    3    appl    198
Insert    3    appl    197
Delete    2    amzn    1,626.03
Update block    2    amzn    1,629.99
Insert    2    amzn    1,626.03
Insert    1    ibm    99.3
```

CopyWindowOdd によって処理されるイベントは次のとおりです:

```
Delete    1    ibm    123.46
Update block    1    ibm    126.44
Insert    1    ibm    123.46
```

保持の理解

ソースまたはコピーウィンドウでは、イベントストリームへの削除の導入を規定する保持ポリシーを設定できます。これらの削除は、途中でモデルを再計算してデータフローに沿って動作します。内部的に生成された削除は保持フラグでフラグが立てられ、この削除に基づくすべての以降のウィンドウ操作にフラグが立てられます。

注: 状況によっては、SAS Micro Analytic Service メソッドを算出ウィンドウにマップするときに、複数の派生イベントを生成することができます。この場合、保持フラグの伝播は確実に行われません。

たとえば、スライディングボリュームベースの 保持ポリシーが 2 のソースウィンドウを考えてみましょう。ソースウィンドウには常に最大 2 つのイベントが含まれます。挿入が到着してソースウィンドウが 3 つのイベントに成長すると、最終更新時刻が最も古いイベントが削除されます。そのイベントの削除が実行されます。

保持種類	説明
time-based (時間ベース)	保持は、イベントの年齢の関数として実行されます。イベントの年齢は、現在時刻からイベントの最終変更時刻を引いたものとして算出されます。時刻は、システム時刻またはイベントに埋め込まれた TIME フィールドから取得されます。時間ベースの保持を持つウィンドウは、イベントの到着によって設定された現在時刻を使用します。 **注:** 時間ベースの保持に指定できる最小時間は 1 秒です。
volume-based (ボリュームベース)	保持は指定されたレコード数に基づいて行われます。ボリュームがその仕様を超えて増加すると、最も古いイベントが削除されます。

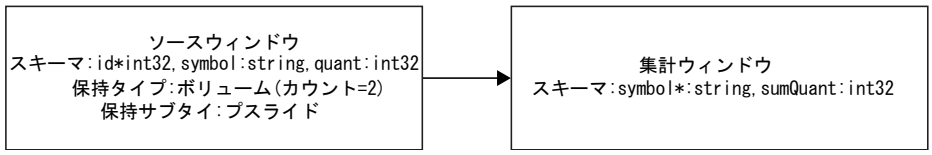
時間とボリュームベースの保持は、次の 2 つの方法のいずれかで発生します。

保持バリアント	説明
sliding (スライド)	イベントを削除する連続的なプロセスを指定します。保持ウィンドウが連続的にスライドすると考えてください。ボリュームベースのスライディングウィンドウの場合、指定したしきい値に達すると、入ってくる各挿入に対して 1 つの削除が実行されます。
jumping (ジャンプ)	指定されたしきい値に達すると内容を完全に消去するウィンドウを指定します。10 分ごとの内容をすべて削除するものとして、10 分間のジャンプウィンドウを考えてみましょう。

標準的な一連のイベントは、キーごとに最大で 1 つのイベントが存在するような縮小されたイベントセットです。同じキーに対する複数の更新と、同じキーに対する複数の更新を挿入したものが折りたたまれています。保持を持つウィンドウは、標準的な一連の変更(イベント)を生成します。次に、保持生成された削除を標準イベントセットの最後に追加します。プロセスの最後に、最終出力ブロックを形成します。

保持機能を持つウィンドウでは、次の形式のイベントブロックが出力されます。{<canonical output events>, <canonical retention deletes>}。他のすべてのウィンドウは、次の形式の出力ブロックを生成します。{<canonical output events>}になります。

次のモデルを考えてみましょう。



次の表記は、イベントを表すために使用されます。[<opcode>/<flags>: f1, ... ,fn]

- 演算コード
  - i-挿入
  - d-削除
  - ub-更新ブロック-ub としてマークされたすべてのイベントの 後には常に d とマークされたイベントが続きます。
- フラグ
  - n-標準
  - r-保持を生成

次のイベントがモデルにストリームされるとします。

ソースイン	ソースアウト-集計	集計
[i/n: 1,ibm,10]	[i/n: 1,ibm,10]	[i/n: ibm,10]
ソースイン	ソースアウト-集計	集計
[i/n: 2,ibm,11]	[i/n: 2,ibm,11]	[ub/n: ibm,21] [d/n: ibm,10]
ソースイン	ソースアウト-集計	集計
[i/n: 3,sas,100]	[i/n: 3,sas,100]	[i/n: sas,100]
	[d/r: 1,ibm,10]	[ub/r: ibm,11] [d/r: ibm,21]
ソースイン	ソースアウト-集計	集計
[i/n: 4,ibm,12]	[i/n: 4,ibm,12]	[ub/r: ibm,12]
	[d/r: 2,ibm,11]	[d/r: ibm,11]

保持追跡モードで実行すると、保持および非保持の変更がシステム全体で引き合わされます。システムがユーザーイベントを処理すると、システムは保持の削除を生成します。ユーザーイベントの結果と保持された削除の結果の両方がシステムにプッシュされます。結果をどのように解釈するかを決めることができます。標準の保持モードでは、これらの 2 つのイベントは、その出力イベントを標準的にレンダリングすることによって、単一のイベントに結合できます。

ソースイン	ソースアウト-集計	集計
[i/n: 1,ibm,10]	[i/n: 1,ibm,10]	[i/n: ibm,10]
ソースイン	ソースアウト-集計	集計
[i/n: 2,ibm,11]	[i/n: 2,ibm,11]	[ub/n: ibm,21] [d/n: ibm,10]
ソースイン	ソースアウト-集計	集計
[i/n: 3,sas,100]	[i/n: 3,sas,100]	[i/n: sas,100]
	[d/r: 1,ibm,10]	[ub/r: ibm,11] [d/r: ibm,21]
ソースイン	ソースアウト-集計	集計
[i/n: 4,ibm,12]	[i/n: 4,ibm,12]	[ub: ibm,23] [d/n: ibm,11]
	[d/r: 2,ibm,11]	[ub/r: ibm,12] [d/r: ibm,23]

ここでは、集計ウィンドウの出力は、最後の入力イベントのため、非正規です。保持追跡モードでは、入力イベントにキーのユーザー入力と同じキーの保持イベントが含まれる場合、キーごとに 2 つの操作を実行できます。

注: パルスモードセットを持つウィンドウは、常に出力イベントの正規ブロックを生成します。パルスが設計どおりに機能するためには、ウィンドウは特定の閾値 TIME まで出力イベントをバッファリングする。出力ブロックは送信前に標準的にレンダリングされます。

実際の保持の例については、4 種類の保持種類とバリアントの組み合わせについて、次の使用例を検討してください。

保持種類とバリアント	ユースケースの例
bytime_sliding	■ リアルタイムダッシュボード ■ 直近のイベントを継続的に分析します
bytime_jumping	■ 期間アラーム ■ ”平均圧力が現在の時間内に 2 ATM に達したら通知します”
bycount_sliding	■ リアルタイムダッシュボード

保持種類とバリアント	ユースケースの例
	■ *n*(100、200、...)最新のイベントを継続的に分析します。
bycount_jumping	■ バッチでイベントを分析または集計します。 ■ ”DB への影響を最小限に抑えるためにロード前に *n* 個のイベントを集計します。”

プライマリインデックスと特殊インデックスについて

概要

演算コードでイベントを処理するには、すべてのウィンドウにプライマリインデックスが必要です。このインデックスは、ウィンドウ内のイベントの迅速な取得、変更、または削除を可能にします。

特殊な目的を果たす他のインデックスを持つウィンドウもあります。

- ソースウィンドウとコピーウィンドウには、保持を助ける追加のインデックスがあります
- 結合ウィンドウには、左右のローカルインデックスとオプションの第 2 インデックスがあります。これらのインデックスは、ロックを回避し、データの一貫性を維持します。
- 集計ウィンドウには、グループ構造を維持するための集計インデックスがあります

表 7.2　特殊なインデックスタイプ

ウィンドウタイプ	保持インデックス	集計インデックス	左ローカルインデックス	右ローカルインデックス
ソースウィンドウ コピーウィンドウ	はい			
結合ウィンドウ			はい オプションの第 2	はい オプションの第 2

ウィンドウタイプ	保持インデックス	集計インデックス	左ローカルインデックス	右ローカルインデックス
集計ウィンドウ		はい		

完全にステートフルなプライマリインデックス

完全にステートフルなプライマリインデックスを使用するウィンドウは、一意のキーセットのカーディナリティに等しいサイズを持ちます。時間またはサイズに基づく保持ポリシーが強制されている場合は例外です。イベントは吸収され、ウィンドウのインデックスにマージされ、インデックスへの標準バージョンの変更がすべての出力ウィンドウに渡されます。

表 7.3　完全にステートフルなプライマリインデックスタイプの比較

プライマリインデックスタイプ	アルゴリズム	ストレージ	利点	欠点
pi_RBTREE	赤黒木	インメモリ	順序付けられたデータとメモリ管理により、スムーズな待機時間が実現します。	ハッシュよりも遅くなります。
pi_HASH	オープンハッシュ	インメモリ	pi_RBTREE より速い結果を提供	サイズが適切でない場合は、待機時間が急増する可能性があります。
pi_HLEVELDB	B 木	ローカルディスク	ビッグデータ ■ 大きなソースまたはコピーウィンドウ、および結合の左または右のローカルディメンションインデックスに使用されるインデックス ■ 保持または集計がない場合や、セカンダリインデックス	I/O パフォーマンス **注:** クエリ、プロジェクト、または任意のウィンドウの名前が MBCS で書かれている pi_HLEVELDB は使用しないでください。

プライマリインデックスタイプ	アルゴリズム	ストレージ	利点	欠点
			が必要な場合に使用できます。	
pi_HLEVELDB_NC	B 木	ローカルディスク	ビッグデータストレージに pi_HLEVELDB と同じ利点があり、再起動後の持続性が保たれます。	I/O パフォーマンス

保持ポリシーが指定されていない場合、完全にステートフルなインデックスの 1 つを使用するウィンドウは、データベーステーブルまたはマテリアライズドビューのように動作します。どの時点でも、イベントログの標準バージョンが含まれています。一般的なイベントはプロジェクトのウィンドウ間で参照カウントされるため、保持されているすべてのイベントが物理メモリを超えないように注意する必要があります。

公開されたイベントの更新と削除の演算コードを使用します(注文の作成、変更、終了などのライフサイクルを持つ資本市場の注文の場合と同様)。ただし、挿入専用のイベントの場合は、ウィンドウ保持ポリシーを使用して、使用可能な物理メモリの量より少ないイベントセットを保持する必要があります。

pi_HLEVELDB および pi_HLEVELDB_NC プライマリインデックスの使用

概要

pi_HLEVELDB および pi_HLEVELDB_NC プライマリインデックスはディスクに値を保存し、最近使用した(MRU)インメモリキャッシュを維持します。セカンダリインデックスの保持、集計、または必要性がない場合は、これらのインデックスタイプを使用できます。

注: クエリ、プロジェクト、または任意のウィンドウの名前が MBCS の変数幅エンコーディングで書かれている場合は、pi_HLEVELDB または pi_HLEVELDB_NC は使用しないでください。

pi_HLEVELDB を使用する場合、何百万ものイベントを 1 つのウィンドウにストリーミングしても、数ギガバイトの実メモリしか消費しません。ただし、モデルをロードする(または、ESP サーバーを再起動する)たびに、ディスク上のインデックスは消去されます。長寿命のサーバー、または再起動時にインデックスを再構築する機能を

持つサーバーの場合、これはシステムリソースの最適な使用方法ではない可能性があります。その場合は、代わりに pi_HLEVELDB_NC インデックスを使用します。そのインデックスは実質的に、初期化時にディスク上の場所を消去しない pi_HLEVELDB インデックスです。

注: ESP サーバーの再起動時に、次を実行するウィンドウではディスク上のインデックスがリロードされます。pi_HLEVELDB_NC に送信されますが、イベントは下位ウィンドウに送信されません。

再起動後の持続性が保たれる大規模ルックアップテーブルの使用

次の例は、効率的な no-regenerate ルックアップ結合を有効にする方法を示しています。ソースウィンドウはどちらもステートレスです。つまり、pi_EMPTY のインデックスがあります。

```
<window-source name="stream_source" index="pi_EMPTY" insert-only='true'>
        <schema>
          <fields>
            <field name='ID' type='int64' key='true'/>
            <field name='matchID' type='int64'/>
          </fields>
        </schema>
        <connectors>
          <connector class='fs' name='pub'>
            <properties>
              <property name='type'>pub</property>
              <property name='fstype'>csv</property>
              <property name='fsname'>input/stream.csv</property>
              <property name='transactional'>true</property>
              <property name='blocksize'>1</property>
              <property name="dateformat">%Y-%m-%d %H:%M:%S</property>
            </properties>
          </connector>
        </connectors>
      </window-source>

<window-source name='lookup_source' index='pi_EMPTY'>
        <schema>
          <fields>
            <field name='ID' type='int64' key='true' />
            <field name='Lookup' type='string' />
          </fields>
        </schema>
        <connectors>
          <connector class='fs' name='pub'>
            <properties>
              <property name='type'>pub</property>
              <property name='fstype'>csv</property>
              <property name='fsname'>input/500m.csv</property>
              <property name='transactional'>true</property>
```

```
            <property name='blocksize'>64</property>
            <property name="dateformat">%Y-%m-%d %H:%M:%S</property>
          </properties>
        </connector>
      </connectors>
</window-source>
```

注:

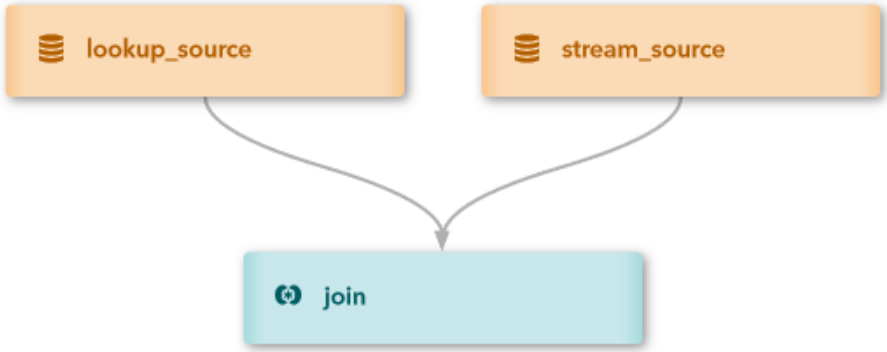
- ルックアップテーブルは HyperLevel DB のディスクに保存されています
- データの単一コピーがモデル内に存在します。処理中はインメモリキャッシュのみが参照されます。
- 結合(Join)はサーバーの再起動に対して回復力があります。つまり、システムがバウンスした場合にルックアップテーブルをリロードする必要はありません。

結合ウィンドウ自体もステートレスです。結合(Join)の右インデックスには pi_HLEVELDB のインデックスがあります。このモデルはルックアップテーブルの初期ロードを実行するため、古いルックアップデータを完全に消去する必要があります。したがって、pi_HLEVELDB_NC ではなく、pi_HLEVELDB を使用します。

```
<window-join name="join" index='pi_EMPTY'>
        <join type="leftouter" no-regenerates='true' right-index='pi_HLEVELDB'>
          <conditions>
            <fields left="matchID" right="ID" />
          </conditions>
        </join>
        <output>
          <field-selection name="matchID" source="l_matchID" />
          <field-selection name="lookup" source="r_Lookup" />
        </output>
        <connectors>
          <connector class='fs' name='sub'>
            <properties>
              <property name='type'>sub</property>
              <property name='fstype'>csv</property>
              <property name='fsname'>join-out.csv</property>
              <property name='snapshot'>true</property>
              <property name="dateformat">%Y-%m-%d %H:%M:%S</property>
            </properties>
          </connector>
        </connectors>
      </window-join>
```

エッジをウィンドウに接続すると次のようになります。

図 7.2　連続クエリ



次の形式のルックアップデータが 5 億行あるとします。

```
i,n,0, lookup string #0
i,n,1, lookup string #1
i,n,2, lookup string #2
.
.
.
i,n,499999997, lookup string #499999997
i,n,499999998, lookup string #499999998
i,n,499999999, lookup string #499999999
```

このモデルを実行すると、何百万行ものデータがロードされ、実行に数分かかることがあります。ルックアップデータがロードされたら、次のストリーミングデータを使用してルックアップをテストします。

```
i,n,1,1
i,n,2,2
i,n,3,9999999
```

これらの挿入イベントを実行すると、次のような結果になります。

```
<event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>1</value>
    <value name='lookup'>lookup string #1</value>
    <value name='matchID'>1</value>
 </event>

 <event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>2</value>
    <value name='lookup'>lookup string #2</value>
    <value name='matchID'>2</value>
 </event>

 <event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>3</value>
    <value name='lookup'>lookup string #9999999</value>
    <value name='matchID'>9999999</value>
 </event>
```

プロジェクトを停止したとします。以前のイベントと同じ新しいイベントを実行しますが、結合のルックアップインデックス内で pi_HLEVELDB_NC を使用します。

```
<join type="leftouter" no-regenerates='true' right-index='pi_HLEVELDB_NC'>
```

ルックアップ側のメンテナンスイベントをいくつか処理します。

```
p,n,14, NEW lookup string #14
u,n,499999999, NEW lookup string #499,999,999
p,n,  4999999, NEW lookup string #4,999,999
d,n, 99999999
```

その後、挿入イベントをいくつか処理して、ルックアップが正しく実行されていることを確認します。

```
i,n,1,1
i,n,2,2
i,n,3,  4999999
i,n,4, 99999999
i,n,5,14
i,n,6,  3333333
i,n,7,499999999
```

新しく結合した結果は次のとおりです。

```
<event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>1</value>
    <value name='lookup'>lookup string #1</value>
    <value name='matchID'>1</value>
 </event>

 <event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>2</value>
    <value name='lookup'>lookup string #2</value>
    <value name='matchID'>2</value>
 </event>

 <event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>3</value>
    <value name='lookup'>NEW lookup string #4</value>
    <value name='matchID'>4999999</value>
 </event>

 <event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>4</value>
    <value name='matchID'>99999999</value>
 </event>

 <event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>5</value>
    <value name='lookup'>NEW lookup string #14</value>
    <value name='matchID'>14</value>
 </event>

 <event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>6</value>
    <value name='lookup'>lookup string #3333333</value>
    <value name='matchID'>3333333</value>
 </event>

 <event opcode='insert' window='pr_01/cq_01/join'>
    <value name='ID'>7</value>
    <value name='lookup'>NEW lookup string #499</value>
    <value name='matchID'>499999999</value>
 </event>
```

ステートフルでないプライマリインデックス

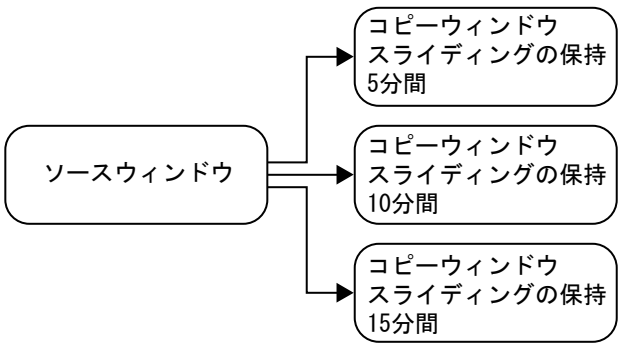
プライマリインデックスタイプ pi_EMPTY を使用するウィンドウは、ステートフルでないかステートレスです。すべての受信イベントのパススルーとして機能します。このインデックスはイベントを格納しません。

空のインデックスタイプを使用するソースウィンドウには、次の制限が適用されます。

- ソースウィンドウが"挿入のみ"に設定されている場合、制限は適用されません。
- ソースウィンドウが挿入専用でない場合は、次のいずれかを続行する必要があります。
  - ステートフルインデックスを持つコピーウィンドウ
  - 関数ウィンドウまたは計算ウィンドウ
- pi_EMPTY インデックスを持つ線形連鎖内のソースウィンドウ、計算ウィンドウ、または関数ウィンドウがモデルを開始するとき、次のいずれかが線形連鎖について True でなければなりません。
  - イベントを挿入のみに変換し、produces-only-inserts プロパティを設定する関数ウィンドウで終了する必要があります。
  - アップサートを挿入に変換するステートフルな計算、関数、またはコピーウィンドウで終了する必要があります。あるいは、ステートフルなインデックスを介して自動的にモデル内をさらに伝播する更新が必要です。

空のインデックスと保持を使用すると、ソースウィンドウから入って来る一般的なイベントストリームから複数の保持ポリシーを指定できます。次の図に示すように、ソースウィンドウは吸収ポイントとして使用され、コピーウィンドウにパススルーされます。

図 7.3　コピーウィンドウ

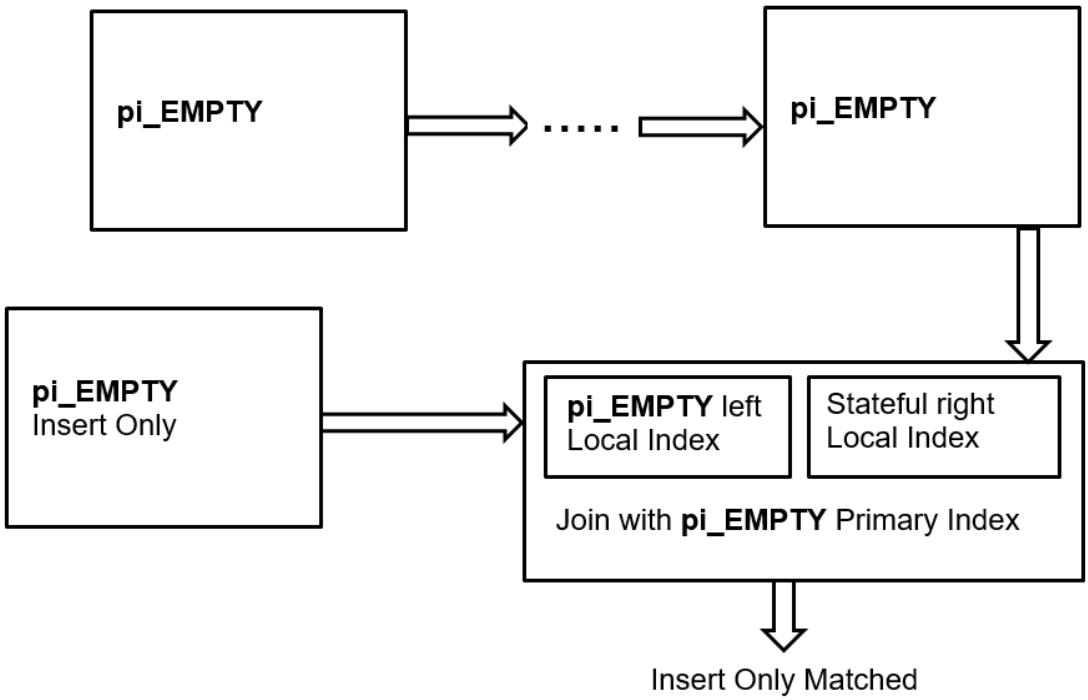


ステートフルなローカル結合インデックスを使用して状態を解決する

過去においては、挿入、更新および削除イベントのパブリッシュによるルックアップテーブルのメンテナンスが必要なストリーミング結合ルックアップでは、大量のメモリが必要でした。このルックアップでは、結合のルックアップ側にステートフルなソースウィンドウが必要です。ステートフルなソースウィンドウでは、挿入、更新および削除イベントが完全に解決され、それらが結合にフィードされます。挿入、更新および削除イベントは、結合のローカルルックアップインデックスに適用されます。デュアルステートフルインデックスメンテナンス(ソースウィンドウと結合ウィンドウのローカルルックアップインデックス)のために、かなりの量のメモリが無駄になっていました。

SAS Event Stream Processing 5.2 から、結合のルックアップ側のローカルステートフルインデックスでは、挿入、更新、アップサート、削除イベントを解決できます。これにより、結合のルックアップ側にフィードするウィンドウを pi_EMPTY にすることができますが、ルックアップメンテナンスのための挿入、更新、アップサート、削除イベントも含まれたままです。

図 7.4　左外部結合のためのステートフルインデックスの使用



次のコードは、メモリ不足の結合を示しています。

注: 制限のあるメモリ占有領域を確保するには、有限数のイベントを管理する必要があります。結合のローカルルックアップインデックスでは、結合のルックアップ側

にストリーミングされるすべてのイベント(削除されたイベントを除く)が管理されます。

ソースウィンドウは次のとおりです。

```
<project name='pr_01' pubsub='auto' threads='3' disk-store-path='./'>
  <contqueries>
    <contquery name='cq_01'>
      <windows>
        <window-source name="stream_source" index="pi_EMPTY" insert-only='true'>
          <schema>
            <fields>
              <field name='ID' type='int64' key='true'/>
              <field name='matchID' type='int64'/>
            </fields>
          </schema>
          <connectors>
            <connector class='fs' name='pub'>
              <properties>
                <property name='type'>pub</property>
                <property name='fstype'>csv</property>
                <property name='fsname'>input/stream.csv</property>
                <property name='transactional'>true</property>
                <property name='blocksize'>1</property>
                <property name="dateformat">%Y-%m-%d %H:%M:%S</property>
              </properties>
            </connector>
            <connector class='fs' name='pub1'>
              <properties>
                <property name='type'>pub</property>
                <property name='fstype'>csv</property>
                <property name='fsname'>input/stream-1.csv</property>
                <property name='transactional'>true</property>
                <property name='blocksize'>1</property>
                <property name="dateformat">%Y-%m-%d %H:%M:%S</property>
              </properties>
            </connector>
          </connectors>
        </window-source>

        <window-source name='lookup_source' index='pi_EMPTY'>
          <schema>
            <fields>
              <field name='ID' type='int64' key='true' />
              <field name='Lookup' type='string' />
            </fields>
          </schema>
          <connectors>
            <connector class='fs' name='pub'>
              <properties>
                <property name='type'>pub</property>
                <property name='fstype'>csv</property>
                <property name='fsname'>input/10.csv</property>
                <property name='transactional'>true</property>
                <property name='blocksize'>64</property>
                <property name="dateformat">%Y-%m-%d %H:%M:%S</property>
```

```
            </properties>
          </connector>
          <connector class='fs' name='pub1'>
            <properties>
              <property name='type'>pub</property>
              <property name='fstype'>csv</property>
              <property name='fsname'>input/10-update.csv</property>
              <property name='transactional'>true</property>
              <property name='blocksize'>64</property>
              <property name="dateformat">%Y-%m-%d %H:%M:%S</property>
            </properties>
          </connector>
        </connectors>
      </window-source>
```

結合ウィンドウは次のとおりです。

```
    <window-join name="join" index='pi_EMPTY'>
        <join type="leftouter" no-regenerates='true' right-index='pi_HASH'>
          <conditions>
            <fields left="matchID" right="ID" />
          </conditions>
        </join>
        <output>
          <field-selection name="matchID" source="l_matchID" />
          <field-selection name="lookup" source="r_Lookup" />
        </output>
        <connectors>
          <connector class='fs' name='sub'>
            <properties>
              <property name='type'>sub</property>
              <property name='fstype'>csv</property>
              <property name='fsname'>join-out.csv</property>
              <property name='snapshot'>true</property>
              <property name="dateformat">%Y-%m-%d %H:%M:%S</property>
            </properties>
          </connector>
        </connectors>
      </window-join>
    </windows>
    <edges>
      <edge source="lookup_source" target="join" role="right" />
      <edge source="stream_source" target="join" role="left"  />
    </edges>
  </contquery>
</contqueries>
```

コネクタグループとエッジは次のとおりです。

```
<project-connectors>
  <connector-groups>
    <connector-group name="group1">
      <connector-entry connector='cq_01/join/sub' state='running'/>
      <connector-entry connector='cq_01/lookup_source/pub' state='finished'/>
    </connector-group>
    <connector-group name="group2">
      <connector-entry connector='cq_01/stream_source/pub' state='finished'/>
    </connector-group>
```

```
      <connector-group name="group3">
        <connector-entry connector='cq_01/lookup_source/pub1' state='finished'/>
      </connector-group>
      <connector-group name="group4">
        <connector-entry connector='cq_01/stream_source/pub1' state='finished'/>
      </connector-group>
    </connector-groups>
    <edges>
      <edge source='group1' target='group2'/>
      <edge source='group2' target='group3'/>
      <edge source='group3' target='group4'/>
    </edges>
  </project-connectors>
</project>
```

4 つの入力データファイルは、次の方法でオーケストレーションされます。

1 最初のファイルには、結合のルックアップ側にフィードされるデータが含まれています。

```
i,n,0, lookup string #0
i,n,1, lookup string #1
i,n,2, lookup string #2
i,n,3, lookup string #3
i,n,4, lookup string #4
i,n,5, lookup string #5
i,n,6, lookup string #6
i,n,7, lookup string #7
i,n,8, lookup string #8
i,n,9, lookup string #9
```

2 2 番目のファイルには、結合のストリーミング側にフィードされるストリーミングデータが含まれ、結合出力が生成されます。

```
i,n,1,1
i,n,2,2
i,n,3,9
```

結合の初期出力を次に示します。

```
I,N, 1,1,lookup string #1
I,N, 2,2,lookup string #2
I,N, 3,9,lookup string #9
```

3 3 番目のファイルには、結合のルックアップ側にフィードされる 2 番目のデータセットが含まれています。結合メンテナンスが発生します。ルックアップテーブルでは、挿入、更新、アップサート、削除が実行されます。

```
u,n,0, NEW lookup string #0
d,n,5,
p,n,9, NEW lookup string #9
d,n,17
```

4 4 番目のファイルには、結合のストリーミング側にフィードされる 2 番目のストリーミングデータセットが含まれています。ルックアップでは、以前に実行された結合メンテナンスが反映されます。

```
i,n,1,1
i,n,2,2
i,n,3,9
i,n,4,5
```

```
i,n,5,0
```

ルックアップテーブルのメンテナンスが適用された後の結合の出力を次に示します。

```
I,N, 1,1,lookup string #1
I,N, 2,2,lookup string #2
I,N, 3,9,NEW lookup string #9
I,N, 4,5,
I,N, 5,0,NEW lookup string #0
```

ウィンドウのプライマリインデックスと入力ウィンドウの制限

ウィンドウのプライマリインデックスまたはその入力ウィンドウで次の制限に違反すると、ESP サーバーはログに記録されている致命的なエラーを返します。その結果、モデルは開始できません。これにより、不適切なモデルがデータを処理して実行時の問題を引き起こす前に、フラグが立てられます。

注: ウィンドウにステートフルなインデックスがあり、挿入のみを受け取る場合、ESP サーバーはメモリが無限に増加するという致命的なエラーを返すことがあります。このような場合、モデルは開始できません。

表 7.4　ウィンドウのプライマリインデックスと入力ウィンドウの制限

ウィンドウ	インデックス制限	入力ウィンドウ制限	演算コード出力
集計	ステートフルインデックスのみを使用	入力ウィンドウは pi_EMPTY インデックスを持つことができず、非挿入を生成	すべて
算出	なし	入力ウィンドウは pi_EMPTY インデックスを持つことができず、非挿入を生成	algorithm='MAS の場合、produces-only-inserts ='true'を設定します。それ以外の場合は、すべての入力ウィンドウが挿入のみ生成する場合は挿入のみを生成
計算	なし	なし	入力ウィンドウが挿入のみ生成する場合は挿入のみを生成

ウィンドウ	インデックス制限	入力ウィンドウ制限	演算コード出力
コピー	ステートフルインデックスのみを使用してください。これには保持を設定する必要があります。ノートを参照してください。	なし	すべて
カウンター	なし	なし	インデックスがpi_EMPTY の場合は挿入のみを生成
フィルター	なし	入力ウィンドウはpi_EMPTY になることができず、非挿入を生成	入力ウィンドウが挿入のみ生成する場合は挿入のみを生成
関数	なし	なし	すべての 入力ウィンドウが挿入のみ生成する場合は挿入のみを生成
ジオフェンス	なし	なし	イベント位置入力ウィンドウが常に挿入を生成する場合は、挿入のみが生成される
結合	なし	なし	結合(Join)のストリーミング側で挿入のみ生成され、結合(Join)が no-regenerates='true'の場合、挿入のみを生成
Lua	なし	なし	入力ウィンドウが挿入のみ生成する場合は挿入のみを生成。produces-only-inserts='false'を設定することで、このデフォルトの動作をオーバーライドできます。これにより、入力ウィンドウが挿入のみを生成する場合でも、Lua ウィンドウは非挿入のオペコ

ウィンドウ	インデックス制限	入力ウィンドウ制限	演算コード出力
			ードを生成できるようになります。
モデルリーダー	pi_EMPTY を排他的に使用	すべての入力ウィンドウは挿入のみを生成しなければならない	常に挿入を生成
モデルスーパーバイザー	pi_EMPTY を排他的に使用	すべての入力ウィンドウは挿入のみを生成しなければならない	常に挿入を生成
オブジェクトトラッキング	pi_EMPTY を排他的に使用	すべての入力ウィンドウは挿入のみを生成しなければならない	常に挿入を生成
パターン	pi_EMPTY を排他的に使用	挿入のみ受信し、非挿入の警告をログに記録	常に挿入を生成
プロシジャ	なし	なし	適宜 produces-only-inserts='true' を設定します。これは、ウィンドウ内で実行されるプロシジャコードによって異なります。
Python	なし	なし	入力ウィンドウが挿入のみ生成する場合は挿入のみを生成。produces-only-inserts='false'を設定することで、このデフォルトの動作をオーバーライドできます。これにより、入力ウィンドウが挿入のみを生成する場合でも、Python ウィンドウは非挿入のオペコードを生成できるようになります。
状態の削除	pi_EMPTY を排他的に使用	なし	常に挿入を生成

ウィンドウ	インデックス制限	入力ウィンドウ制限	演算コード出力
スコア	pi_EMPTY を排他的に使用	すべての入力ウィンドウは挿入のみを生成しなければならない	常に挿入を生成
ソース	なし	ソースウィンドウへの入力ウィンドウはありません。	挿入専用に設定する場合は挿入のみを生成
StateDB リーダー	pi_EMPTY を排他的に使用	すべての入力ウィンドウは挿入のみを生成しなければならない	generate-deletes = 'false'の場合に、挿入を生成します。generate-deletes = 'true'の場合に、挿入と削除を生成します。
StateDB ライター	pi_EMPTY を排他的に使用	すべての入力ウィンドウは挿入のみを生成しなければならない	常に挿入を生成
テキストカテゴリ	pi_EMPTY を排他的に使用	すべての入力ウィンドウは挿入のみを生成しなければならない	常に挿入を生成
テキストコンテキスト	pi_EMPTY を排他的に使用	すべての入力ウィンドウは挿入のみを生成しなければならない	常に挿入を生成
テキストセンチメント	pi_EMPTY を排他的に使用	すべての入力ウィンドウは挿入のみを生成しなければならない	常に挿入を生成
テキストトピック	pi_EMPTY を排他的に使用	すべての入力ウィンドウは挿入のみを生成しなければならない	常に挿入を生成
学習	pi_EMPTY を排他的に使用	すべての入力ウィンドウは挿入のみを生成しなければならない	常に挿入を生成

ウィンドウ	インデックス制限	入力ウィンドウ制限	演算コード出力
転置	pi_EMPTY を排他的に使用	すべての入力ウィンドウは挿入のみを生成しなければならない	常に挿入を生成
和結合(Union)	入力ウィンドウとストリクト設定に依存	インデックスタイプに依存	いずれかの入力ウィンドウで未解決の更新または削除が発生した場合、そのインデックスはステートフルでなければなりません。すべての入力ウィンドウが常に挿入を生成し、和結合(Union)がストリクトである場合、挿入のみ生成されます。インデックスは pi_EMPTY です。

注: 挿入のみを受け取り、スプリッター式を使用するコピーウィンドウは、pi_EMPTY インデックスを使用できます。

デザインパターンについて

デザインパターンの概要

デザインパターンは、ソフトウェア設計の特定のコンテキスト内の共通の問題に対する再利用可能なソリューションです。デザインパターンで使用するウィンドウの組み合わせによって、迅速かつ効率的なイベントストリーム処理が可能になります。

イベントストリーム処理モデルは、ステートレス、ステートフル、またはミックスすることができます。選択したモデルのタイプは、その設計方法に影響します。混合モデルを設計するときの課題の 1 つは、ステートフルでなければならないセクションとステートレスにするセクションを特定し、それらを適切に結合することです。

ステートレスモデルは、すべてのウィンドウのインデックスが **pi_EMPTY** 型のタイプです。イベントはどのウィンドウにも保持されず、本質的に変換されて渡されます。ステートレスモデルは高速なパフォーマンスを示し、メモリをほとんど使用しません。入力が挿入であり、単純なフィルタリング、計算、テキストコンテキスト解析、

またはパターンマッチングだけが必要な操作である場合は、これらはタスクに適しています。

ステートフルモデルは、データを格納するインデックスタイプ(通常は **pi_RBTREE** または **pi_HASH**)のウィンドウを使用するモデルです。これらのモデルは、演算コードの挿入、更新、または削除によってイベントを完全に処理できます。ステートフルモデルは、結合や集計などの複雑な関係演算を容易にします。イベントはインデックスに保持されるため、すべてのイベントが挿入のみの場合、ウィンドウはメモリ内で無限に拡大します。したがって、ステートフルモデルでは、メモリに制限された状態を維持するために、挿入、更新、および削除の組み合わせを処理する必要があります。

演算コードの組み合わせは、次の 2 つの方法のいずれかで発生します。

- データソースと入力イベントには、キーのカーディナリティがあります。つまり、入力ストリームに固定数の一意のキー(顧客 ID など)があります。キーのカーディナリティが有限であれば、これらのキーを多く更新することができます。
- 保持ポリシーは、データ量が TIME またはイベント数によって制限されるデータ流入に対して強制されます。内部保持削除イベントの生成によって、データは自動的にシステムから削除されます。

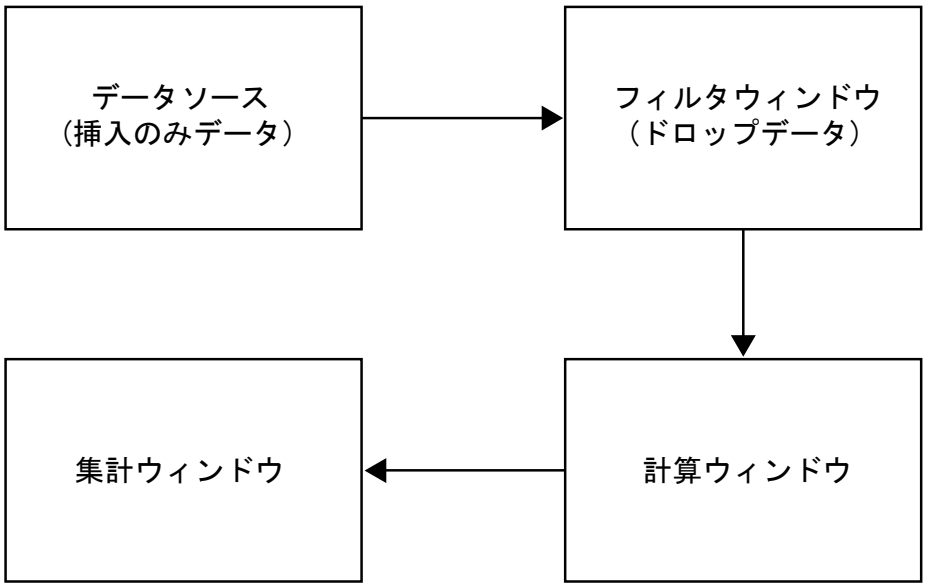
混合モデルには、ステートレスでステートフルな部分があります。パーツをステートレスなフロントエンドとステートフルなバックエンドに分けることが可能なことがよくあります。

ステートレスモデルとステートフルモデルをリンクするデザインパターン

混在したモデルでメモリの増加を抑制するには、ステートレスとステートフルのパーツを保持ポリシーを適用するコピーウィンドウにリンクします。このデザインパターンは、ステートレスな方法で事前処理できる挿入専用データがある場合に使用します。ステートフルな処理が必要なモデルのセクション(結合、集計、またはその両方を使用)にデータを流す前に、データを前処理します。

たとえば、次のモデルを考えてみましょう。

図 7.5　挿入専用データによるイベントストリーム処理モデル



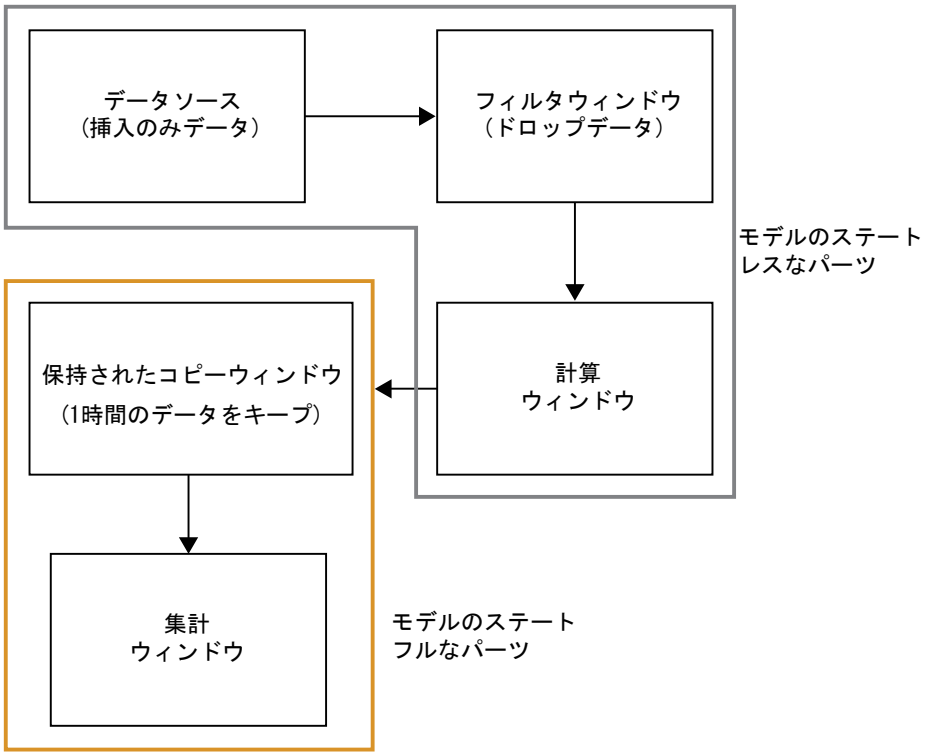
ここでのデータソースは純粋に挿入によるものです。したがって、インデックスタイプ **pi_EMPTY** を使用すると、モデルをステートレスにすることができます。フィルターはソースからの挿入を受け取り、フィルター基準に基づいてイベントのいくつかをドロップするため、出力として挿入セットが生成されます。したがって、インデックスタイプ **pi_EMPTY** を使用することによっても、フィルターをステートレスにすることができます。

計算ウィンドウは、出力フィールドのいくつかを選択することによって、入力イベントの値を変換します。入力イベントの値に基づいて他のフィールドの値を計算します。それは挿入だけを生成するので、ステートレスにすることができます。

計算ウィンドウの後には、集計ウィンドウがあります。このウィンドウタイプはイベントを保持する必要があります。集計ウィンドウはデータをグループ化し、グループを単一のイベントに圧縮します。集計ウィンドウに挿入ストリームが供給されると、無限に拡張されます。

この成長を抑制するには、モデルの 2 つのセクションを保持ポリシーのあるコピーウィンドウに接合します。

図 7.6　ステートレスおよびステートフルなパーツを含む修正されたイベントストリーム処理モデル



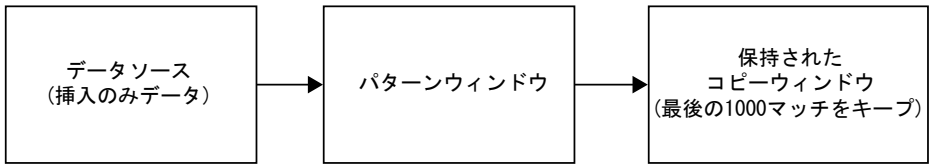
入力データのローリングウィンドウに基づいて、モデルのステートフル部分が正確に計算されます。このモデルはメモリ内に拘束されています。

パターンウィンドウの一致の制御

パターンウィンドウで生成されたパターンマッチは挿入されます。パターンウィンドウにソースウィンドウが表示されているとします。パターンウィンドウは挿入のみを生成するため、**pi_EMPTY** のインデックスタイプを指定することによってステートレスにする必要があります。これにより、パターンウィンドウが無限に拡大するのが防止されます。通常、最近のパターンマッチのいくつかを保持したいとします。パターンがどのくらい頻繁に一致を生成するかわからないので、パターンウィンドウにカウントベースのコピーウィンドウを表示します。

コピーウィンドウで最後の 1000 個のパターンマッチを保持するように指定したとします。

図 7.7　保持を伴うコピーによるイベントストリーム処理モデル



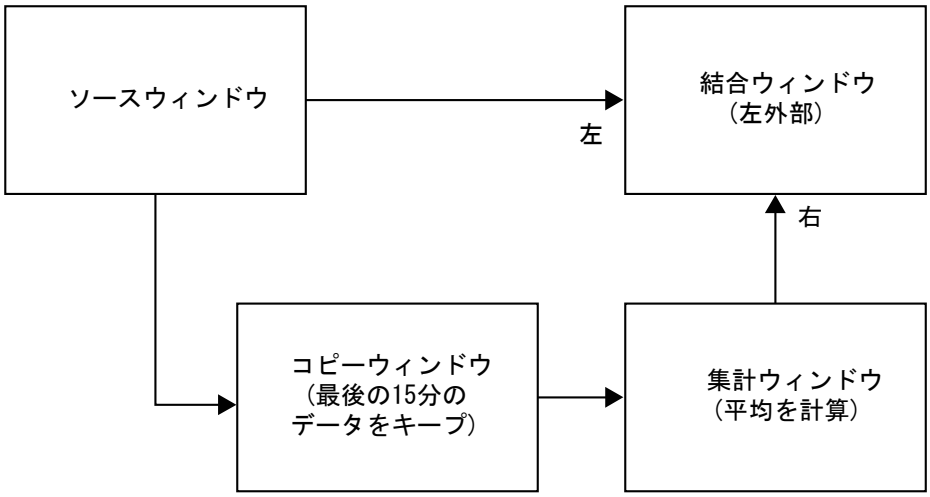
このような場合、コピーウィンドウは、アダプターを使用して外部から照会されるか、クライアントをパブリッシュ/サブスクライブする可能性が高くなります。コピーウィンドウは、モデルの他のセクションにも送ることができます。

ローリング統計による受信イベントの拡張

イベントの挿入ストリームがあり、1 つ以上の値がイベントに関連付けられているとします。いくつかのローリング統計を使用して各入力イベントを増強し、拡張イベントを生成したいとします。この問題を解決するには、モデリング環境の高度な機能を使用する必要があります。

たとえば、ストリームや株式取引が入っていて、過去に平均株価を上回るようにしたいとします。次のモデルを作成します。

図 7.8　高度な機能を使用するイベントストリーム処理モデル



集計ウィンドウを制御するには次にようにします。

- その前に保持を入れてください(コピーウィンドウ)。
- それをシンボル(バインドされたもの)でグループ化し、各イベントをその計算のために格納する必要のない加算的集計関数平均(ESP_aAve)を使用します。

結合ウィンドウに問題がある可能性があります。通常、結合ウィンドウはステートフルと考えることができます。結合は、ファクトウィンドウまたはディメンションウィンドウのデータを保持し、マッチを実行します。この場合、特別であるものの、頻繁に発生する動作が必要です。入力が入ると、入力に対応する集計が処理されるまで結合時に一時停止します。次に、2 つをリンクさせて、拡張された挿入物を渡します。

この方法で入力を処理するには次のようにします。

1 結合を左外部結合にし、ソースは左ウィンドウをフィードし、集計は右ウィンドウをフィードします。

2 プロジェクトのタグ付きトークンデータフローモデルを設定します。これは、出力イベントを生成する前に、フォークの両側が再結合するのを待つ単一のイベントのフォークを発生させる特別な機能をオンにします。

3 結合のインデックスタイプを **pi_EMPTY** に設定し、結合をステートレスにします。ステートレスな左外部結合は、左の駆動ウィンドウ(ファクトウィンドウ)のローカルコピーを作成しません。結合の格納された結果は保持されません。ただし、常に参照カウントのコピーがルックアップウィンドウに表示されます。左外部結合の場合、これは右ウィンドウです。ルックアップウィンドウは、この場合の保持によって制御されるため、境界があります。

4 通常、ディメンションウィンドウが変更されると、結合がファクトウィンドウ内の一致するすべてのイベントを検索し、結合されたマッチの更新を発行しようとします。ロックステップでイベントを照合しているため、この動作は必要ありません。さらに、ファクトデータを格納しないため、単純に不可能です。ディメンションウィンドウの変更ごとにこの再生成を防止するには、結合ウィンドウで **no-regenerates** オプションを設定します。

このようにして、高速で軽量な結合を作成します。この結合では、検索側のみが格納され、挿入されたファクトイベントごとに挿入ストリームが生成されます。左と右の外部結合では、ステートレス結合が可能です。

次の XML コードはこのモデルを実装しています。

```
<project name='trades_proj' pubsub='auto'
      use-tagged-token='true' threads='4'>
        <contqueries>
          <contquery name='trades_cq'>

            <windows>
              <window-source name='Trades'
                        insert-only='true'
                        index='pi_EMPTY'>
                <schema>
                  <fields>
                    <field name='tradeID' type='string' key='true'/>
                    <field name='security' type='string'/>
                    <field name='quantity' type='int32'/>
                    <field name='price' type='double'/>
                    <field name='traderID' type='int64'/>
                    <field name='time' type='stamp'/>
                  </fields>
                </schema>
              </window-source>

              <window-copy name='TotalIn'>
                <retention type='bytime_sliding'>15{minutes}</retention>
              </window-copy>

              <window-aggregate name='RunTotal'>
                <schema>
                  <fields>
                    <field name='tradeID' type='string'/>
                    <field name='security' type='string' key='true'/>
                    <field name='quantityTotal' type='double'/>
                  </fields>
                </schema>
```

```
            <output>
              <field-expr>ESP_aLast(tradeID)</field-expr>
              <field-expr>ESP_aSum(quantity)</field-expr>
            </output>
          </window-aggregate>

          <window-join name='JoinTotal'
                   index='pi_RBTREE'>
            <join type="leftouter"
                 no-regenerates='true'>
              <conditions>
                <fields left='tradeID' right='tradeID'/>
                <fields left='security' right='security'/>
              </conditions>
            </join>
            <output>
              <field-selection name='quantity'
                source='l_quantity'/>
              <field-selection name='price'
                source='l_price'/>
              <field-selection name='traderID'
                source='l_traderID'/>
              <field-selection name='time'
                source='l_time'/>
              <field-selection name='quantityTotal'
                source='r_quantityTotal'/>
            </output>
          </window-join>
        </windows>

        <edges>
          <edge source='Trades'
              target='JoinTotal'/>
          <edge source='Trades'
              target='TotalIn'/>
          <edge source='TotalIn'
              target='RunTotal'/>
          <edge source='RunTotal'
              target='JoinTotal'/>
        </edges>

      </contquery>
    </contqueries>
</project>
```

高度なウィンドウ操作

アプリケーションに埋め込むための集計関数の作成

概要

0、1、2 または 3 つの引数で集計関数を書きます。引数は、集計の入力スキーマの整数値式、整数定数、またはフィールド名のいずれかでなければなりません。

最も一般的に使用される集計関数は、入力スキーマフィールド名を持つ 1 つのパラメーター関数です(たとえば、集計関数 ESP_aMax(*fieldname*)です)。入力スキーマのフィールド名の場合、入力イベントへのフィールド索引は、フィールドの値ではなく集計関数に渡されます。これは、グループを扱うときに重要です。グループ内のすべてのイベントを反復処理し、各入力イベントからイベントインデックスで値を抽出する必要があります。

集計関数を記述したら、それを C++コードに埋め込み、イベントストリームプロセッサアプリケーションで使用します。関数の名前が **My_Aggregation_Function** であるとします。**functions.cpp** の最後に、集計関数用のラッパー関数を作成します。

```
// the uMyFunction wrapper:
   // every aggregation function must be wrapped like this.
   //
   int dfESPaggrfunc_uMyFunctionWrapper(dfESPexpEngine::exp_engine_t *e,
                                        dfESPexpEngine::exp_sym_value_t *returnval,
                                        int parmcount,
                                        dfESPexpEngine::exp_sym_value_t **parms)
                                        {
      return dfESPaggrfunc_Wrapper((void *)my_aggrergation_function, e,
                                        returnval, parmcount, parms);
                                        }
```

ラッパー関数のユーザー定義関数リストに項目を作成します。

```
// Get all user-defined aggregation functions during initialization
   //
   void add_user_aggrFunctions() {
      dfESPengine *e = dfESPengine::getEngine();

      dfESPptrList<aggr_function_t *> &uFuncts = e->getUDAFs();

      // push back as many user defined functions as you like:
      //   the paramters are: <callable name>, <function pointer>,
      //   <arg type vect>, <additive flag>,  <additive flag for insert only>,
      //   <description>
      uFuncts.push_back(new aggr_function_t("USER_uSum_nadd",
```

```
    (void *)dfESPaggrfunc_uSumNAddWrapper, "a", false,
     false, "summarization"));
  uFuncts.push_back(new aggr_function_t("USER_uSum_add",
    (void *)dfESPaggrfunc_uSumAddWrapper, "a", true,
     true, "summarization"));
```

引数の数、追加フラグ、および説明フィールドをそれに応じて調整します。

加算的集計関数の作成

以前のフィールド状態と新規作成フィールド値に基づいて計算する集計関数は、加算集計関数と呼ばれます。これらの関数は、集計関数よりも計算上の利点があります。

加算的な集計関数は、2 つの理由で複雑になる可能性があります。

- 現在の状態(たとえば、最後に計算された状態)を調べる必要があります。
- 適切な調整を行うために、受信イベントの種類を評価する必要があります。

グループのフィールドの合計の最終値の状態を保持しているとします。新規作成イベントが到着すると、受信イベントが挿入、削除、または更新のいずれであるかを状態ベースで条件付きで調整できます。挿入イベントの場合は、単に新規作成値で状態を増やします。削除の場合は、削除した値で状態を減らします。更新の場合は、それぞれ新規作成値と古い値で増減します。現在、この関数はすべてのグループ値をループする必要はありません。以前の状態とそのグループに影響を与える最新のイベントに基づいて新規作成合計を判断できます。

次のコードは、これらの基本的な手順を実行します。

1. 入力と出力の種類を調べ、互換性を確認してください。
2. 指定された出力種類の戻り変数を初期化します。
3. その関数が以前に呼び出されたかどうかを判定します。つまり、以前の状態値の有無です。
   - そうであれば、それを検索して使用してください。
   - そうでない場合は、到着する値で状態を設定できるように、到着する挿入物で新規作成グループを作成します。
4. 演算コードをオンにして、状態値を調整します。
5. 計算上の ERROR をチェックし、エラー値または状態値を結果として返します。

```
// an additive summation function
  //
  // vgs is the groupstate object passed as a (void *) pointer
  // fid is the filed ID in internal field order of the field on
  //    which we sum.
  dfESPdatavarPtr uSum_add(void *vgs, dfESPexpEngine::exp_sym_value_t *fid) {

    dfESPdatavar    *rdv;
    // placeholder for return value
    dfESPgroupstate *gs = (dfESPgroupstate *)vgs;
    // the passed groupstate cast back to dfESPgroupstate object.
```

```
// get the 1) aggregate schema (output schema)
// and 2) the schema of input events
//
dfESPschema  *aSchema = gs->getAggregateSchema();
dfESPschema  *iSchema = gs->getInputSchema();

// get the type of 1) the field we are computing in the aggregate schema and
// 2) the input field we are summing.
//
dfESPdatavar::dfESPdatatype aType = aSchema->getTypeEO(gs->getOperField());

bool te=false;
int64_t fID = exp_param_getI64(fid, te);
if (te) {
  cerr << "could not obtain the integer field offset (field ID)" << endl;
  rdv = new dfESPdatavar(aType); rdv->null();
  return rdv;
}

dfESPdatavar::dfESPdatatype iType = iSchema->getTypeIO(fID);

dvn_error_t   retCode = dvn_noError;
// return code for using the datavar numerics package.

// If the input fields or the output field is non-numeric,
// flag an error.
//
if ( (!isNumeric(aType)) || (!isNumeric(iType)) ) {
cerr << "summation must work on numeric input, produce numeric output."
    << endl;
return NULL;
}

// in the ESP type system, INT32 < INT64 < DOUBLE < DECSECT.
//   This checks compatibility. The output type must be greater
//   equal the input type. i.e. we cannot sum a column of int64
//   and puit them into an int32 variable.
//
if (iType > aType) {
cerr << "output type is not precise enough for input type" << endl;
return NULL;
}

// fetch the input event from the groupstate object (nev)
// and, in the case of an update, the old event that
// is being updated (oev)
//
dfESPeventPtr nev = gs->getNewEvent();
dfESPeventPtr oev = gs->getOldEvent();

// Get the new value out of the input record
//
dfESPdatavar iNdv(iType);
// a place to hold the input variable.
dfESPdatavar iOdv(iType);
```

```
// a place to hold the input variable (old in upd case).
nev->copyByIntID(fID, iNdv);
// extract input value (no copy) to it (from new record)

// Get the old value out of the input record (update)
//
if (oev) {
   oev->copyByIntID(fID, iOdv);
// extract input value to it (old record)
}

// Note: getStateVector() returns a reference to the state vector for
//       the field we are computing inside the group state object.
//
dfESPptrVect<dfESPdatavarPtr> &state = gs->getStateVector();

// create the datavar to return, of the output type and set to zero.
//
rdv = new dfESPdatavar(aType);     // NULL by default.
rdv->makeZero();

// If the state has never been set, we set it and return.
//
if (state.empty()) {
   dv_assign(rdv, &iNdv);
   //  result = input
   state.push_back(new dfESPdatavar(rdv));
   //  make a copy and push as state
   return rdv;
}

// at this point we have a state,
// so lets see how we should adjust it based on opcode.
//

dfESPeventcodes::dfESPeventopcodes opCode = nev->getOpcode();
bool  badOpcode = false;
int   c = 0;
switch (opCode)  {
case dfESPeventcodes::eo_INSERT:
   if (!iNdv.isNull())
      retCode = dv_add(state[0], state[0], &iNdv);
   break;
case dfESPeventcodes::eo_DELETE:
   if (!iNdv.isNull())
      retCode = dv_subtract(state[0], state[0], &iNdv);
   break;
case dfESPeventcodes::eo_UPDATEBLOCK:
   retCode = dv_compare(c, &iNdv, &iOdv);
   if (retCode != dvn_noError) break;
   if (c == 0)  // the field value did not change.
      break;
   if (!iNdv.isNull())
   // add in the update value
      retCode = dv_add(state[0], state[0], &iNdv);
   if (retCode != dvn_noError) break;
```

```
    if (!iOdv.isNull())
    // subtract out the old value
      retCode = dv_subtract(state[0], state[0], &iOdv);
    break;
  default:
  cerr << "got a bad opcode when running uSum_add()" << endl;
  badOpcode = true;
  }

  if ( badOpcode || (retCode != dvn_noError) ) {
    rdv->null();
    // return a null value.
  cerr << "uSum() got an arithmetic error summing up values" << endl;
  } else
    dv_assign(rdv, state[0]);
  // return the adjusted state value

  return rdv;
}
```

Sum()集計関数を使用してグループを 反復処理し、新規作成グループが変更されたときに新しい合計を計算することができます。新規作成イベントが挿入、更新、または削除である場合、dfESPdatavar オブジェクトの dfESPgroupstate に Sum()を保持し、そのオブジェクトを入力値で増分または減分すると、より高速な結果が得られます。この関数は、このフィールド状態を調整して最新の状態にし、グループの別の変更が発生したときに再び使用することができます。

非加算的集計関数の作成

状態を維持せず、加法でもない集計関数を作成できます。この関数は、グループ内の各イベントを繰り返して集計します。集計ウィンドウには、すべてのグループのすべての 入力イベントのコピーを保持する必要があります。

次のコードは、これらの基本的な手順を実行します。

1 入力と出力の種類を調べ、互換性を確認してください。

2 指定された出力種類の戻り変数を初期化します。

3 グループ内のすべてのイベントをループし 、集計関数を実行します。

4 計算上の ERROR をチェックし、ERROR または結果を返します。

```
// a non-additive summation function
  //
  // vgs is the groupstate object passed as a (void *) pointer
  // fid is the filed ID in internal field order of the field on
  //    which we sum.
  dfESPdatavarPtr uSum_nadd(void *vgs, dfESPexpEngine::exp_sym_value_t *fid) {

    dfESPdatavar    *rdv;
    // placeholder for return value
    dfESPgroupstate *gs = (dfESPgroupstate *)vgs;
    // the passed groupstate cast back to dfESPgroupstate object.
```

```
// get the 1) aggregate schema (output schema)
//     and 2) the schema of input events
//
dfESPschema  *aSchema = gs->getAggregateSchema();
dfESPschema  *iSchema = gs->getInputSchema();

// get the type of 1) the field we are computing in the aggregate schema and
// 2) the input field we are summing.
//
dfESPdatavar::dfESPdatatype aType = aSchema->getTypeEO(gs->getOperField());


bool te=false;
int64_t fID = exp_param_getI64(fid, te);
if (te) {
  cerr << "could not obtain the integer field offset (field ID)" << endl;
  rdv = new dfESPdatavar(aType); rdv->null();
  return rdv;
}

dfESPdatavar::dfESPdatatype iType = iSchema->getTypeIO(fID);
dvn_error_t   retCode = dvn_noError;
// return code for using the datavar numerics package.

// If the input fields or the output field is non-numeric,
// flag an error.
//
if ( (!isNumeric(aType)) || (!isNumeric(iType)) ) {
cerr << "summation must work on numeric input, produce numeric output."
    << endl;
return NULL;
}

// in the ESP type system, INT32 < INT64 < DOUBLE < DECSECT.
//   This checks compatibility. The output type must be greater
//   equal the input type. i.e. we cannot sum a column of int64
//   and puit them into an int32 variable.
//
if (iType > aType) {
 cerr << "output type is not precise enough for input type" << endl;
 return NULL;
}

dfESPeventPtr nev = gs->getNewEvent();
dfESPeventPtr oev = gs->getOldEvent();

// create the datavar to return, of the output type and set to zero.
//
rdv = new dfESPdatavar(aType);     // NULL by default.
rdv->makeZero();


dfESPeventPtr gEv = gs->getFirst(); // get the first event in the group.
dfESPdatavar iNdv(iType);          // a place to hold the input variable.
while (gEv) {                      // iterarate until no more events.
   gEv->copyByIntID(fID, &iNdv);   // extract value from record into iNdv;
```

```
    if (!iNdv.isNull()) {          // input not null
       if ((retCode = dv_add(rdv, rdv, &iNdv)) != dvn_noError)
          break;                  // rdv = add(rdv, iNdv)
    }
    gEv = gs->getNext();           // get the first event in the group.
}

if (retCode != dvn_noError ) {      // if any of our arithmitic fails.
 rdv->null();                       // return a null value.
 cerr << "uSum() got an arithmetic error in summing up values" << endl;
}

return rdv;
```

周期的(またはパルス)ウィンドウ出力の実装

ほとんどの場合、SAS Event Stream Processing API は完全にイベント駆動型です。つまり、ウィンドウは入力を変換するとすぐに出力を生成し続けます。しかし、ウィンドウがデータを保持し、更新の正規のバッチを書きたい場合があります。この場合、共通のキー値への操作は単一の操作に集約されます。

バッチ処理された出力が便利な場合が 2 つあります。

- ビジュアライゼーションクライアントは、1 秒に 1 回更新を取得したいと思うかもしれません。イベントデータがパルス化されると、クライアントは共通キー値のデータ圧縮を利用してデータを視覚化することができます。
- パルスウィンドウに続くウィンドウは、そのウィンドウからの周期的なスナップショット間の差分を比較することに関心があります。

次の呼び出しを使用して、出力パルスをウィンドウに追加します。

```
dfESPwindow::setPulseInterval(size_t us);
```

注: 周期性はマイクロ秒単位で指定されます。ただし、ほとんどの非リアルタイムオペレーティングシステムのクロック分解能を考慮すると、パルス期間に指定する必要がある最小値は 100 ミリ秒です。XML コードでは、ウィンドウのパルス間隔属性を使用します。値の既定値はミリ秒です。

パブリッシュ時にイベントを部分更新としてマーク

概要

ほとんどの場合、イベントはすべてのフィールドが利用可能でパブリッシュされます。フィールド値の一部はヌルになることがあります。削除演算コードを含むイベントでは、キーフィールドだけが非ヌルである必要があります。

更新されるキーフィールドとフィールドのみがイベントの更新に必要または利用可能な場合があります。これは金融フィードの典型的なものです。たとえば、ブローカーは、未処理の注文の price または quantity を更新することができます。イベントを部分更新(標準ではなく)としてマークして、選択したフィールドを更新することができます。

イベントを部分更新としてマークすると、キーフィールドと更新されるフィールドの値のみが提供されます。この場合、更新されないフィールドはデータ型 dfESPdatavar::ESP_LOOKUP としてマークされます。このマーキングで、システムに保持されているイベントのキーフィールドと現在のイベントを一致させ、現在のイベントのフィールドを更新しないように、SAS Event Stream Processing に指示します。

パブリッシュされたイベントを部分的な更新としてタグ付けするには、イベントにソースウィンドウの既存のイベントと一致するすべての非ヌルキーフィールドが含まれている必要があります。部分的な更新はソースウィンドウにのみ適用されます。

部分的なイベントを含むトランザクショナルイベントブロックを使用する場合、すべての部分的な更新はすでにソースウィンドウにあるキーフィールド用であることに注意してください。トランザクションプロパティを持つ単一のイベントブロックにキー値の更新を伴うキー値の挿入を含めることはできません。トランザクションのイベントブロックがアトミックに処理されるため、この試行は失敗し、ログに記録されます。トランザクションブロックが全体として適用される前に、そのブロック内のすべての操作が既存のウィンドウ状態と照合されます。

ソースウィンドウに部分イベントをパブリッシュ

部分的なイベントをソースウィンドウにパブリッシュするときは、この 3 つの点を考慮してください。

- 部分的なイベントを構成するには、イベントのすべてのフィールドを表す必要があります。フィールドの種類と値、またはフィールドの値と種類がないことを示すプレースホルダーフィールドを指定します。このように、このキーフィールドの組み合わせの既存のフィールド値は、更新されたイベントのために残ります。これらのフィールド値と種類は、イベントを構築する datavars として提供することができます。あるいは、コンマ区切り値(CSV)文字列として提供することもできます。

CSV 文字列を使用する場合は、'^U'(control-U、10 進値 21 など)を使用して、フィールドがプレースホルダーフィールドであり、更新しないことを指定します。一方、datavars を使用して個々のフィールドを表す場合は、完全指定フィールドが有効である必要があります。それらを値（非ヌルまたはヌル）と一緒に datavars として入力しします。プレースホルダーフィールドを型 dfESPdatavar::ESP_LOOKUP の空の datavars として指定します。

- フィールドの値と種類を表現するためにどのような形式を使用しても、部分的な更新を公開するための呼び出しに表現を含める必要があります。フィールドに加えて、レコードが標準更新か部分更新かを示すフラグを使用します。部分的な更新を指定する場合、イベントは更新または更新に解決される更新でなければなりません。部分更新フィールドの使用は、既存または保持されているソースウィンドウイベントの更新のコンテキストでのみ意味があります。このため、イベントの演算コードを更新に解決する必要があります。更新に解決されない場合、イベントマージ ERROR が生成されます。

  イベントコンストラクターを使用して CSV 文字列からこのバイナリイベントを生成すると、これが部分的な更新であることを示す"u,p"がその CSV 文字列の先頭に含まれます。代わりに、event->buildEvent()を使用してこの部分的な更新イベントを作成するには、イベントフラグパラメーターを dfESPeventcodes::ef_PARTIALUPDATE、そしてイベント演算コードパラメーターを dfESPeventcodes::eo_UPDATE として指定します。

- 1 つまたは複数のイベントがベクトルにプッシュされ、そのベクトルがイベントブロックを作成するために使用されます。イベントブロックはソースウィンドウにパブリッシュされます。パフォーマンス上の理由から、通常、各イベントブロックには複数のイベントが含まれています。イベントブロックを作成するときは、dfESPeventblock::ebt_TRANS を使用してイベントブロックの種類をトランザクションまたはアトミックとして指定するか、dfESPeventblock::ebt_NORMAL を使用して標準どおりに指定する必要があります。

  部分的な更新でトランザクションブロックを使用しないでください。そのような使用法は、イベントブロック内のすべてのイベントをアトミックとして 扱います。イベントの元の挿入が部分的な更新と同じイベントブロックにある場合は失敗します。イベントブロック内のイベントは、イベントブロックがアトミックに適用される前にウィンドウインデックスに対して解決されます。部分的な更新を実行するときは、標準のイベントブロックを使用します。

例

ここでは、前のセクションで説明した 3 つの点のバリエーションのサンプルコードをいくつか示します。

部分的な更新 datavar を作成し、それを datavar ベクトルにプッシュします。

```
// Create an empty partial-update datavar.
dfESPdatavar* dvp  = new dfESPdatavar(dfESPdatavar::ESP_LOOKUP);
// Push partial-update datavar onto the vector in the appropriate
// location.
// Other partial-update datavars might also be allocated and pushed to the
// vector of datavars as required.
dvVECT.push_back(dvp); // this would be done for each field in the update
    event
```

そのベクトルにプッシュされた部分的な更新と標準の datavars を使用して部分的な更新を作成します。

```
// Using the datavar vector partially defined above and schema,
// create event.
dfESPeventPtr eventPtr = new dfESPevent();
eventPtr->buildEvent(schemaPtr, dvVECT, dfESPeventcodes::eo_UPDATE,
dfESPeventcodes::ef_PARTIALUPDATE);
```

CSV フィールドを使用して部分更新イベントを定義します。ここで、'^U'値は部分更新フィールドを表します。ここで明示的に'^U'を表示しています。しかし、実際のテキストでは、個々のエディターが異なる方法で制御文字を表示するため、Ctrl-U の文字表現が表示されることがあります。

ここでイベントは更新('u'のため)で、部分更新('p'のため)、キー値は 44001、"ibm" は変更されていない計測器です。機器はフィールドに含まれています。price は 100.23 で、変更された可能性があります。3000 は変更された quantity である可能性があるため、最後の 3 つのフィールドは更新されません。

```
p = new dfESPevent(schema_01,
(char *)"u,p,44001,ibm,100.23,3000,^U,^U,^U");
```

保持操作と復元操作の実装

SAS Event Stream Processing を使用すると、次のことが可能になります。

- 完全なモデル状態をファイルシステムに保持すること
- 直前の持続操作によって作成された永続ディレクトリからモデルを復元すること
- エンジン全体を保持して復元すること
- プロジェクトの持続と復元

モデルの永続オブジェクトを作成するには、クラスコンストラクターにパス名を指定します。 dfESPpersist(char **baseDir*);になります。*baseDir* パラメータは、複数の実行中のイベントストリームプロセッサ間で共有されるディスクを含む、任意の有効なディレクトリを指すことができます。

パス名を指定した後、次の 2 つのパブリックメソッドのいずれかを呼び出します。

```
bool persist();
bool restore(bool dumpOnly=false);
// dumpOnly = true means do not restore, just walk and print info
```

persist()メソッドはいつでも呼び出すことができます。それは高価であることに注意してください。すべてのプロジェクトのイベントブロックインジェクションが中断され、すべてのプロジェクトが静止し、永続データが収集されてディスクに書き込まれ、すべてのプロジェクトが標準の実行状態に戻ります。

restore()メソッドは、プロジェクトが開始される前に呼び出す必要があります。持続ディレクトリに持続データが含まれていない場合、restore()コールは何もしません。

持続オペレーションは、C、Java、Python パブリッシュ/サブスクライブ API でもサポートされています。これらの API 関数では、ターゲットエンジンを示すために *host*:*port* パラメーターが必要です。

C パブリッシュ/サブスクライブ API メソッドは次のとおりです。int C_dfESPpubsubPersistModel(char *hostportURL, const char *persistPath)

Java パブリッシュ/サブスクライブ API メソッドは次のとおりです。 boolean persistModel(String hostportURL, String persistPath)

持続および復元機能の 1 つのアプリケーションは、イベントストリームプロセッサのシステムメンテナンス全体で状態を保存しています。この場合、モデルには、プロジェクトを開始する前に前述した restore()関数の呼び出しが含まれています。実行中のエンジンで後でメンテナンスを実行する場合

1 すべてのパブリッシュクライアントを 調整して一時停止します。

2 1 つのクライアントで前述のパブリッシュ/サブスクライブ持続 API コールを実行させます。

3 システムを停止させ、メンテナンスを行い、システムを復旧させます。

4 restore()関数を実行し、すべてのウィンドウを 手順 2 で保持された状態に復元するイベントストリームプロセッサモデルを Restart します。

5 手順 1 で一時停止したパブリッシングクライアントを再開します。

エンジン全体を持続させるには、次の関数を使用します。

bool dfESPengine::persist(const char * *path*);

void dfESPengine::set_restorePath(const char **path*);

persist に指定するパスは、set_restorePath に指定するパスと同じにすることができます。

プロジェクトを持続させるには、次の関数を使用します。

bool dfESPproject::persist(const char **path*)

bool dfESPproject::restore(const char **path*);

持続させる前に、エンジンを開始してデータをパブリッシュしてください。持続させる際に、アクティブであり 、データを受け取ることができます。

エンジンを持続させるには、call dfESPengine::persist(*path*); を使用します。システムは次を実行します。

1 すべての 受信メッセージを一時停止します(パブリッシュ/サブスクライブを中断する)。

2 キューに入れられたデータの処理を完了します。

3 すべてのキューに 入れられたデータが処理された後、エンジンの状態を指定されたディレクトリに保存し、必要に応じてディレクトリを作成します。

4 エンジン状態が保持された後、パブリッシュ/サブスクライブを再開し、新規作成データをエンジンに流します。

エンジンを復元するには、エンジンを初期化して dfESPengine::set_restorePath(*path*); を呼び出します。dfESPengine::startProjects()の呼び出しが行われると、エンジンの状態全体が復元されます。

プロジェクトコールを持続させるには dfESPproject::persist(*path*);を呼び出します。コールはパブリッシュ/サブスクライブを OFF にし、システムを静止させ、プロジェクトを持続し、パブリッシュ/サブスクライブを再度有効にします。復元に指定されたパスは、通常、保持のパスと同じです。

プロジェクトを復元するには、プロジェクトが開始される前に dfESPproject::restore(*path*);を呼び出します。次に dfESPengine::startProject(*project*);を呼び出します。

注: オンラインプロジェクトをトレーニングするモデルを永続化する場合、学習ウィンドウはモデルの復元時にすぐに新しいモデルのトレーニングを開始することに注意してください。ユーザーの介入は必要ありません。

注: オフライン学習(モデルリーダーウィンドウでモデルを ESP サーバーにパブリッシュするモデル)を保持してからモデルを復元する場合は、モデルリーダーウィンドウにモデルを再公開して、スコアウィンドウで新規作成イベントをスコアリングできるようにする必要があります。

待機時間測定の収集と保存

dfESPlatencyController クラスは、イベントストリーム処理モデルでの待機時間測定の収集と保存をサポートします。待機時間は、待機時間測定が可能なウィンドウを流れるイベント内に 64 ビットのマイクロ秒単位の粒度タイムスタンプを格納することによって算出されます。

さらに、待機時間統計は、待機時間測定の固定サイズの集計で算出されます。これらの測定には平均、最小、最大、および標準偏差が含まれます。集計サイズは設定可能なパラメーターです。待機時間コントローラーのインスタンスを使用して、インジェクトされたイベントが通過するすべてのソースウィンドウといくつかのダウンストリームウィンドウ間の待機時間を測定できます。

待機時間コントローラーを使用すると、イベントブロックの入力ファイルと、それらのイベントがソースウィンドウにインジェクトされる速度を指定できます。ディスク読み込みが要求されたインジェクト速度を歪ませないように、インジェクション前に完全な入力ファイルをメモリにバッファリングします。

測定データを含む出力テキストファイルを指定します。このテキストファイルの各行には、イベントのバケット上で収集された待機時間に関する統計が含まれています。バケット内のイベントの数は、構成された集計サイズです。ラインには、モデルを流れるイベントの次のバケットの統計などが含まれます。

出力テキストファイルの各行は、タブで区切られた 3 つの列で構成されています。左から順に、これらの列には次のものが含まれます。

- バケット内の最大待機時間
- バケット内の最小待機時間
- バケットの平均待機時間

集計サイズは、イベントの合計数より少ない任意の値に設定できます。実行可能な値は、意味のある平均値を得るのに十分な大きさですが、実行中のさまざまな TIME にいくつかのサンプルを取得するのに十分小さい値です。

パブリッシュ/サブスクライブクライアントが関与している場合は、パブリッシャー/サブスクライバーコードを変更するか、ファイル/ソケットアダプターを使用してネットワーク待機時間も含めることができます。

モデル内の待ち時間のみを測定する場合

1 モデルに"int/dfESPlatencyController.h"を含め、main()に dfESPlatencyController オブジェクトのインスタンスを追加します。

2 dfESPlatencyController オブジェクトの次のメソッドを呼び出して構成します。

方法	説明
void set_playbackRate(int32_t r)	要求されたインジェクト速度を設定します。
void set_bucketSize(int32_t bs)	前述の bucketSize パラメーターを設定します。
void set_maxEvents(int32_t me)	インジェクトするイベントの最大数を設定します。
void set_oFile(char *ofile)	待機時間統計情報を含む出力ファイルの名前を設定します。
void set_iFile(char *ifile)	バイナリイベントブロックデータを含む入力ファイルの名前を設定します。
void set_stampBlkSize(int64_t stampsize)	待機時間モードでタイムスタンプを格納するために割り当てるメモリのブロックサイズ(イベント数)を指定します。追加ブロックは必要に応じて割り当てられます。
void set_skipSize(int32_t ss)	待機時間算出で無視する集計開始ブロックと集計終了ブロックの数を指定します。

3 終了タイムスタンプでイベントにタイムスタンプを付けるウィンドウに、サブスクライバーコールバックを追加します。コールバック内で、

dfESPlatencyController オブジェクトの次のメソッドへの呼び出しを追加します。void record_output_events(dfESPeventblock *ob)になります。これにより、イベントブロック内のすべてのイベントに終了タイムスタンプが追加されます。

4 プロジェクトを開始した場合、dfESPlatencyController オブジェクトの次のメソッドを呼び出します。

方法	説明
void set_injectPoint(dfESPwindow_source *s)	イベントのタイムスタンプを開始 TIME スタンプとするソースウィンドウを設定します。
void read_and_buffer()	設定された入力ファイルから入力イベントブロックを読み取り、バッファリングします。
void playback_at_rate()	TIME スタンプはイベントを入力し、設定されたイベント数まで、設定されたレートでモデルにインジェクトします。

5 モデルを休止し、dfESPlatencyController オブジェクトの次のメソッドを呼び出します。void generate_stats()になります。および 4 と 0 を metaHigh および metaLow パラメータをそれぞれ指定します。これにより、各イベントの正しいメタデータ位置から開始および終了タイムスタンプが収集され、構成された出力ファイルに待機時間統計量が書き込まれます。

パブリッシュ/サブスクライブクライアントを変更してモデルとネットワークの待ち時間を測定する場合

1 モデルでは、dfESPengine setLatencyMode()関数をプロジェクトを開始する前に呼び出します。

2 パブリッシャークライアントアプリケーションでは、C_dfESPpublisherInject()を呼び出す直前に C_dfESPlibrary_getMicroTS()を呼び出して現在のタイムスタンプを取得します。イベントブロック内のすべてのイベントをループし、イベントごとに C_dfESPevent_setMeta(event, 0, timestamp)を呼び出して、タイムスタンプをイベントに書き込みます。これにより、パブリッシュ/サブスクライブインジェクションタイムスタンプがメタロケーション 0 に記録されます。

3 モデルインジェクトおよびサブスクライバーコールバックタイムスタンプは、すべてのイベントでメタロケーション 2 および 3 に自動的に記録されます。これは、待機時間モードがエンジンで有効になっているためです。

4 インジェクションループにコードを追加して、固定インジェクト速度を実装します。サンプル速度制限コードについては、待機時間パブリッシュ/サブスクライブクライアントの例を参照してください。

5 サブスクライバークライアントアプリケーションで、"int/dfESPlatencyController.h"を含有し、dfESPlatencyController のインスタンスを追加します。

6 前に説明したように、待機時間コントローラーの bucketSize パラメーターと playbackRate パラメーターを設定します。

7 サブスクライバーコールバックが待機時間コントローラーにアクセスできるように、待機時間コントローラーオブジェクトをコンテキストとして C_dfESPsubscriberStart()に渡します。

8 サブスクライバーコールバックをイベントブロックと共に待機時間コントローラーを C_dfESPlatencyController_recordOutputEvents()に渡すようにします。これにより、パブリッシュ/サブスクライブコールバックタイムスタンプがメタロケーション 4 に記録されます。

9 サブスクライバークライアントアプリケーションがすべてのイベントを 受信すると、各イベントに記録された 4 つのタイムスタンプの任意のペア間の待機時間に関する統計を生成できます。出力ファイルを設定するには、最初に C_dfESPlatencyController_setOFile()を呼び出します。次に、C_dfESPlatencyController_generateStats() を呼び出して待機時間コントローラーと 2 つのタイムスタンプを渡して、ファイルに統計を書き込みます。可能なタイムスタンプペアのリストとその TIME スパンは次のとおりです。

   - (0、2)-パブリッシャークライアントからのインジェクトからモデルによるインジェクト
   - (0、3)-パブリッシャークライアントからモデルによるサブスクライバーコールバックへのインジェクト
   - (0、4)-パブリッシャークライアントからサブスクライバークライアントによるコールバックへのインジェクト(フルパス)
   - (2、3)-モデルからサブスクライバーコールバックへのモデルからのインジェクト
   - (2、4-)-モデルによってインジェクトされてからサブスクライバークライアントによるコールバック
   - (3、4)-モデルによるサブスクライバーコールバックからサブスクライバークライアントによるコールバック

10 他のタイムスタンプのペアの統計をさらに生成するには、出力ファイルをリセットし、C_dfESPlatencyController_generateStats()を再度呼び出します。

ファイル/ソケットアダプターを使用してモデルとネットワークの待機時間を測定するには、パブリッシャーアダプターとサブスクライバーアダプターを 標準どおり実行しますが、これらの追加スイッチを使用して実行します。

	パブリッシャー
—r *rate*	要求された送信率を 1 秒あたりのイベント数で指定します。
-m *maxevents*	パブリッシュするイベントの最大数を指定します。

パブリッシャー	
-p	パブリッシュする前にすべてのイベントをバッファリングするように指定します。
-n	待機時間モードを有効にします。

サブスクライバー	
-r *rate*	要求された送信率を 1 秒あたりのイベント数で指定します。
-a *aggrsize* <*aggr_blocks_to_ignore*>	集計バケットサイズを指定します。カンマ区切りの値の後に、待機時間算出で無視する集計開始ブロックと集計終了ブロックを指定できます。
-n	待機時間モードを有効にします。
-N*latencyblksize*	待機時間モードでタイムスタンプを格納するために割り当てるメモリのブロックサイズ(イベント数)を指定します。追加ブロックは必要に応じて割り当てられます。

サブスクライバーアダプターは、それぞれのパブリッシャーおよびサブスクライバーアダプターの URL で指定されたウィンドウについて前述した 4 つのタイムスタンプをすべて 収集します。実行の最後に、現在のディレクトリのファイルに統計データを書き込みます。これらのファイルの名前は latency_**transmit rate_high timestamp_low timestamp**"となります。ここで表される high timestamp および low timestamp は、前述のタイムスタンプペアに対応します。

finalized-callback を有効化

一部のデータ構造は、ウィンドウとエッジが作成されたときに完全に作成されますが、プロジェクトの開始直前にファイナライズされます。これらのデータ構造には、導出されたスキーマおよび特定の種類のウィンドウインデックスが含まれます。finalized-callback 関数は、すべてのデータ構造体が完全に初期化されたときに、イベントがウィンドウに流入を開始する前に呼び出されます。finalized-callback 関数は、アプリケーションまたは XML モデルに必要な状態情報または接続情報を初期化できます。

finalized-callback を次のように有効にします。

- XML の finalized-callback エレメントを使用します。ウィンドウコールバック関数とウィンドウが呼び出す関数の名前を含むライブラリの名前を指定します。

  <finalized-callback name='library' function='fin_callback'>

- C++で次の関数を使用します dfESPwindow::addFinalizeCallback(dfESPwindowCB_func *cbf*)。

Using Code-Based Routing

Overview

Code-based routing enables you to configure communication between ESP servers by defining routes within a project model. Multiple routes are allowed. Each route consists of independently defined objects (for example, connections, sources, destinations, routers, and modifiers) that are connected using edges. For more information, see "モデルの構造のための XML 言語要素" in *SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*.

Stock Trade Example

This example uses stock trade data. It consists of three models called source.xml, exchanges.xml, and sectors.xml. The input data file was created by GitHub Copilot and contains 98 of the most active symbols:

```
{
  rank = 1,
  symbol = "NVDA",
  company = "NVIDIA Corporation Common Stock",
  exchange = "NASDAQ",
  sector = "Technology",
  min_trade_price = 138.83,
  max_trade_price = 236.54,
  min_trade_quantity_est = 161622849,
  max_trade_quantity_est = 162981498,
 },
 {
  rank = 2,
  symbol = "AAPL",
  company = "Apple Inc. Common Stock",
  exchange = "NASDAQ",
  sector = "Technology",
  min_trade_price = 195.07,
  max_trade_price = 316.94,
  min_trade_quantity_est = 44311442,
  max_trade_quantity_est = 45325667,
 },
 ...
```

The source.xml model contains a Lua connector that generates events and route definitions. The connector reads the input data and generates random entries from the file to match the Source window schema:

```
<window-source name="trades" insert-only="true"> 1
 <schema>
  <fields> 2
   <field key="true" name="id" type="int64"/>
   <field name="symbol" type="string"/>
   <field name="company" type="string"/>
   <field name="sector" type="string"/>
   <field name="exchange" type="string"/>
   <field name="price" type="double"/>
   <field name="quantity" type="int32"/>
   <field name="broker" type="string"/>
   <field name="buyer" type="string"/>
   <field name="seller" type="string"/>
  </fields>
 </schema>
</window-source>
```

1 This line defines a Source window called trades that accepts insert events only.

2 Several event fields are defined using names and types.

The exchanges.xml model consists of two continuous queries, one for NYSE and one for NASDAQ. The input Source window is the same as the window for source.xml with the addition of an event field called tradevalue with type **int64**.

The sectors.xml model also consists of two continuous queries, one for focus sectors and one for other sectors. The input Source window is the same as the window for exchanges.xml.

This example uses a source model that generates stock trade data that is pushed into Source windows that are in two other servers. The router configuration is shown here:

```
<routes>
 <route name="traderoute">
  <connections> 1
   <connection name="exchangesserver" esp-project="exchanges" url="http://
espsrv04:3333"/>
    <connection name="sectorsserver" esp-project="sectors"  url="http://espsrv04:4444"/>
  </connections>
  <sources>
   <source name="src" window="trades"/> 2
  </sources>
  <destinations> 3
   <destination name="NYSE" connection="exchanges" window="NYSE/src"/>
   <destination name="NASDAQ" connection="exchanges" window="NASDAQ/src"/>
   <destination name="focusedSectors" connection="sectors" window="focused/src"/>
   <destination name="otherSectors" connection="sectors" window="other/src"/>
  </destinations>
  <routers>
   <router name="trades" func="routeTrade" use="exchange,sector"/> 4
  </routers>
  <modifiers>
```

```
  <modifier name="tradeValue" func="addTradeValue" use="quantity,price"/>
 </modifiers>
 <edges>
  <edge source="src://src" target="router://trades" /> 5
  <edge source="dest://*" target="mod://tradeValue" /> 6
 </edges>
 <code><![CDATA[

  function addTradeValue(event)
     local   data = {}
     data.tradevalue = event.quantity * event.price
     return data
  end

  function routeTrade(event)

     local   dests = {}

     dests[#dests + 1] = event.exchange

     if (event.sector == "Technology" or event.sector == "Health Care")
     then
      dests[#dests + 1] = "focusedSectors"
     else
      dests[#dests + 1] = "otherSectors"
     end

     return dests
  end

 ]]></code>
 </route>
</routes>
```

1 The connections element defines the other two ESP servers. They are called exchangesserver and sectorsserver.

2 The source element defines the single Source window to receive events from.

3 The destinations element defines four destinations. Each server has two destinations. The destinations point to different continuous queries in the respective servers.

4 The router element defines a single router to route events using the routeTrade function. The function uses the exchange event field to route the event to the matching destination, which is either NYSE or NASDAQ. The sector event field is used to route the event to either the focusedSectors or otherSectors destination. Because exchange and sector are the only event fields required for this routing, they are specifies in the use attribute.

5 This edge takes elements from the src source and sends them through the trades router, which returns the appropriate destinations.

6 This edge applies modifications to all events. The modifications are defined in the tradeValue modifier.

2 部

複数のウィンドウに共通の機能

8

共通の Lua 機能

ESP プロジェクトのユーザー提供のコード

SAS Event Stream Processing は、独自のコード(Python、Lua、またはその他のカスタムコンポーネントなど)でプロジェクトを拡張できます。使用することを決定したコードのビヘイビアーとセキュリティについては、ユーザーとその組織のみが責任を負っています。SAS Event Stream Processing は、セキュリティ目的でユーザーが提供したコードをレビューまたはテストしません。ESP プロジェクトに追加するコードやその他のカスタマー提供の資料は、ユーザーの責任のままです。

注意
安全でないコードを実行しないでください。 安全でないコードを実行すると、環境がデータの損失、データの破損、セキュリティ違反、またはシステムの不安定性につながる可能性があります。組み込むすべてのコードが安全であり、テストされ、組織のポリシーと適用される法律に準拠していることを確認するのはあなたの責任です。

Lua プログラミングの概念

Lua は動的に型付けされたプログラミング言語です。軽量で組み込み可能で、プラグインプログラムの作成に最適になるように設計されています。Lua は ESP サーバーに組み込まれているため、プロジェクト内で作成できる Lua エンティティ内で Lua コードを実行するために、外部の Lua インタープリターは必要ありません。Lua 言語の詳細については、Lua ドキュメントを参照してください。

重要 Lua コードには、次の 2 つの例外を除き、標準 Lua ライブラリで提供される関数を含めることができます:

- os.execute
- io.popen

これらの関数のいずれかを使用すると実行時エラーが発生し、プロジェクトをロードできなくなります。

テーブルは、Lua の主なデータ構造を提供します。テーブルは、キーまたはインデックスによってアクセスするオブジェクトです。Lua テーブルを使って、SAS Event Stream Processing オブジェクトを Lua オブジェクトに変換したり、逆に SAS Event Stream Processing オブジェクトを Lua テーブルに変換したりすることがで

きます。Lua のオブジェクトは、JavaScript のオブジェクトと同じように扱うことができます。

employee という名前の Lua テーブルを見てみましょう。次の 2 つのフィールドがあります。

表 8.1 Lua テーブル

キー	値
name	Jose
age	30

括弧内のキーを参照できます。

```
local employee = {}
employee["name"] = "Jose"
employee["age"] = 30
```

また、ドット表記でキーを参照することもできます。

```
local employee = {}
employee.name = "Jose"
employee.age = 30
```

次の Lua コードは、前のテーブルの値を繰り返し処理します。

```
for key,value in pairs(employee)
do
   print(key,"=",esp_toString(value))
end
```

次の Lua テーブルは、インデックス付きの配列を使ってフィールド値にアクセスしています。

注: Lua のインデックス付き配列は、インデックス 0 ではなく、インデックス 1 から始まります。

```
local names = {}
o[1] = "Jose"
o[2] = "Sally"
o[3] = "Miko"
```

次の Lua のコードは、そのテーブルの値を繰り返し処理します。

```
for index,value in ipairs(names)
do
   print(esp_toString(index),"=",value)
end
```

Lua エンティティの初期化

*初期化*とは、変数データの定義済みの値を見つけて使用するプロセスです。ESP サーバーの起動時に、SAS Event Stream Processing プロジェクト内のすべての Lua エンティティで初期化メカニズムを使用できます。Lua エンティティが異なれば、初期化の方法も異なります。

次の Lua エンティティは、対応する XML 要素の init 属性を通して初期化メカニズムを実装します。

- Lua ウィンドウ(window-lua)
- Lua ベースのパターンウィンドウ(window-pattern、init 属性は各パターンに適用されます)
- Lua ベースのフィルターウィンドウ(window-filter)

Lua コネクタエンティティは、Lua コードの初期化関数を指すオプションの init パラメーターを提供します。

init と呼ばれる関数を Lua コードに含めることで、Lua splitter エンティティを初期化します。

ESP サーバーが起動すると、Lua コードの処理に関連するアクションが次の順序で実行されます。

1. Lua スニペットの永続的なプロパティを読み込みます。
2. Lua ウィンドウの永続的なプロパティを読み込みます。
3. 1 つのプロパティを含む Lua スニペットの場合、init 関数を呼び出します。
4. スレッド化された Lua スニペットを開始します。
5. 1 つのプロパティを含む Lua ウィンドウ、Lua ベースパターンウィンドウ、または Lua ベースフィルターウィンドウの場合、init 関数を呼び出します。ウィンドウに 1 つのプロパティを含む Lua スプリッターが含まれている場合、init 関数を呼び出します。

上記のすべての操作は、Lua エンティティがプロジェクトで定義されている順序で実行されます。

他の Lua エンティティからプロパティ 値を取得する Lua エンティティがある場合は必ず、初期化関数を使用することをお勧めします。たとえば、次のコードを考えてみましょう。

```
local myvalue = esp_getProperty({name="value"})
```

コードが呼び出されたときに、以前の Lua エンティティが **value** プロパティを設定していなかった場合、コードは **nil** 値を生成します。ただし、コードを初期化関数に入れると、他の Lua エンティティが初期化されたときに、プロパティが設定されていることを確認できます。

```
local   myvalue = nil
```

```
function init()
   myvalue = esp_getProperty({name="value"})
end
```

Lua での SAS Event Stream Processing 関数の使用

概要

ESP サーバーは、次のアクティビティを 可能にする Lua コード内で呼び出すことができる関数を提供します。

表 8.2　アクティビティごとに 整理された関数

アクティビティ	関数
イベントの操作	esp_publish esp_publishAsynch esp_query esp_subscribe esp_unsubscribe esp_waitForEvent
JSON の解析とエンコード	esp_jsonEncoder esp_parseJson esp_parseJsonFrom
XML の解析	esp_parseXml esp_parseXmlFrom
正規表現の操作	esp_getExpressionValues esp_getExpressionValuesFrom esp_setExpression
HTTP 要求の操作	esp_sendHttp
GUID の生成	esp_getGuids

アクティビティ	関数
デバッグ	esp_logMessage
CSV データを直接挿入する	esp_injectCsv
その他の関数	esp_getUTCOffset esp_getUTCOffsetHours esp_inlist esp_isDone esp_toString esp_getServerProperty esp_getProjectProperty
文字列関数	esp_strings
イベントコンテキスト関数	esp_getEventFlags esp_getEventOpcode
日付と時間の関数	esp_dateMath esp_formatDate esp_formatTime esp_getSystemMicro esp_getSystemMilli esp_parseDate
Lua プロパティの操作 (271 ページ)	esp_setProperty esp_getProperty esp_clearProperty esp_addPropertyListener
リレーショナルデータベースへのアクセス	esp_sqlConnect esp_sqlDisconnect esp_sqlSelect esp_sqlOpen esp_sqlNext esp_sqlClose esp_sqlInsert esp_sqlUpdate esp_sqlDelete

重要　2022.10 より前のリリースでは、関数には esp_接頭語が含まれていませんでした。たとえば、esp_parseJson 関数は parseJson でした。2022.10 以降、多くの関数のパラメーターが変更されました。2022.10 より前のリリースからプロジェクトを移行する場合は、これらの違いに注意してください。

イベントの操作

イベントのパブリッシュ

esp_publish または esp_publishAsynch 関数を使用して、イベントをソースウィンドウにパブリッシュします。この関数は、現在のイベントストリームとは異なるイベントストリームにイベントデータを送信する場合に役立ちます。

表 8.3　esp_publish 関数と esp_publishAsynch 関数のパラメーター

パラメーター	説明	
window	ソースウィンドウの名前を指定します。次のいずれかの形式を使用します。 ■ *project/query/window* ■ *query/window* ■ *window*	
events	パブリッシュするイベントデータ(単一の Lua オブジェクトまたはオブジェクトの配列)を指定します。	
opts	次のパブリッシュオプションのいずれかを指定します。	
	logs=true | false	true の場合、ESP サーバーのログはパブリッシュ中に収集され、応答で返されます(esp_publish の場合のみ有効)。
	blocksize	パブリッシュの *blocksize* を指定します。デフォルト値は **1** です。

例:

```
local response = esp_publish({window="query/window",events=events,opts={logs=true}})
local response = esp_publishAsync({window="query/window",events=events})
```

esp_publish 関数は、値を返す前にクエリにイベントを挿入します。esp_publishAsync 関数は、イベントをキューに入れて非同期に処理します。戻り値には、コードフィールドとメッセージフィールドが含まれます。logs パラメーターを指定すると、サーバーログが logs フィールドに戻されます。

デフォルトでは、Insert オペコードはイベントのパブリッシュに使用されます。Lua オブジェクトまたは一連のオブジェクトの_opcode フィールドの値を設定すると、その値はオペコードの設定に使用されます。

たとえば、株式取引データを処理し、ブローカーの統計を独自のソースウィンドウに送信するとします。次の Lua ウィンドウは、イベントストリームから取引イベントを受け取ります。最初にブローカーの名前(brokerName)を使用して、各ブローカーの取引統計(price、quant)を保存します。次に、計算を実行して取引数量と価格の最大値と平均値(stats.max_*x*、stats.avg_*x*)を決定します。

```
<window-lua name="publishStats" events="f">
 <schema>
  <fields>
   <field name="id" type="string" key="true"/>
  </fields>
 </schema>
 <use>brokerName,price,quant</use>
 <code><![CDATA[

  local   brokerdata = {}

  function f(data)
     local   stats = brokerdata[data.brokerName]

     if stats == nil
     then
        stats =
{broker=data.brokerName,numtrades=1,max_price=data.price,avg_price=data.price,max_q
uant=data.quant,avg_quant=data.quant,total_price=data.price,total_quant=data.quant}
        stats._opcode = "upsert"
        brokerdata[data.brokerName] = stats
     else
        stats.numtrades = stats.numtrades + 1
        stats.total_price = stats.total_price + data.price
        stats.total_quant = stats.total_quant + data.quant

        if data.price > stats.max_price
        then
           stats.max_price = data.price
        end

        if data.quant > stats.max_quant
        then
           stats.max_quant = data.quant
        end

        stats.avg_price = stats.total_price / stats.numtrades
        stats.avg_quant = stats.total_quant / stats.numtrades
     end

     esp_publish("brokerstats",stats)
```

```
    return nil
  end

 ]]></code>
</window-lua>
```

イベント作成関数が nil を返すため、イベント自体がストリーム内に作成されないことに注意してください。この関数は、イベントを別のパスにパブリッシュするだけです。

Lua ウィンドウは、同じ連続クエリで統計を brokerstats ウィンドウにパブリッシュします。

```
<window-source name="brokerstats" index="pi_HASH">
 <schema>
  <fields>
   <field name="broker" type="string" key="true"/>
   <field name="numtrades" type="int32"/>
   <field name="max_price" type="double"/>
   <field name="avg_price" type="double"/>
   <field name="max_quant" type="double"/>
   <field name="avg_quant" type="double"/>
  </fields>
 </schema>
</window-source>
```

イベントのサブスクライブ

イベントをサブスクライブするには、esp_subscribe 関数を使用します。この関数は、単一の Lua オブジェクトをパラメーターとして受け取ります。

表 8.4 esp_subscribe 関数のパラメーター

パラメーター	説明	
window	イベントを受信するウィンドウを指定します。	
handler	イベントの受信に使用される Lua 関数の名前を指定します。	
output	次の出力オプションのいずれかを指定します。	
	fields	イベント配信に含めるフィールドのカンマ区切りリストを指定します。
	exclude=true | false	*fields* に指定された項目を含めるか除外するかを指定します。デフォルト値は **false** です。

パラメーター	説明	
	encode_binary=true | false	イベント配信用にバイナリデータをエンコードするために Base64 を使用するかどうかを指定します。デフォルト値は **false** です。

例:

```
local subscriber = esp_subscribe({window="project/query/window",handler="handler"})
function handler(events) print("received some events: "..esp_toString(events)) end
```

イベントをサブスクライブするとサーバーにサブスクリプションリソースが作成されるため、このリソースが不要になった場合はクリーンアップする必要があります。esp_unsubscribe を使用します。

```
function done()
   esp_unsubscribe(subscriber)
end
```

subscriber パラメーターは、esp_subscribe 関数呼び出しによって返された値です。

以下は、あるウィンドウをサブスクライブし、別のウィンドウにパブリッシュする Lua スニペットの例です。

```
<lua-snippet name="snippet" init="init" destroy="done">
  <code><![CDATA[

    local  subscriber = nil

    function init()
      subscriber = esp_subscribe({window="p/cq/alert",handler="handler"})
    end

    function handler(events)
      esp_publish({window="cq2/storeAlerts",events=events,opts={logs=true}})
    end

    function done()
      esp_unsubscribe(subscriber)
    end

  ]]></code>
</lua-snippet>
```

イベントのクエリ

esp_query 関数を使用してイベントをクエリします。この関数は、単一の Lua オブジェクトをパラメーターとして受け取ります。

表 8.5　esp_subscribe 関数のパラメーター

パラメーター	説明	
window	イベントを受信するウィンドウを指定します。	
output	次の出力オプションのいずれかを指定します。	
	fields	イベント配信に含めるフィールドのカンマ区切りリストを指定します。
	exclude=true | false	fields に指定された項目を含めるか除外するかを指定します。デフォルト値は **false** です。
	encode_binary=true | false	イベント配信用にバイナリデータをエンコードするために Base64 を使用するかどうかを指定します。デフォルト値は **false** です。
filter	次のフィルタオプションのいずれかを指定します。	
	func	イベントのフィルタリングに使用される関数の名前を指定します。関数がイベントを受け取り、イベントが目的の基準に一致する場合に true を返すようにコードを作成します。それ以外の場合は、関数が false を返すようにします。
	use	次のオプションのいずれかを使用して、フィルター関数に送信するフィールドを指定します。 ■ fields: 並べ替えるフィールドのカンマ区切りリストを指定します。 ■ dir: ソート方向を*昇順*または*降順*(デフォルト)に指定します。 ■ maxevents は、maxevents 返されるイベントの最大数を指定します。デフォルトで

パラメーター	説明
	は、すべてのイベントが返されます。

例:

```
local   patients = esp_query({window="patients"})
local   vitals = esp_query({window="vitals"})
```

株式取引を処理するモデルがあり、ブローカー Joe による上位 10 件の取引を数量順に取得したいとします。

```
local   events = esp_query({window="p/cq/trades",
                    filter={func="filter",use="broker"},
                    output={fields="symbol,quantity,price"},
                    sort={fields="quantity",maxevents=10})

print("events\n"..esp_toString(events))

function filter(event)
   return event.broker == "Joe"
end
```

イベントの待機

esp_waitForEvent 関数を使用して、イベントが発生するのを待ってから処理を続行します。

表 8.6 esp_waitForEvent 関数のパラメーター

パラメーター	説明
window	イベントを受信するウィンドウを指定します。
match	イベントが一致するかどうかを判断するために使用される関数の名前を指定します。このパラメーターは **true** または **false** を返します。
timeout	イベントの発生を待つ時間を指定します。**10 秒**や **1 分**などの単位を使用します。このパラメータのデフォルトはタイムアウトなしです。
send_update_deletes=true | false	更新ブロックの一致関数に削除イベントを送信するかどうかを指定します。デフォルト値は **false** です。
snapshot=true | false	ウィンドウの現在の内容を処理するかどうかを指定します。**false** (デフォルト値)の場合、関数は将来のイベントのみを処理します。

パラメーター	説明	
use	match 関数に送信されるイベントフィールドの処理方法を指定します。	
	fields	match 関数に送信されるイベントに含めるフィールドのカンマ区切りリストを指定します。
	exclude=true | false	フィールド内の項目を含めるか除外するかを指定します。デフォルト値は **false** です。
	encode_binary=true | false	イベント配信用にバイナリデータをエンコードするために Base64 を使用するかどうかを指定します。デフォルト値は **false** です。
output	出力イベントに含めるイベントフィールドを処理する方法を指定します。	
	fields	match 関数に送信されるイベントに含めるフィールドのカンマ区切りリストを指定します。
	exclude=true | false	フィールド内の項目を除外するかどうかを指定します。デフォルト値は **false** です。
	encode_binary=true | false	イベント配信用にバイナリデータをエンコードするために Base64 を使用するかどうかを指定します。デフォルト値は **false** です。

例:

```
local   event,err = esp_waitForEvent({window="p/cq/brokerAlertsAggr",
                                      match="match",
                                      timeout="5 seconds",
                                      snapshot=true})

function match(event)
   return event.value > 10
end
```

この関数は、必要なイベントが発生した場合にそれを返します。それ以外の場合、関数は nil またはエラーを返します。エラーがあった場合、2 番目の戻り値には失敗の理由を示すメッセージが含まれます。

株式取引を分析するプロジェクトがあるとします。**ジョー**という名前のトレーダーが 10 件の取引違反を蓄積してから行動を起こしたいと考えています。この機能を実行するように Lua スニペットを設定できます。

```
<lua-snippet name="snippet">
  <lua-thread func="dowait" interval="1 second"/>
  <code><![CDATA[

    function dowait()
      io.write("running dowait, waiting for event...")
      io.flush()

      local  event,err = esp_waitForEvent({window="p/cq/brokerAlertsAggr",
                                           match="match",
                                           timeout="5 seconds",
                                           use={fields="brokerName,total"},
                                           output={fields="frontRunningBuy",exclude=true},
                                           snapshot=true})

      if (err ~= nil)
      then
        io.write("failed to get event, "..err.."\n")
      else
        io.write("got event, dowait continuing\n")
        io.write(esp_toString(event))
      end

      return true
    end

    function match(event)
      return event.brokerName == "Joe" and event.total > 10
    end

  ]]></code>
</lua-snippet>
```

イベント生成時に GUID を生成する

esp_getGuids 関数を使うと、イベントを生成する際にグローバルに一意な識別子(GUID)を生成できます。

表 8.7　esp_getGuids 関数

関数	説明
esp_getGuids(*numguids*)	ESP サーバーから *numguids* GUID を取得します。GUID はインデックス付きの配列で Lua に返されます。

関数	説明
	numguids のデフォルト値は **1** です。

たとえば、次のコードでは 5 つの GUID を生成しています。

```
local   guids = esp_getGuids(5)

for index,guid in ipairs(guids) do
   -- Generate event with GUID
end
```

JSON の解析

JSON は属性と値のペアでデータを送信します。これは、リアルタイムのステートレス通信プロトコルで広く使用されています。

次の関数を使用すると、Lua コードで JSON を解析できます。

表 8.8 JSON 解析関数

関数	説明
esp_parseJson(*stringdata*)	Lua から ESP サーバーに文字列データを送信すると、それが解析され、Lua オブジェクトとして返します
esp_parseJsonFrom("*stringdata*")	ESP サーバーから Lua に文字列データを送信します。文字列は、入力イベントのフィールドの名前を指定する必要があり 、それが解析され、Lua オブジェクトとして返されます。

注: この関数は、入力イベントが処理される場合にのみ使用してください。

JSON 形式のセンサーデータを含む sensorinfo という名前のイベントフィールドがあるとします。esp_parseJson への呼び出しを次のように指定します。

```
local   parsed = esp_parseJson(data.sensorinfo)
```

関数はイベントデータで渡されているフィールドを解析しているため、オブジェクト表記が使用されます。esp_parseJson 関数は、イベントデータだけでなく、あらゆるデータに使用できる汎用ユーティリティです。

esp_parseJsonFrom の呼び出しを次のように指定します。

```
local   parsed = esp_parseJsonFrom("sensorinfo")
```

esp_parseJsonFrom 関数では、呼び出しが ESP サーバーからデータをピックアップしているため、Lua コードにデータを渡す必要はありません。

sensorinfo というイベントフィールドに、次のような入力データ(JSON 形式)が届いたとします。

```
[
  {
    device-id:1000,
    sensor-id:10,
    value:1.54
  },
  {
    device-id:1000,
    sensor-id:20,
    value:9.23
  },
  {
    device-id:2000,
    sensor-id:10,
    value:3.24
  }
]
```

この JSON データからイベントを作成するとします。Lua ウィンドウは次のようになります。

```
<window-lua name="lua" events="create">
 <schema>
  <fields>
   <field name="id" type="string" key="true"/>
   <field name="device_id" type="string"/>
   <field name="sensor_id" type="string"/>
   <field name="value" type="double"/>
  </fields>
 </schema>
 <use/>
 <code><![CDATA[

  local   eventId = 1

  function create(data,context)

    local   events = {}
    local   sensorinfo = esp_parseJsonFrom("sensorinfo")

    for index,value in ipairs(sensorinfo)
    do
      local   e = {}

      e.id = esp_toString(eventId)
      e.device_id = value.device_id
      e.sensor_id = value.sensor_id
      e.value = value.value
      events[index] = e

      eventId = eventId + 1
    end

    return(events)
  end
```

```
 ]]></code>
</window-lua>
```

例については、Lua ウィンドウを使用して JSON からイベントを生成するを参照してください。

JSON のエンコード

esp_jsonEncoder(*object*,*format*)関数を使用して、Lua オブジェクトを JSON 文字列としてエンコードします。

表 8.9 esp_jsonEncoder 関数のパラメーター

パラメーター	説明
object	JSON に変換する Lua オブジェクトの名前を指定します。
format	JSON をフォーマットする(整形する)かどうかを指定するブール値です。デフォルト値は **false** です。

次のイベント作成関数を考えてみましょう。

```
function create(data,context)
   local   e = {}
   e.id = 1
   local   afc = {"East","North","South","West"}
   local   nfc = {"East","North","South","West"}
   local   nfl = {nfl={divisions={afc,nfc}}}
   e.league = esp_jsonEncoder(nfl)
   return e
end
```

この関数は、次のようなイベントを作成します。

```
<event opcode="insert" window="p/cq/code">
 <value name="id">1</value>
 <value name="league">
{
   "nfl":{
      "divisions":[
         [
            "East",
            "North",
            "South",
            "West"
         ],
         [
            "East",
            "North",
```

```
            "South",
            "West"
          ]
        ]
      }
}
</value>
</event>
```

XML の解析

次の関数を使用すると、Lua ウィンドウで XML を解析できます。

表 8.10 XML 解析関数

関数	説明
esp_parseXml(*stringdata*)	Lua から ESP サーバーに文字列データを送信すると、それが解析され、Lua オブジェクトとして返します
esp_parseXmlFrom("*stringdata*")	ESP サーバーから Lua に文字列データを送信します。文字列には、入力イベントのフィールドの名前を指定でき 、それが解析され、Lua オブジェクトとして返されます。

XML 形式のセンサーデータを含む sensorinfo というイベントフィールドがあるとします。esp_parseXml の呼び出しは次の通りです。

```
local parsed = esp_parseXml(data.sensorinfo)
```

esp_parseXml 関数は、イベントデータだけでなく、あらゆるデータに使用できる汎用ユーティリティです。

esp_parseXmlFrom の呼び出しは次の通りです。

```
local parsed = esp_parseXmlFrom("sensorinfo")
```

sensorinfo というフィールドに、次のようなセンサーデータ(XML 形式)が届いたとします。

```
<data>
  <sensorinfo>
    <device_id>1000</device_id>
    <sensor_id>10</sensor_id>
    <value>1.54</value>
  </sensorinfo>
  <sensorinfo>
    <device_id>1000</device_id>
    <sensor_id>20</sensor_id>
    <value>9.23</value>
  </sensorinfo>
  <sensorinfo>
```

```
    <device_id>2000</device_id>
    <sensor_id>10</sensor_id>
    <value>3.24</value>
  </sensorinfo>
</data>
```

この XML からイベントを作成したい場合は、次のように Lua ウィンドウを指定できます。

```
<window-lua name="lua" events="create">
 <schema>
  <fields>
   <field name="id" type="string" key="true"/>
   <field name="device_id" type="string"/>
   <field name="sensor_id" type="string"/>
   <field name="value" type="double"/>
  </fields>
 </schema>
 <use/>
 <code><![CDATA[

  local   eventId = 1

  function create(data,context)

    local   events = {}
    local   sensorinfo = esp_parseXmlFrom("sensorinfo")
    print(esp_toString(sensorinfo))

    for index,value in ipairs(sensorinfo[1].data)
    do
      local   e = {}

      e.id = esp_toString(eventId)
      e.device_id = value.sensorinfo.device_id
      e.sensor_id = value.sensorinfo.sensor_id
      e.value = value.sensorinfo.value
      events[index] = e

      eventId = eventId + 1
    end

    return(events)
  end

 ]]></code>
</window-lua>
```

注: XML データが直感的に Lua オブジェクトにマッピングしない場合、print(esp_toString(sensorinfo))を使って、データの構造を公開することができます。次に、データを取得するためにその構造をたどることができます。

正規表現の操作

ESP サーバーは、esp_setExpression 関数を通して正規表現を評価する機能を備えています。

esp_setExpression(*name*,*regex*,ignorecase)

この関数は、コンパイルされたバージョンの正規表現(*regex*)を ESP サーバーに保存し、*name* でアクセスできるようにします。

ignorecase パラメーターはオプションです。式で大文字と小文字を区別しないようにする場合は、パラメーターを **true** に設定します。ignorecase のデフォルト値は **false** です。

次に、これらの関数を使用して式を参照できます。

- esp_getExpressionValues(*expr*,*text*)。ここで *expr* は定義済みの式の名前を指定し、*text* は式を適用できるテキストフィールドを指定します
- esp_getExpressionValuesFrom(*expr*,*field*)。ここで *expr* は定義済みの式の名前であり、*field* は入力フィールドの名前です

注: 入力イベントを処理する場合のみ、esp_getExpressionValuesFrom を使用します。

次のコードでは、esp_setExpression は\w+正規表現をコンパイルして、ESP サーバーに保存します。次に、コードは一致を検出するため **words** という名前を使用し、esp_getExpressionValues 関数を通して式をテキスト値に適用します。

```
esp_setExpression("words","\\w+")

local   words,count = esp_getExpressionValues("words","This is a test")

for i=1,count do
   print("word is: "..words[i])
end
```

段落形式の入力があり、その段落を個々の文に分割したいとします。次のように 2 つの関数を使用できます:

```
<window-lua name="code" events="create">
   <schema>
      <fields>
         <field name="id" type="int32" key="true"/>
         <field name="sentence" type="string"/>
      </fields>
   </schema>
   <use/>
   <code><![CDATA[

      esp_setExpression("sentences","[A-z0-9 ]*\\.")

      local   id = 1

      function create(data,context)
```

```
    local   sentences, count = esp_getExpressionValuesFrom("sentences","msg")
    local   events = {}

    for i=1,count
    do
      local   sentence = sentences[i]

      if (string.len(sentence) > 5) then
        local   event = {}
        event.id = id
        event.sentence = sentence
        events[i] = event
        id = id + 1
      end
    end

    return(events)
  end

]]></code>
</window-lua>
```

入力フィールド msg に段落のテキストが含まれている場合、Lua コードは ESP サーバーにコールバックして、段落を個々のイベントの作成に使用される文に分割します。

HTTP リクエストの送信と応答の受信

HTTP リクエストを送信してから、esp_sendHttp(*fields*)関数を使用して、応答を Lua ベースウィンドウの Lua コードに戻します。リクエストオブジェクトをフォーマットし、次の *fields* を使用して関数に渡します。

表 8.11 リクエストオブジェクトフィールド

フィールド	説明
url	リクエストの URL を指定します。
method	使用する HTTP メソッドを指定します(**GET**、**PUT**、**POST**、**DELETE**)。デフォルト値は **GET** です。
headers	サーバーに送信する HTTP リクエストヘッダーを指定します。
body	**PUT** および **POST** の HTTP リクエストの本文を指定します。
tolua	HTTP 応答が JSON または XML の場合、ESP サーバーが応答を返す前に応答を Lua テーブルに変換するために、このフィールドを **true** に設定できます。

フィールド	説明
connect-timeout	サーバーとの接続が正常に確立されるまでのタイムアウトを秒数で指定します。デフォルト値は **0** です(タイムアウトなし)。
data-timeout	サーバーからデータを正常に受信するまでのタイムアウトを秒数で指定します。デフォルト値は **0** です(タイムアウトなし)。

たとえば、次のコードは OpenWeather Weather API からの情報をリクエストします。

```
local   request = {}

request["url"] = "http://api.openweathermap.org/data/2.5/group?
id=4487042&APPID=d0b38229a8f66dac4fbce8bf76be97b4&units=imperial"
request["method"] = "GET"
request["headers"] = {accept="application/json"}

local   response = esp_sendHttp(request)

print(esp_toString(response))
```

そのコードは、API から次のコードのような応答を引き出します。

```
status=200
response={"cnt":1,"list":[{"coord":{"lon":-78.6386,"lat":35.7721},
"sys":{"country":"US","timezone":-14400,"sunrise":1650450904,"sunset":1650498685},
"weather":[{"id":801,"main":"Clouds","description":"few clouds","icon":"02d"}],
"main":
{"temp":40.91,"feels_like":40.91,"temp_min":35.08,"temp_max":46.42,"pressure":1031
,"humidity":89},
"visibility":10000,"wind":{"speed":0,"deg":0},"clouds":{"all":20},
"dt":1650456592,"id":4487042,"name":"Raleigh"}]}
```

tolua フィールドを **true** に設定すると、オブジェクトはすぐに使用できる Lua テーブルを返します。たとえば、次のコードを考えてみましょう。

```
local   request = {}

request["url"] = "http://api.openweathermap.org/data/2.5/group?
id=4487042&APPID=d0b38229a8f66dac4fbce8bf76be97b4&units=imperial"
request["method"] = "GET"
request["headers"] = {accept="application/json"}
request["tolua"] = true

local   response = esp_sendHttp(request)

print(esp_toString(response))
```

そのコードは、次の例のようなものを返します。

```
status=200
response=
    list=
        1=
            clouds=
                all=20
            main=
                temp_max=46.42
                temp_min=35.08
                pressure=1031
                temp=40.91
                humidity=89
                feels_like=40.91
            visibility=10000
            dt=1650456592
            weather=
                1=
                    id=801
                    description=few clouds
                    icon=02d
                    main=Clouds
            sys=
                sunset=1650498685
                timezone=-14400
                sunrise=1650450904
                country=US
            coord=
                lon=-78.6386
                lat=35.7721
            wind=
                speed=0
                deg=0
            name=Raleigh
            id=4487042
    cnt=1
```

デバッグ

概要

診断出力を Lua 関数に直接追加できます。この出力を表示するには 2 つの方法があります。

- コンソールを介して
- ESP サーバーログを介して

コンソールでのデバッグ

Lua print()数を使用して、診断出力をコンソールに出力できます。この機能は、コマンドライン環境で開発し、モデルをコンソールで直接実行する場合に役立ちます。

たとえば、株式取引情報のシーケンスを調べている場合、関数の実行時に現在のイベントをコンテキストと共にダンプできます。関数の最後で、戻り値をダンプできます。

```
function f3(event,context)

    print("in function e3")
    print(esp_toString(event))
    print(esp_toString(context))

    local   code = event.broker == "George"
          and event.price < context.events.john.price
          and event.price < context.events.paul.price

    print("returning: "..code.." from f3")

    return code
end
```

前のモデルを実行すると、次の例のような出力が表示されます。

```
in function e3

        price=50.0
        quantity=50
        broker=George
        id=6
        symbol=IBM


        name=george
        window=1
        events=
            john=
                price=97.34
                quantity=500
                broker=John
                id=0
                symbol=IBM
            paul=
                price=57.34
                quantity=500
                broker=Paul
                id=3
                symbol=IBM

    returning: true from f3
```

イベントとコンテキストをダンプすると、パターンに従って、特定のイベントが特定のパターンをトリガーする理由を判断できます。

ESP サーバーログのデバッグ

esp_logMessage 関数を使用して、Lua から ESP サーバーログにメッセージを記録できます。esp_logMessage 関数は、次のパラメーターを取ります。

表 8.12 esp_logMessage 関数のパラメーター

パラメーター	説明
logcontext	ESP サーバーでコンテキストオブジェクトを指定します(**DF.ESP** など)。このパラメーターは必須です。
message	ログに記録するメッセージを指定します。このパラメーターは必須です。
level	使用するログレベルを指定します。指定されたログコンテキストがそのログレベル以上に設定されている場合、メッセージはログに記録されます。それ以外の場合、メッセージは破棄されます。 level の可能な値は次のとおりです。 ■ **trace** ■ **error** ■ **warn** ■ **fatal** ■ **info** ■ **debug** このパラメーターはオプションで、デフォルトは **info** です。
line	メッセージが記録される Lua コードの行番号を指定します。この値は次の Lua 呼び出しで生成されます。 debug.getinfo(1).currentline line の値を指定しない場合、Lua の行番号はログに記録されません。

イベントを受け取り、データを少し変更してイベントを生成する次の関数を考えてみましょう。"mylogger"ログコンテキスが **debug** 以上に設定されている場合に、コードは受信イベントと生成されたイベントをログに記録します。

```
local   eventId = 1

function create(data,context)

   esp_logMessage("mylogger","creating event
from"..esp_toString(data),"debug",debug.getinfo(1).currentline)

   local   e = {}

   e.id = esp_toString(eventId)
   e.new_string = "\""..data.string.."\""
```

```
    e.new_number = data.number * 3
    e.new_array = {}
    for index,value in ipairs(data.array) do
       e.new_array[index] = value * 2
    end

    eventId = eventId + 1

    esp_logMessage("mylogger","created
event"..esp_toString(e),"debug",debug.getinfo(1).currentline)

    return(e)
end
```

ログ出力は、次の例のようになります。

```
_timestamp=2021-08-23 19:19:48
logName=mylogger
logLevel=Debug
messageContent={81dec492-c3f2-42ba-9cb9-1793ed7fedf7}[XMLServer0001] creating
event from
    id=1
    string=string data
    array=
        1=11
        2=24
        3=55
    number=33
 (Lua line 7)
messageFile=./src/dfESPwindow_lua.cpp
messageLine=774

_timestamp=2021-08-23 19:19:48
logName=mylogger
logLevel=Debug
messageContent={81dec492-c3f2-42ba-9cb9-1793ed7fedf7}[XMLServer0001] created event
    id=1
    new_string="string data"
    new_array=
        1=22
        2=48
        3=110
    new_number=99
 (Lua line 21)
messageFile=./src/dfESPwindow_lua.cpp
messageLine=774
```

パターンウィンドウでパターンマッチングをデバッグするには、次のコードを使用して ESP サーバーログにメッセージを書き込むことができます。

```
function f3(event,context)

    local   message = "in function e3"
    message = message.."\n"..esp_toString(event)
    message = message.."\n"..esp_toString(context)

    local   code = event.broker == "George"
          and event.price < context.events.john.price
          and event.price < context.events.paul.price

    message = message.."\n".."returning: "..esp_toString(code).." from f3"
```

```
    esp_logMessage("mypatternlogger",logmessage,"debug",debug.getinfo(1).currentline)

    return code
  end
```

このコードは、level=debug のログエントリと一致します。ログコンテキストを適切に設定すると仮定すると、ESP サーバーログに次のような内容が表示されます。

```
{
  "_timestamp": "2021-12-16T12:36:43.312000-05:00",
  "logger": "mypatternlogger",
  "loggerLevel": "DEBUG",
  "messageContent": "{f3b6119e-e65c-4ad9-a09b-36075fc43339}[XMLServer0001] in
function e3\n\n    symbol=IBM\n    quantity=50\n    id=6\n    broker=George\n
price=50.0\n\n\n    window=1\n    name=george\n    events=\n        paul=
\n            symbol=IBM\n            quantity=500\n            id=3\n
broker=Paul\n            price=57.34\n        john=\n            symbol=IBM
\n            quantity=500\n            id=0\n            broker=John
\n            price=97.34\n\nreturning: true from f3 (Lua line 23)",
  "messageFile": "dfESPwindow_luabase.cpp",
  "messageLine": "130",
  "thread": "00000034"
}
```

esp_getLogLevel()関数を使用して、ESP サーバー内のロガーの現在のログレベルを取得します。この関数は、名前が指定されていない場合、個々のログコンテキストまたはすべてのコンテキストに関する情報を返します。

数値レベルとレベル名の両方がデータに返されます。

表 8.13 数値レベルとログレベル名

数値	ログレベル
2	Trace
3	Debug
4	Info
5	Warn
6	Error
7	Fatal
8	Off

個々のロガーを取得するには、次のコマンドを入力します。

```
logger = esp_getLogLevel("DF.ESP")
```

データは次のようになります。

```
levelname=info
name=esp.http
level=4}
```

すべてのロガーを取得するには、次のコマンドを入力します。

```
logger = esp_getLogLevel()
```

データは次のようになります。

```
1=
    levelname=error
    name=DF.ESP
    level=6
2=
    levelname=info
    name=DF.ESP.AUTH
    level=4
3=
    levelname=info
    name=common.http
    level=4
...
```

CSV データを直接挿入する

Lua コネクタで esp_injectCsv (data,*options*)関数を使用すると、CSV データを Lua オブジェクトに変換せずに直接挿入できます。

表 8.14 esp_injectCsv 関数のパラメーター

パラメーター	説明
data	CSV データへの参照を指定します。
options	次のフィールドを持つ Lua オブジェクトを指定します: ■ opcode は、データにオペコードが含まれていない場合に使用するオペコードを指定します。 ■ flags は、データにフラグが含まれていない場合に使用するフラグを指定します。 ■ quiesce は、挿入後にプロジェクトを静止するかどうかを決定するオプションのブール変数を指定します。デフォルト値は **false** です。

次のコードは、CSV 形式でブローカーデータを挿入します。

```
<property name="code"><![CDATA[
  local   data = [[
     1012112,Joe,ESP & SAS,SAS Campus Drive Cary NC 27513,919-123-4567
     1012223,Sally,ESP & SAS,SAS Campus Drive Cary NC 27513,919-123-4567
```

```
    1012445,Curt,ESP & SAS,SAS Campus Drive Cary NC 27513,919-123-4567
    1012334,Lisa,ESP & SAS,SAS Campus Drive Cary NC 27513,919-123-4567
    1012001,John,ESP & SAS,SAS Campus Drive Cary NC 27513,919-123-4567
  ]]

  function publish()
    esp_injectCsv(data,{opcode="i",flags="n",quiesce=true})
    return true
  end
]]></property>
```

汎用ユーティリティ 関数

その他の関数

表 8.15　その他の関数

関数	説明
esp_isDone()	ESP サーバーを呼び出して、コネクタが停止しているかどうか(たとえば、サーバーが終了している場合)を確認し、ブール値を返します。 関数が **true** を返す場合、コネクタが ESP サーバーで停止していることを示します。Lua コードは実行ループから抜け出す必要があります。
esp_sleep(*milliseconds*)	Lua コードに戻る前に、ESP サーバーが指定された *milliseconds* 数スリープすることを指定します。これにより、イベントのパブリッシュ方法を制御できます。 関数が **false** を返す場合、コネクタが停止したことを示します。Lua コードは実行ループから抜け出す必要があります。
esp_getUTCOffset(*time*)	指定された *time* の協定世界時(UTC)からのシステムロケールのオフセットを秒単位で取得します。*time* パラメーターはオプションです。時間パラメーターが指定されていない場合、関数は現在の時間を使用します。時間を渡す場合、次のように os.time()を使用して作成できます。 local offset = esp_getUTCOffset(os.time{year=2024,month=7,day=28})

関数	説明
esp_getUTCOffsetHours(*time*)	指定された *time* の協定世界時(UTC)からのシステムロケールのオフセットを時間単位で取得します。*time* パラメーターはオプションです。時間パラメーターが指定されていない場合、関数は現在の時間を使用します。時間を渡す場合、次のようにos.time()を使用して作成できます。 local offset = esp_getUTCOffsetHours(os.time{year=2024, month=7,day=28})
esp_inlist(*item*,*strings*)	*strings* 配列で *item* を探します。 例: e.code = inlist("cary", {"raleigh","durham","chapel hill"})
esp_toString(*object*)	Lua *object* を読み取り可能な文字列に変換します。 function create(data,context) -- Write the contents of the input event print(esp_toString(data)) ... end
esp_getServerProperty(name)	ESP サーバーからサーバーレベルプロパティの値を取得します。これらは、サーバーコマンドライン、esp-properties.yml、または環境で指定された値です。たとえば、REST 要求を送信して、実行しているサーバーから ESP プロジェクトとスキーマを取得する場合: local host = esp_getServerProperty("HOST") local port = esp_getServerProperty("http") local url = "http://"..host..":"..port.."/eventStreamProcessing/v2/projects?sche local request = {} request["url"] = url request["method"] = "GET" request["headers"] = {accept="application/json"} request["tolua"] = true local response = esp_sendHttp(request) print(esp_toString(response))
esp_getProjectProperty(name,[project])	プロジェクトレベルプロパティの値を取得します。プロジェクトパラメーターはオプションです。プロジェクト名を指定せず、サーバー内にプロジェクトが 1 つしかない場合は、そのプロジェクトが使用さ

関数	説明
	れます。これらの値は、プロジェクト XML で定義されています。 <project> <properties> <property name="property_1">value 1</property> <property name="property_2">value 2</property> </properties> ... </project>

イベントコンテキスト関数

次の関数は、Lua コードが ESP イベントのコンテキストで実行される場合に役立ちます。これらの関数を使用して、そのイベントの演算コードとフラグ値を取得できます。

表 8.16 イベントコンテキスト関数

関数	説明
esp_getEventOpcode()	現在のイベントの演算コードを返します。返される値は、次のいずれかです。 ■ **I** - 挿入 ■ **U** - 更新 ■ **D** - 削除 ■ **P** - アップサート ■ **SD** - 安全な削除 ■ **UB** - ブロックの更新 ■ **N** - 操作なし
esp_getEventFlags()	現在のイベントのイベントフラグを返します。返される値は、次のいずれかです。 ■ **N** - 標準 ■ **P** - 一部更新

日付と時間の関数

表 8.17 日付と時間の関数

関数	説明
esp_dateMath(*date*,*unit*,*value*)	指定された *date* でカレンダー計算を実行し、結果をエポック日付として返します。*unit* の *value* は、基準日の値に追加されます。有効なユニット値は次のとおりです。**second**,**seconds**、**minute**、**minutes**、**hour**、**hours**、**day**、および **days**。 例: e.seconds = esp_dateMath(os.time(),"days",-3) このインスタンスは、現在の日付から 3 日を減算します。
esp_formatDate(*date*,*format*)	指定された *format* で *date* をフォーマットします。これは、UNIX strftime 関数でサポートされている任意の日付形式にすることができます。 *format* が指定されていない場合、デフォルトのシステム形式が使用されます。 例: e.datestring = esp_formatDate(os.time())
esp_formatTime(*time*,*format*)	指定された *format* で *time* をフォーマットします。これは、UNIX strftime 関数でサポートされている任意の時間形式にすることができます。

関数	説明
	format が指定されていない場合、デフォルトのシステム形式が使用されます。 例: e.timestring = esp_formatTime(os.time())
esp_getSystemMicro()	1970 年 1 月 1 日から現在時刻までのマイクロ秒数を返します。
esp_getSystemMilli()	1970 年 1 月 1 日から現在時刻までのミリ秒数を返します。
esp_parseDate(*datestring*,*format*)	指定された *format* に従って *datestring* を解析し、エポック日付の値を返します。*Format* は、UNIX strftime 関数でサポートされている任意の時間形式にすることができます。 例: e.date = esp_parseDate("2022-04-21","%Y-%m-%d")
esp_parseTime(*timestring*,*format*)	指定された *format* に従って *timestring* を解析し、エポック時間の値を返します。*Format* は、UNIX strftime 関数でサポートされている任意の時間形式にすることができます。 例: e.time = esp_parseTime("2022-04-21T04:09:40","%Y-%m-%dT%H:%M:%S")
esp_timeMath(*time*,*unit*,*value*)	指定された *time* でカレンダー計算を実行し、結果をエポック時間として返します。*unit* の *value*

関数	説明
	が基準時間値に加算されます。有効なユニット値は次のとおりです。**second**,**seconds**、**minute**、**minutes**、**hour**、**hours**、**day**、および **days**。 例: e.seconds = esp_timeMath(os.time(),"hours",6) このインスタンスは、現在の時間に 6 時間を追加します。

文字列関数

Lua のサポートは、マルチバイト文字を含む文字列に対する操作に制限されています。ESP サーバーは、esp_strings 関数を通じて追加のサポートを提供します。

表 8.18 文字列関数

関数	説明
esp_strings({func="*function*",value=*value*})	入力値と、それに対して実行する機能を指定します。ESP サーバーは、文字列値に対して指定された操作を実行します。 サポートされている関数は次のとおりです。 ■ to_lower 文字列を小文字に変換します ■ to_upper 文字列を大文字に変換します この戻り値は、次のフィールドを含む Lua テーブルです: ■ value 操作後の文字列の値です(エラー発生時は **nil**) ■ error 問題が発生した場合にその原因を示します 例: local utf8_string = "zaÅ¼Ã³Ä‡ gÄ™Å›lÄ... jaÅºÅ„" local e = {} e.id = 1 e.utfstr = esp_strings({func="to_upper",value=utf8_string}).value ...

Lua プロパティの操作

概要

Lua プロパティを使用して、SAS Event Stream Processing プロジェクト内の Lua エンティティ間でコミュニケーションできます。Lua プロパティは、ESP サーバーに格納される名前と値のペアです。これらの値は Lua *コンテキスト*に保存されます。通常、デフォルトのコンテキストは ESP サーバーによって自動的に作成されるため、プロパティにアクセスするときにコンテキストを指定する必要はありません。ただし、Lua プロパティへのアクセスをより効率的にできると思われる場合はいつでも、複数のコンテキストを作成できます。

次のコードは、2 つの異なるコンテキストで myproperty の値を設定します。

```
esp_setProperty({name="myproperty",value="one"})
esp_setProperty({name="myproperty",value="one",context="newcontext"})
```

最初の呼び出しでは、デフォルトの Lua コンテキストに myproperty が設定されます。2 番目の呼び出しでは、newcontext という名前のコンテキストにプロパティを設定します。どちらの場合も、呼び出しによりプロパティの値がハードコードされた文字列 **one** に設定されます。変数の値を参照してプロパティの値を設定するには、引用符を省略します。

すべての Lua エンティティは、プロパティ値を読み取ることができます。

ESP サーバーは、Lua オブジェクトに関連付けられたプロパティをロックする必要がありますが、プロパティを取得するには READ ロックのみが必要です。プロパティを所有する Lua オブジェクトがプロパティを変更するたびに、プロパティは WRITE ロックでロックされます。ロックにより、プロパティの操作効率が向上します。

Lua プロパティの値は、ESP サーバーに保存される Lua オブジェクト(文字列、数値、テーブルなど)です。

プロパティの設定

プロジェクト内の任意の Lua コードから Lua プロパティを設定できます。esp_setProperty 関数を使用して、Lua プロパティを設定します。

表 8.19 esp_setProperty 関数

構文	説明
esp_setProperty({name=*name*,value=*value*[,field=*field*][,context=*context*][,persist=true | false]})	name には、Lua プロパティの名前を指定します。 value には、Lua プロパティ(任意の Lua オブジェクト)の値を指定します。 field パラメーターを使用して、Lua プロパティのフィールドを設定します。 context パラメーターを使用して、プロパティを設定する Lua コンテキストを指定します。 プロパティを永続的にするには、persist=true のように設定します。永続的なプロパティは、設定時にファイルシステムに保存されます。ESP サーバーは、これらの保存された値を使用して、起動時にプロパティ値を初期化します。

Lua プロパティ内のフィールドを指定するには、ドット表記を使用します。たとえば、次の従業員情報が Lua プロパティとして格納されているとします。

```
local   employees = {}

employees.Jose =
{department="Marketing",position="Marketer",info={vacation_days=30,years_of_service=8}
}
employees.Sally = {department="Research and
Development",position="Developer",info={vacation_days=30,years_of_service=10}}
employees.Miko = {department="Research and
Development",position="Tester",info={vacation_days=10,years_of_service=3}}
employees.Helen = {department="Support",position="Customer Support
Engineer",info={vacation_days=24,years_of_service=30}}

esp_setProperty({name="employee_data",value=employees})
```

次の Lua コードは、Jose の休暇日を設定します。

```
esp_setProperty({name="employee_data",value=25,field="Jose.info.vacation_days"})
```

次の Lua ウィンドウは、ハートビートを使用して Lua プロパティを設定します。

```
<window-lua name="lua" events="create" heartbeat="hb">
 <schema>
  ...
 </schema>
  <code><![CDATA[

     function create(data,context)
        ...
     end

     function hb()
        local   info = {}
        info.foo = "bar"
        info.num = 100
```

```
        esp_setProperty({name="info",value=info})
      end

    ]]></code>
  </window-lua>
```

esp_setProperty 関数は、プロパティが正常に設定された場合は **true** を返し、プロパティの設定中にサーバーでエラーが発生した場合は **false,err** を返します。通常、エラーをキャッチするための呼び出しは次のようになります。

```
local   code,err = esp_setProperty({name="product",value="Esp"})

if (code == false)
then
   print("cannot set property: "..tostring(err))
end
```

プロパティの取得

すべての Lua エンティティは、esp_getProperty 関数を使用してサーバーから Lua プロパティを取得できます。

表 8.20 esp_getProperty 関数

構文	説明
esp_getProperty({name=*name*, [field=*field*][,context=*context*]})	name には、取得するプロパティの名前を指定します。field パラメーターを使用して、取得するフィールドを指定します。context パラメーターを使用して、プロパティの取得元となる別の Lua コンテキストを指定します。

この関数は、Lua プロパティの値を返すか、プロパティが見つからない場合は **nil** を返します。エラーが発生した場合(たとえば、ESP サーバーがエンティティを 見つけられない場合)、2 番目のパラメーターとして **nil** が返されます。

```
local value,err = esp_getProperty({name="foo"})
if err ~= nil then
   ...
end
```

前のトピックで提供された従業員情報の Lua プロパティ全体を取得したいとします。また、Lua エンティティの名前は test だとします。次のコードを考えてみましょう。

```
local   data,err = esp_getProperty({name="employee_data"})
print(esp_toString(data))
```

次の出力が生成されます。

```
Helen=
        info=
            years_of_service=30.0
            vacation_days=24.0
        department=Support
        position=Customer Support Engineer
    Jose=
        info=
            years_of_service=8.0
            vacation_days=30.0
        department=Marketing
        position=Marketer
    Miko=
        info=
            years_of_service=3.0
            vacation_days=10.0
        department=Research and Development
        position=Tester
    Sally=
        info=
            years_of_service=10.0
            vacation_days=30.0
        department=Research and Development
        position=Developer
```

Sally 専用に info フィールドが必要だとします。次のコードを考えてみましょう。

```
local   data,err = esp_getProperty({name="employee_data",field="Sally.info"})
```

次の出力が生成されます。

```
years_of_service=10.0
vacation_days=30.0
```

プロパティのクリア

esp_clearProperty は、Lua コンテキストからプロパティを削除します。

表 8.21　esp_clearProperty 関数

構文	説明
esp_clearProperty({name=*name*[,field=*field*][,context=*context*][,persist=true | false]})	name を使用して、クリアするプロパティの名前を指定します。field パラメーターを使用して、クリアするプロパティのフィールドを指定します。context を使用して、プロパティをクリアする別の Lua コンテキストを指定します。persist を使用して、ディスクからプロパティ 値を削除するかどうかを指定します。

esp_clearProperty 関数は、プロパティが正常にクリアされた場合は **true** を返し、プロパティをクリアしようとしたときに ESP サーバーでエラーが発生した場合は

false,err を返します。通常、エラーをキャッチするための呼び出しは次のようになります。

```
local   code,err = esp_clearProperty({name="product"})

if (code == false)
then
   print("cannot clear property: "..tostring(err))
end
```

プロパティリスナー

esp_addPropertyListener 関数を使用すると、プロパティの変更を登録できるため、esp_getProperty()を継続的に呼び出す必要はありません。

表 8.22　esp_addPropertyListener 関数

構文	説明
esp_addPropertyListener({name="*property*", change="*function*", [clear="*function*"], [context=*context*], [init="true| false"]})	name パラメーターを使用して、登録する *property* の名前を取得します。*を指定して、すべてのプロパティを 取得できます。 change パラメーターを使用して、イベントが変更されたときに呼び出される Lua 関数の名前を指定します。次のように*関数*を指定します。 function myfunc(*property*) ここで、*property* は次のフィールドを持つ Lua オブジェクトを参照します: ■ *name* は、関連するプロパティの名前を指定します ■ *value* は、プロパティの値を指定します ■ *field* は、変更を受け取りたいプロパティ内のフィールドを指します(該当する場合) ■ *context* は、プロパティが存在する別の Lua コンテキストの名前を指定します clear パラメーターを使用して、イベントがクリアされたときに呼び出される Lua 関数の名前を指定します。関数のシグネチャは value フィールドが存在しないことを除いて、change 関数と同じです。

構文	説明
	context パラメーターを使用して、プロパティの別の Lua コンテキストを指定します。 init を使用して、プロパティ値の初期ダウンロードが必要かどうかを示します。**true** に設定すると、要求されたプロパティに対してコールバックが呼び出されます。デフォルト値は **false** です。

プロパティリスナーに異なるコンテキストを設定した場合の結果に注意してください。たとえば、次のコードを考えてみましょう。

```
esp_addPropertyListener( {name="ITEMS_RAF234_1_1",change="setFilterCriteria",context="FILTER"} )
```

このコードでは、リスナーは esp_setProperty が *FILTER* コンテキストを指定した場合にのみ呼び出されます。

```
esp_setProperty( {name="ITEMS_RAF234_1_1",value=data.FILTER_CONDITION,context="FILTER"} )
```

次のコードは、デフォルトのコンテキストに登録されているリスナーにのみ通知します。

```
esp_setProperty( {name="ITEMS_RAF234_1_1",value=data.FILTER_CONDITION} ).
```

この場合、プロパティリスナーで 指定された setFilterCriteria 変更関数は呼び出されません。

5 分ごとにファイルからトークンを読み取る Lua スニペットがあるとします。

```
<lua-snippet name="access_token">
  <lua-thread func="readToken" interval="5 minutes"/>
  <code><![CDATA[

    function readToken()
      local   f = io.open("token.txt","r")
      local   token = f:read("*all")
      f:close()
      esp_setProperty({name="token",value=token})
    end

  ]]></code>
</lua-snippet>
```

次の Lua コネクタは、そのトークンを使用して HTTP リクエストを送信します。

```
<connector class="lua" name="pub">
 <properties>
  <property name="type">pub</property>
  <property name="code"><![CDATA[

    local   token = nil

    function setToken(property)
      token = property.value
```

```
    end

    esp_addPropertyListener({name="token",change="setToken",init=true})

    function publish()

      -- use token to make REST request, parse response, and inject events

      local  request = {}

      request["url"] = "http://somedata.org"
      request["method"] = "GET"
      request["headers"] = {Authorization="Bearer "..token}

      local  response = esp_sendHttp(request)
      ...
    end

  ]]></property>
 </properties>
</connector>
```

Lua スニペットがファイルからトークンを読み取り、プロパティを設定するたびに、setToken 関数が呼び出され、HTTP リクエストで使用する新しい値が保存されます。特定のフィールドがプロパティに設定されると、そのフィールドは ESP サーバーによって配信されます。

次のハンドラー関数を記述したとします。

```
function handler(property,value,field)
  print("employee data changed, field "..property.field)
  print(esp_toString(property.value))
end
```

プロパティを変更できるようになりました(Miko をマネージャーにして、より多くの休暇を与えます)。

```
...
local  markdata = esp_getProperty({name="test",field="Miko"})
markdata.info.position = "Manager"
esp_setProperty({name="employee_data",value=markdata,field="Miko"})
...
```

ハンドラー関数を呼び出すと、次の出力がコンソールに表示されます。

```
employee data changed, field Miko

info=
    position=Manager
    years_of_service=3.0
    vacation_days=30.0
department=Research and Development
position=Tester
```

永続的なプロパティ

永続的なプロパティは、設定時にファイルシステムに保存されます。ESP サーバーは、これらの保存された値を使用して、起動時にプロパティ値を初期化します。

Lua ウィンドウまたは Lua スニペットで永続プロパティを有効にするには、の財産 esp_setProperty 関数または esp_clearProperty 関数の persist プロパティを設定します。

```
esp_setProperty({name="myproperty",value="foo",persist=true})
esp_clearProperty({name="myproperty",persist=true})
```

プロパティは、プロジェクトの Lua ルートディレクトリから派生したディレクトリに格納されます。Lua ルートディレクトリは、次のいずれかの方法を優先順に使用して指定できます。

- プロジェクトルート要素の属性:

  ```
  <project
  name="myproject" threads="4" luaroot="/mnt/data/lua_root">
  ```

- サーバーのコマンドラインオプション: $ dfesp_xml_server ... -luaroot /mnt/data/lua_root
- Esp-properties.yml ファイルのサーバーセクションのエントリ: server: luaroot: /mnt/data/lua_root

プロパティは、指定された Lua ルートディレクトリの properties サブディレクトリ下に階層的に格納されます。

```
properties/
 context/
    property1
    property2
    property...
    propertyN
```

各プロパティは、そのプロパティ名で独自のファイルに格納されます。

たとえば、デバイス統計を生成し、プロパティ last_device および info を設定する Lua ウィンドウがあるとします。

```
<window-lua name="lua" events="create">
 <schema>
  <fields>
     <field name="device_id" type="string" key="true"/>
     <field name="description" type="string"/>
     <field name="value" type="int32"/>
     <field name="unit" type="string"/>
  </fields>
 </schema>
 <code><![CDATA[

  function create(data)

     esp_setProperty({name="last_device",value=data,persist=true})
```

```
    local   info = {}
    info.last_device = data.device_id
    info.timestamp = string.format("%s",os.date())

    esp_setProperty({name="info",value=info,persist=true})

    return data
  end

]]></code>
</window-lua>
```

次の構造が Lua ルートの下に存在します。

```
properties/
  default/
    info
    last_device
```

これらのプロパティは、JSON オブジェクトとして格納されます。

```
{"description":"Measures height","device_id":"2","unit":"feet","value":7}
```

ESP サーバーは起動時に、Lua ルートディレクトリの下に存在するプロパティファイルを検索します。これらのプロパティファイルを読み取って適切な Lua コンテキストにロードし、サーバーの起動時に使用できるようにします。

Lua スニペットからの関数の呼び出し

esp_callLua スニペット内で定義された関数を呼び出す関数を使用できます。

```
esp_call({name="snippet_name",func="function_to_call"},parameter,[parameter,...])
```

最初のパラメータについては、esp_call 関数では、次のフィールドを含む Lua テーブルを指定します。

- name- 参照する Lua スニペットの名前を識別します。
- func- 呼び出すスニペット内の関数の名前を指定します。

残りのパラメータは、指定された関数にデータとして送信されます。

戻り値は、次のフィールドを含む Lua テーブルです。

- code- 関数が正常に実行されたかどうかを示すブール値。
- results- 関数からの戻り値の Lua テーブル(配列)(成功した場合)。
- error- エラーの説明(呼び出しが失敗した場合)。

次のスニペットを考えてみましょう:

```
<lua-snippet name="math">
 <code><![CDATA[

    function divide(dividend,divisor)

      if (divisor == 0)
      then
```

```
            error("division by 0, "..tostring(dividend).."/"..tostring(divisor))
        end

        return dividend / divisor
    end

    ]]></code>
</lua-snippet>
```

次のように divide 関数をコールできます。

```
function create(data)

    local   e = {}
    local   dividend = data["dividend"]
    local   divisor = data["divisor"]

    local   response = esp_call({name="math",func="divide"},dividend,divisor)

    if (response.code == true)
    then
        e["quotient"] = response.results[1]
    else
        e["error"] = response.error
    end

    return(e)
end
```

0 による除算がある場合、応答の code フィールドが **false** になり、応答の error フィールドにエラーが示されます。それ以外の場合、code フィールドが **true** になり、results フィールドには商を提供する単一のエントリが含まれます。

パブリッシャーアダプターの管理

概要

一連の Lua 関数を使用してパブリッシャーアダプターを管理し、データをプロジェクトにストリーミングできます。これらの機能を使用することは、同じ目的でアダプターコネクタを使用する代わりの実行可能な方法となります。この関数により、次のアクションが可能になります。

- サポートされている種類のアダプターを作成します。
  - Audio(audio)
  - Bacnet(bacnet)
  - SAS Cloud Analytic Services(cas)
  - Database(db)
  - Azure Event Hubs(eventhubs)

  - File and Socket(fs)
  - Kafka(kafka)
  - Kinesis(kinesis)
  - IBM WebSphere MQ(mq)
  - MQTT(mqtt)
  - OPC-UA(opcua)
  - PI Web(piweb)
  - QuasarDB(quasardb)
  - RabbitMQ(rmq)
  - Solace(sol)
  - Video Capture(videocap)
- アダプターを開始または再起動する
- アダプターを停止する
- アダプターのパラメーターを設定する
- アダプターを削除します

アダプターの詳細については、"Overview of Adapters" (*SAS Event Stream Processing: Connectors and Adapters*)を参照してください。

> 重要　これらの関数は、Lua スニペットを通じてのみ利用できます。

コールバックの使用

Lua コードは、次のコールバック関数を通じてアダプターと非同期に対話できます。

- adapter_started(*adapter*)- アダプターの開始時に呼び出されます
- adapter_stopped(*adapter*, status)- アダプターが停止したときに呼び出されます。status パラメーターは、アダプターの終了ステータスです。
- adapter_stdout(*adapter*,text)- アダプターが標準出力に書き込むときに呼び出されます。
- adapter_stderr(*adapter*,text)- アダプターが標準エラーに書き込むときに呼び出されます。
- adapter_error(*adapter*,errno)- アダプターの開始で問題が発生した場合に呼び出されます。errno パラメーターはシステムエラー番号です。

Lua コードでコールバック関数を定義すると、それらは適切なタイミングで呼び出されます。

アダプターの作成

次の関数を使用してアダプターを作成します。

```
local adapter = esp_adapter_create({class="type",name="name",parms={params})
```

この関数は、単一の Lua オブジェクトをパラメーターとして受け取ります。このオブジェクトは次のフィールドをサポートします。

- *type*- アダプターの種類を指定します(例: db、fs、等々)。
- *name*- インスタンスのオプションの名前を指定します。
- parms - アダプターパラメーター(-C コマンドラインパラメーターでアダプターに渡される名前と値のペア)を含む Lua オブジェクト params を指定します。

たとえば、次のコードは myadapter という名前のファイルおよびソケット(fs)アダプターを作成します(ただし、開始はしません):

```
local   parms = {
   fstype="csv",
   url="dfESP://@HOSTNAME@:@pubsub@/p/cq/s"
}

adapter = esp_adapter_create({class="fs",name="myadapter",parms=parms})
```

注: ここで、ESP サーバーは HOSTNAME をアダプターが実行されているホストで解決し、pubsub を ESP サーバーが使用するパブリッシュ/サブスクライブのポートに接続します。

この関数の戻り値は、後続のアダプター関数呼び出しのパラメーターとして使用される Lua オブジェクトです。

エラーが発生すると、応答のコードフィールドが**-1** に設定され、エラーメッセージが送信されます。

```
message=Adapter type myadapter is not supported
code=-1
```

アダプターの開始

次の関数を使用してアダプターを開始します。

```
local response = esp_adapter_start(adapter)
```

adapter パラメーターは、esp_adapter_create 関数からの戻り値です。

応答は、コードとオプションでメッセージ フィールドを含む Lua オブジェクトです。アダプターが現在実行されていない場合、サーバーはアダプターを開始しようとし、コード 0 を返します。そうしないと、次のエラーコード-1 が表示されます。

adapter_started コールバック関数を定義した場合、プロセスが正常に開始されたときに呼び出されます。

アダプターの停止

次の関数を使用してアダプターを停止します。

local response = esp_adapter_stop(*adapter*)

adapter パラメーターは、esp_adapter_create 関数からの戻り値です。

応答は、コードとオプションでメッセージフィールドを含む Lua オブジェクトです。アダプターが現在実行されてい場合、サーバーはアダプターを停止しようとし、コード 0 を返します。そうしないと、次のエラーコード-1 が表示されます。

adapter_stopped コールバック関数を定義した場合、プロセスが正常に停止されたときに呼び出されます。

アダプターの再起動

アダプターを再起動するには、次の関数を使用します。

local response = esp_adapter_restart(*adapter*)

adapter パラメーターは、esp_adapter_create 関数からの戻り値です。再起動を実行するためにアダプターが現在実行されている必要はありません。アダプターが実行中の場合、ESP サーバーはアダプターを停止してから開始しようとします。それ以外の場合、ESP サーバーはアダプターの開始のみを試みます。アダプターの状態に応じて、適切なコールバック関数が呼び出されます。

応答は、コードとオプションでメッセージフィールドを含む Lua オブジェクトです。アダプターが正常に再起動されると、コード 0 が表示されます。そうしないと、ERROR メッセージとしてエラーコード-1 が表示されます。

アダプターパラメーターの設定

アダプターのパラメーターはいつでも設定できます。ただし、パラメーター値は、アダプターが起動または再起動されるまで有効になりません。

次の関数を使用して、アダプターのパラメーターを設定します。

local response = esp_adapter_set(*adapter*,{parm1="*a*",parm2="*b*",...})

応答は、コードとオプションでメッセージフィールドを含む Lua オブジェクトです。アダプターのパラメーターが正常に設定されると、コード 0 が得られます。そうしないと、ERROR メッセージとしてエラーコード-1 が表示されます。

アダプターの削除

アダプターを削除するには、アダプターが実行されていない必要があります。

アダプターを削除するには、次の関数を使用します。

```
local response = esp_adapter_remove(adapter)
```

adapter パラメーターは、 esp_adapter_create 関数からの戻り値です。

応答は、コードとオプションでメッセージフィールドを含む Lua オブジェクトです。アダプターが正常に削除されると、コード 0 が表示されます。そうしないと、ERROR メッセージとしてエラーコード-1 が表示されます。

Lua コンポーネントからのリレーショナルデータベースへのアクセス

概要

Lua コンポーネントから ODBC ドライバーを介してリレーショナルデータベースにアクセスできます。

ODBC ドライバーを使用する場合は、システムデータソース名(DSN)を介してデータベースへの接続を確立します。この DSN および関連するデータベースユーザー資格情報は、必須の構成パラメーターです。DSN の一般的な形式は次のとおりです。

```
DSN=data_source[;attribute=value[;attribute=value]...]
```

データベース接続に使用する目的のワイヤプロトコルを指定する *data_source* を指定する必要があります。指定したデータソースに対して、1 つ以上の関連属性を列挙します。ほとんどのワイヤプロトコルでは、データベースにアクセスするために *ユーザー ID* および *パスワード* を指定する必要があります。多くのワイヤプロトコルでは、データベースの場所を識別する *ホスト名* を指定することも必要です。

data_source に指定した各属性は、SAS Event Stream Processing で提供される odbc.ini ファイルに設定された値をオーバーライドします。

詳細については、“Using DataDirect ODBC Drivers” (*SAS Event Stream Processing: Connectors and Adapters*)を参照してください。

データベース接続は Lua コンテキストに保存されます。ほとんどの場合、デフォルトのコンテキストは ESP サーバーによってに作成されるため、接続を作成するときにコンテキストを指定する必要はありません。

重要 Lua コンポーネントは、次のリレーショナルデータベースへのアクセスをサポートしています。

- Db2
- Gleenplum
- MySQL
- Oracle
- PostgreSQL

サポートされていないデータベースはご自身の判断で使用してください。

データベースへの接続

データベースに接続するには、esp_sqlConnect 関数を使用します。次に例を示します:

```
local code,err = esp_sqlConnect(options)
```

この関数の **options** は、次のフィールドを含む Lua オブジェクトです。

- name は、SQL 接続の名前です。
- connstr は、DSN です。ここに説明されています。
- pwdencrypted は、パスワードを暗号化するかどうかを指定するオプションのブール値です。
- context は、接続を作成する Lua コンテキストを指定するオプションの値です。

接続の名前は、後続のデータベース呼び出しで使用されます。

esp_sqlConnect 関数は、1 つまたは 2 つの値を返します。最初に返される値は、接続が成功したかどうかを示すブール値です。接続が失敗した場合は、2 番目の値を通じて失敗の理由が返されます。

たとえば、次のコードは、PostgreSQL データベースに接続する方法を示しています:

```
local connstr = "DSN=PostgreSQL Wire
Protocol;UID=postgres;PWD=mypass123;servername=myserver;port=4444;database=espd
emo" local code,err = esp_sqlConnect({name="db",connstr=connstr})
```

接続が完了したら、esp_sqlDisconnect 関数を私用して接続を削除できます。

```
local code,err = esp_sqlDisconnect(options)
```

この関数の **options** は、次のフィールドを含む Lua オブジェクトです。

- name は、切断する SQL 接続の名前です。
- context は、接続が存在する Lua コンテキストを指定するオプションの値です。

esp_sqlDisconnect 関数は、1 つまたは 2 つの値を返します。最初に返される値は、切断が成功したかどうかを示すブール値です。切断が失敗した場合は、2 番目の値を通じて失敗の理由が返されます。

次のコードは、Postgres 接続を切断する方法を示しています:

```
local code,err = esp_sqlDisconnect({name="db"})
```

データの取得

データベースからデータを取得するには 2 つの方法があります。1 つ目の方法は、SQL select ステートメントを ESP サーバーに送信することによって、SQL クエリからすべての 結果を取得します。

```
local resp = esp_sqlSelect(options)
```

このステートメントにおいて、**options** は、次のフィールドを含む Lua オブジェクトです。

- conn は、使用する SQL 接続の名前です。
- query は、SQL select ステートメントです。
- context は、接続が存在する Lua コンテキストを指定するオプションの値です。

この戻り値は、次のフィールドを含む Lua テーブルです:

- code は、クエリが成功したかどうかを示すブール値です。
- data は、データベースのデータです。
- message は、クエリが失敗したときに送信される情報メッセージです。

esp_sqlSelect は、中程度のサイズのクエリで使用できます。たとえば、何百万もの株式取引が取引テーブルに保存されているとします。次のコードは、非常に大規模な取引(100 万株以上)を取得する方法を示しています。

```
local resp = esp_sqlSelect({conn="db",query="select * from trades where quant >
1000000"})

print(esp_toString(resp))
```

```
code=true
data=
  1=
    broker=101667
    buyer=0
    msecs=724
    symbol=XEL
    buysellflg=1
    price=21.7
    seller=8123541
    currency=87236
    quant=1286200
    id=18564845
    time=1280928728
    venue=55555
  2=
```

```
broker=101667
buyer=1012223
msecs=418
symbol=C
buysellflg=0
price=4.15
seller=8382739
currency=87236
quant=2063400
id=2669875
time=1280928609
venue=55555
```

ただし、1 回のパスですべての取引を選択すると、メモリを大量に消費します。データベースから大量のデータを取得する必要がある場合は、データを段階的に取得することをお勧めします。次の関数を使用してクエリを開き、データを段階的に取得します。

- esp_sqlOpen - クエリを開きます。
- esp_sqlNext - 結果から指定された数の行を取得します。
- esp_sqlClose - クエリを閉じます。

次のように esp_sqlOpen 関数を指定します:

```
local handle = esp_sqlOpen(options)
```

この関数の **options** は、次のフィールドを含む Lua オブジェクトです。

- conn は、ESP サーバーへの SQL 接続の名前です。
- query は、SQL select ステートメントです。
- context は、接続が存在する Lua コンテキストを指定するオプションの値です。

この戻り値は、次のフィールドを含む Lua テーブルです:

- code は、クエリが成功したかどうかを示すブール値です。
- message は、クエリが失敗した場合に送信される情報メッセージです。

クエリを正常に開くことができたら、esp_sqlNext 関数を使用してデータを取得できます。

```
local data,count = esp_sqlNext(handle,numrows)
```

- handle は、esp_sqlOpen から返される値です。
- numrows は、返される行の数を指定します。

numrows パラメーターを指定しない場合は、結果の残りの行が返されます。

この関数は 2 つの値を返します。最初の値はクエリ結果のデータ、またはエラーが発生した場合は **nil** です。2 番目の値は返されるレコードの数で、エラーが発生した場合、またはデータが使い果たされた場合は **nil** になります。関数がカウント 0 を返すたびに、データは使い果たされます。その場合、サーバーはクエリを閉じるため、esp_sqlClose を呼び出す必要はありません。

クエリが終了したら、esp_sqlClose 関数を使用してサーバーからクエリを削除します。

```
local code,err = esp_sqlClose(handle)
```

この関数は 2 つの値を返します。最初の値は、終了が成功したかどうかを示すブール値です。2 番目の値には、失敗の原因に関する情報を提供するエラーメッセージが含まれています。

前述したように、すべての行を取得したら、ESP サーバーがクエリを閉じる必要はありません。

データの挿入

データベースからレコードを挿入するには、esp_sqlInsert 関数を使用します。

```
local resp = esp_sqlInsert(options)
```

この関数の **options** は、次のフィールドを含む Lua テーブルです:

- conn は、使用する SQL 接続の名前です。
- table は、データを挿入するデータベーステーブルです。
- data は、挿入するレコードを含む 1 つ以上の Lua オブジェクトが含まれます。
- context は、接続が存在する Lua コンテキストを指定するオプションの値です。

株式取引を処理している場合は、ソースウィンドウで Lua サブスクライバーコネクタを使用して受信取引を取得し、データベースに書き込むことができます。

```
<window-source name="trades" insert-only="true">
  <schema>
    <fields>
      <field key="true" name="id" type="int64"/>
      <field name="symbol" type="string"/>
      <field name="currency" type="int32"/>
      <field name="time" type="int64"/>
      <field name="msecs" type="int32"/>
      <field name="price" type="double"/>
      <field name="quant" type="int32"/>
      <field name="venue" type="int32"/>
      <field name="broker" type="int32"/>
      <field name="buyer" type="int32"/>
      <field name="seller" type="int32"/>
      <field name="buysellflg" type="int32"/>
    </fields>
  </schema>
  <connectors>
    <connector class="lua" name="sub">
      <properties>
        <property name="type">sub</property>
        <property name="code"><![CDATA[
          function subscribe(data)
            local   response = esp_sqlInsert({conn="db",table="trades",data=data})
            if (response.code == false)
            then
              print(esp_toString(response))
            end
          end
        ]]></property>
```

```
        </properties>
      </connector>
    </connectors>
  </window-source>
```

esp_sqlInsert からの戻り値は、次のフィールドを含む Lua オブジェクトです。

- code は、挿入のステータスです。成功の場合は**真** 、失敗の場合は**偽**になります。
- message は、障害が発生した場合の障害に関する情報が含まれています。
- object は、失敗したレコードが発生したときのデータです。

データの更新

データベースを更新には、esp_sqlUpdate 関数を使用します。

```
local resp = esp_sqlUpdate(options)
```

この関数の **options** は、更新プロパティを含む Lua テーブルです。

Lua コンポーネントを通じてデータベース更新するには 2 つの方法があります。1 つ目の方法は、更新するフィールドを含むデータのみを送信することです。このアプローチを使用する場合、次のフィールドが options パラメーターで受け入れられます。

- conn は、使用する SQL 接続の名前です。
- table は、更新するデータベーステーブルです。
- data は、更新を含む 1 つ以上の Lua オブジェクトが含まれます。
- upsert は、レコードが存在しない場合にレコードの挿入を試みるオプションのブール値です。デフォルト値は **false** です。
- context は、接続が存在する Lua コンテキストを指定するオプションの値です。

サーバーはデータからデータベースキーを導出するため、データには各オブジェクトの少なくともすべてのキーフィールドが 含まれている必要があります。この更新メソッドは、関数呼び出しで複数の Lua オブジェクトを受け入れることができます。各オブジェクトはデータベースに対して個別の更新リクエストを生成します。

たとえば、株式取引を処理していて、各シンボルの最低価格と最高価格を保存したいとします。次のテーブルを作成します。

```
create table trades_data
(
  symbol text,
  min_price float,
  max_price float,
  range float,
  primary key (symbol)
);
```

次のコードストリームは、各シンボルの現在の最低価格と最高価格を取引して維持します:

```
local   resp = esp_sqlUpdate({conn="db",
                              table="trades_data",
```

```
                upsert=true,
                data={
                 {symbol="ABC",min_price=100,max_price=120,range=20},
                 {symbol="XYZ",min_price=10,max_price=20,range=10}
                }})
```

upsert 値は**真**であることに注意してくださいその結果、レコードが存在しない場合、サーバーはレコードを追加します。

更新を実行する 2 番目の方法は、更新するデータベースレコードを識別する SQL WHERE 句を含む単一の Lua オブジェクトを追加することです。次のコードでは、リクエストのデータが WHERE 句に一致するすべてのレコードに適用されます。

```
local   resp = esp_sqlUpdate({conn="db",
                table="trades_data",
                data={min_price=0,max_price=0,range=0},
                where="1=1"})
```

データ内のキーフィールドは無視されます。前の例の trades_data テーブルが初期化され、以前に存在していた値がゼロになります。

esp_sqlUpdate からの戻り値は、次のフィールドを含む Lua オブジェクトです。

- code は、更新のステータスです。成功の場合は**真** 、失敗の場合は**偽**になります。
- message は、障害が発生した場合の障害に関する情報です。
- object は、失敗したレコードが発生したときのデータです。

データの削除

データベースからレコードを削除するには、esp_sqlDelete 関数を使用します。

```
local resp = esp_sqlDelete(options)
```

この関数の **options** は、削除プロパティを含む Lua テーブルです。

Lua コンポーネントを通じてデータベースからレコードを削除するには 2 つの方法があります。1 つ目の方法は、削除するレコードのキーフィールドデータを含む 1 つ以上の Lua オブジェクトを送信することです。このアプローチを使用すると、options パラメーターは次のフィールドを受け入れます:

- conn は、使用する SQL 接続の名前です。
- table は、レコードを削除するデータベーステーブルです。
- data は、削除するレコードのキーフィールドを含む 1 つ以上の Lua オブジェクトが含まれます。
- context は、接続が存在する Lua コンテキストを指定するオプションの値です。

次のコードは、更新セクションの trades_data テーブルからいくつかのレコードを削除します:

```
local   resp = esp_sqlDelete({conn="db",
                 table="trades_data",
                 data={{symbol="ABC"},{symbol="XYZ"}}})
```

レコードを削除する 2 番目の方法は、削除するデータベースレコードを指定する SQL WHERE 句を追加することです。次のコードは、特定の条件(max_price > 10 を満たすすべてのレコードを trades_data から削除します。

```
local resp = esp_sqlDelete({conn="db",table="trades_data",where="max_price > 10"})
```

esp_sqlDelete からの戻り値は、次のフィールドを含む Lua オブジェクトです。

- code は、削除のステータスです。成功の場合は**真** 、失敗の場合は**偽**になります。
- message は、障害が発生した場合の障害に関する情報です。
- object は、失敗したレコードが発生したときのデータです。

Lua スニペットの使用

Lua スニペットは、自己完結型の Lua コードのブロックです。Lua スニペットはプロジェクトレベルで作成します。

lua-snippet 要素を使用して、スニペットを定義します。詳細については、"lua-snippet" (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

code 要素を使用して、スニペットのコードを指定します。

コードはインラインで記述することも、外部ファイルに記述することもできます。外部ファイルにある場合は、ファイル属性を使用してそのファイルを参照できます。

```
<code file="mycode.lua" />
```

次のスニペットを考えてみましょう。初期化関数とクリーンアップ関数を実行した後、**token.txt** という名前のファイルを開いて、そこからトークンを読み取ります。次に、スニペットは ESP サーバーに token と呼ばれるプロパティを設定します。このプロパティには、他の Lua エンティティからアクセスできます。

```
<project...>
  <lua-snippets>
   <lua-snippet name="access_token" init="initfunc" destroy="cleanup">
      <code><![CDATA[
         function initfunc()
            print("initializing snippet")
         end

         function cleanup()
            print("cleaning up snippet")
         end

         function load()
            local   f = io.open("token.txt","r")
            local   token = f:read("*all")
            f:close()
            esp_setProperty({name="token",value=token})
         end
```

```
        load()

      ]]></code>
    </lua-snippet>
    ...
</project>
```

次の Lua コネクタを考えてみましょう。Lua スニペットによって設定されたトークンを使用して、REST 要求を作成します。

```
<connector class="lua" name="pub">
 <properties>
  <property name="type">pub</property>
  <property name="interval">5 seconds</property>
  <property name="code"><![CDATA[

    local   token = esp_getProperty({name="token"})

    function publish()
      -- use token to make REST request, parse response, and inject events
      return true
    end

  ]]></property>
 </properties>
</connector>
```

注: esp_getProperty 関数は、Lua プロパティの名前を取得します。

Lua スニペットを使用すると、指定した関数を指定した間隔で呼び出すスレッドを開始することもできます。次のコードでは、一定期間後にトークンの有効期限が切れると、スニペットが 5 分ごとに新しいトークンを取得してプロパティを設定するよう、前のスニペットを改訂しています。

```
<project ...>
 <lua-snippets>
  <lua-snippet name="largetrades">
    <lua-thread func="load" interval="5 minutes" />
    <code><![CDATA[

      function load()
        local   f = io.open("largetrade.txt","r")
        local   contents = f:read("*all")
        f:close()
        esp_setProperty({name="token",value=token})
        return true
      end

      load()

    ]]></code>
  </lua-snippet>
 </lua-snippets>
 ...
</project>
```

関数からの戻りコードは、スレッドを実行し続けるかどうかをサーバーに伝えるブール値です。**true** を返すと、スレッドは引き続き実行されます。**false** を返すと、スレッドは終了します。

改訂されたスニペットを参照すると、Lua コネクタは発行するたびにトークンを取得して、要求に対応する新しいトークンがあることを確認できます。

```
<connector class="lua" name="pub">
 <properties>
  <property name="type">pub</property>
  <property name="code"><![CDATA[

    function publish()
      local   token = esp_getProperty({name="token"})
      -- use token to make REST request, parse response, and inject events
      return true
    end

  ]]></property>
 </properties>
</connector>
```

Lua スニペット内のコードは、スニペットが XML で宣言されている順序で読み込まれます。別のスニペットが後続のスニペットで設定された変数を必要とする前にスニペットが宣言されている場合、それらの変数は使用できません。

次のコードを考えてみましょう。

```
<lua-snippets>
  <lua-snippet name="thisSnippet">
    <code><![CDATA[

      esp_setProperty({name="theproperty",value="hello, world"})

    ]]></code>
  </lua-snippet>
  <lua-snippet name="thatSnippet">
    <code><![CDATA[

      local   value = esp_getProperty({name="theproperty"})
      print("VALUE IS: "..tostring(value))

    ]]></code>
  </lua-snippet>
</lua-snippets>
```

これらのスニペットが指定どおりに含まれている場合、コードは ESP ログに次の出力を返します。

```
VALUE IS: hello, world
```

スニペットが転置されているとします。

```
<lua-snippets>
  <lua-snippet name="thatSnippet">
    <code><![CDATA[

      local   value = esp_getProperty({name="theproperty"})
```

```
        print("VALUE IS: "..tostring(value))

      ]]></code>
    </lua-snippet>
    <lua-snippet name="thisSnippet">
      <code><![CDATA[

        esp_setProperty({name="theproperty",value="hello, world"})

      ]]></code>
    </lua-snippet>
  </lua-snippets>
```

thatSnippet のコードが thisSnippet のコードより前に実行されるため、次の出力が ESP ログに返されます。

```
VALUE IS: nil
```

Lua エンティティの起動順序については、“Lua エンティティの 初期化”を参照してください。

Lua モジュールの使用

Lua モジュールは、プロジェクトレベルで存在する Lua コードの自己完結型ブロックです。モジュール内のコードは、他の Lua コンポーネントにコピーされるためだけに存在します。Lua モジュールを使用すると、コードを 1 回作成して、他の任意の数の Lua コンポーネントで再利用できます。

```
<project name="name"...>
  <lua-modules>
    <lua-module name="mod_1">
      <code><![CDATA[
       -- Lua code
      ]]></code>
    </lua-module>
    <lua-module name="mod_2">
      <code><![CDATA[
       -- Lua code
      ]]></code>
    </lua-module>
  </lua-modules>
  <contqueries>
    <contquery name="cq1">
    ...
    </contquery>
  </contqueries>
</project>
```

Lua モジュールを参照するには、他の Lua コンポーネントで require 属性を指定します。

```
<lua-module name="mod_1" require="mod_2"> <!-- 1 -->
<window-lua name="name" require="mod_1,mod_2">
<lua-snippet name="name"  require="mod_1,mod_2">
<window-filter name="name" require="mod_1,mod_2">
<window-pattern name="name">
   ...
   <code require="mod_1,mod_2"> <!-- 2 -->
```

1 Lua モジュールは別の Lua モジュールを呼び出すことができます。

2 Lua コードを使用するパターンウィンドウの code 要素で、require 属性を使用します。

Lua コネクタは、require プロパティを持つ Lua モジュールを取り込むことができます:

```
<connectors>
  <connector class="lua" name="pub">
   <properties>
    <property name="type">pub</property>
    <property name="interval">500 milliseconds</property>
    <property name="init">init</property>
    <property name="require">mod_1,mod_2</property>
   </properties>
  </connector>
 </connectors>
```

ESP サーバーのログをカプセル化して、任意の Lua エンティティで利用できるようにするクラスが必要だとします:

```
<lua-module name="logger">
   <code><![CDATA[
      Logger = {
         _context = nil
      }

      function Logger:new(context)
         o = {}
         setmetatable(o,self)
         self.__index = self
         self._context = context or "mylogger"
         return o
      end

      function Logger:error(msg,line)
         esp_logMessage(self._context,msg,"error",line)
      end

      function Logger:warn(msg,line)
         esp_logMessage(self._context,msg,"warn",line)
      end

      function Logger:fatal(msg,line)
         esp_logMessage(self._context,msg,"fatal",line)
      end

      function Logger:info(msg,line)
         esp_logMessage(self._context,msg,"info",line)
```

```
    end

    function Logger:debug(msg,line)
      esp_logMessage(self._context,msg,"debug",line)
    end

    function Logger:trace(msg,line)
      esp_logMessage(self._context,msg,"trace",line)
    end


  ]]></code>
</lua-module>
```

イベントに入ってくる数値を分割する Lua ウィンドウがあるとします。除数が 0 の場合、このモジュールの Logger クラスを使用してサーバーにエラーメッセージを記録する必要があります。次のコードでは、require 属性は、Logger クラスを含むモジュールを取り込みます。

```
<window-lua name="lua" events="create" require="logger">
 <schema>
  <fields>
   <field name="id" type="string" key="true"/>
   <field name="dividend" type="int32"/>
   <field name="divisor" type="int32"/>
   <field name="quotient" type="double"/>
  </fields>
 </schema>
 <copy>id,dividend,divisor</copy>
 <use>dividend,divisor</use>
 <code><![CDATA[
  function create(data)

    local   e = {}
    local   dividend = data["dividend"]
    local   divisor = data["divisor"]

    if (divisor == 0)
    then
      local   logger = Logger:new("modules.example")
      logger:error("division by 0, "..tostring(dividend).."/"..tostring(divisor))
      return nil
    end

    e["quotient"] = dividend / divisor

    return(e)
  end

 ]]></code>
</window-lua>
```

Lua コードはクラスのインスタンスを作成し、それを使用してメッセージを ESP サーバーログに記録します。0 による除算が発生すると、サーバーログにエラーが表示されます。

```
{
    "level":"ERROR",
    "message":"{6c00732f-5d9d-4203-9c42-731d4e442b30}[XMLServer0001] division by
0, 31\/0",
    "properties":
    {
        "caller":"Lua.cpp:1910",
        "logger":"modules.example",
        "thread":"00000033"
    },
    "timeStamp":"2023-09-13T16:19:04.891000-04:00"
}
```


Lua コードの例外の処理

Lua コードは例外が発生する可能性があります。init 関数からの例外により、プロジェクトの読み込みに失敗します。create 関数、heartbeat 関数、およびスニペット(スレッド化された場合)からの例外は、project 要素の on-script-exception 属性に基づいて処理されます。詳細については、*"project" (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*を参照してください。

9

一般的な Python 機能

ESP プロジェクトのユーザー提供のコード

SAS Event Stream Processing は、独自のコード(Python、Lua、またはその他のカスタムコンポーネントなど)でプロジェクトを拡張できます。使用することを決定したコードのビヘイビアーとセキュリティについては、ユーザーとその組織のみが責

任を負っています。SAS Event Stream Processing は、セキュリティ目的でユーザーが提供したコードをレビューまたはテストしません。ESP プロジェクトに追加するコードやその他のカスタマー提供の資料は、ユーザーの責任のままです。

注意
安全でないコードを実行しないでください。 安全でないコードを実行すると、環境がデータの損失、データの破損、セキュリティ違反、またはシステムの不安定性につながる可能性があります。組み込むすべてのコードが安全であり、テストされ、組織のポリシーと適用される法律に準拠していることを確認するのはあなたの責任です。

Python エンティティの初期化

ESP サーバーは、いくつかの Python エンティティとそれらの間の通信を提供します。ディスクに保存され、サーバーの起動時に再度読み込まれる永続プロパティもあります。サーバーの起動時にこれらすべてがどのように初期化されるかを理解しておくことが重要です。

すべての Python エンティティは初期化メカニズムを提供します。Python ウィンドウは、モデル XML の init 属性を通じてこのメカニズムを介して実装します。

Python コネクタは、Python コードの初期化関数を指すオプションの init パラメーターを提供します。

他の Python エンティティからプロパティ値を取得する Python エンティティがある場合は、初期化関数を使用することが重要です。たとえば、次のコードは、コードの呼び出し時に他の Python エンティティが value プロパティを設定していない可能性があるため、None 値を生成する可能性があります。

```
myvalue = esp.getProperty(name="value")
```

ただし、プロパティ値を取得するアクションを初期化関数内に配置すると、他のエンティティが初期化されている場合にプロパティが設定されていることを確認できます。

```
myvalue = None
def init():
   myvalue = esp.getProperty(name="value")
```

ESP サーバーが起動すると、Python 関連のアクションが次の順序で実行されます:

1. Python スニペットの永続プロパティを読み込みます。
2. Python ウィンドウの永続プロパティを読み込みます。
3. init 関数を持つ Python スニペットに対してそれを呼び出します。
4. スレッド化された Python スニペットを開始します。
5. 任意の Python ウィンドウでも、init 関数がある場合はそれを呼び出します。また、ウィンドウに Python スプリッターがある場合、そのウィンドウで init 関数が呼び出されます。

上記の操作はすべて、XML モデルでエンティティが定義されている順序で実行されます。

Python での SAS Event Stream Processing 関数の使用

概要

ESP サーバーは、Python コード内で呼び出すことができる関数を提供し、次のアクティビティを可能にします。esp と esp_utils というパッケージが 2 つあります。ESP パッケージには、ESP サーバーと通信する関数が含まれています。esp_utils パッケージは、ONNX モデルおよびイメージの操作に使用されます。次の表は、パッケージとアクティビティ 別に関数をまとめたものです。

ESP 関数を使用するには、Python コードに次の行を含めます。

```
import esp
```

表 9.1　アクティビティによって 編成された ESP 関数

アクティビティ	関数
“イベントの操作”	esp.publish esp.query
HTTP リクエストの操作	esp.sendHttp
“ESP サーバーログのデバッグ”	esp.logMessage
“Python プロパティの操作”	esp.setProperty esp.getProperty esp.clearProperty
“Python スニペットからの関数の呼び出し”	esp.call
“その他の関数”	esp.getServerProperty esp.getProjectProperty
リレーショナルデータベースへのアクセス	esp.sqlConnect esp.sqlSelect esp.sqlOpen

アクティビティ	関数
	esp.sqlNext
	esp.sqlInsert
	esp.sqlUpdate
	esp.sqlDelete

esp_utils 関数を使用するには、Python コードに次の行を含めます。

```
import esp_utils
```

表 9.2　アクティビティによって 編成された esp_utils 関数

アクティビティ	関数
“画像の操作”	esp_utils.image_conversion.sas_wide_image_to_opencv_image esp_utils.image_conversion.opencv_image_to_sas_wide_image esp_utils.image_conversion.blob_image_to_opencv_image esp_utils.image_conversion.opencv_image_to_blob_image
“テンソルの操作”	np_array_to_tensor tensor_to_np_array
“後処理”	bbox_letterbox_to_original xy_letterbox_to_original

イベントの操作

イベントのパブリッシュ

このトピックは esp パッケージに関連しています。esp.publish 関数を使用して、イベントをソースウィンドウにパブリッシュします。

```
local response = esp.publish(window="cq2/storeAlerts",events=events)
```

表 9.3　esp.publish 関数のパラメーター

パラメーター	説明
window	ソースウィンドウの名前を指定します。

パラメーター	説明
events	パブリッシュするイベントデータ(配列または単一オブジェクト)を指定します。
opts	パブリッシュオプションを指定します。 ■ blocksize - パブリッシュするイベントブロックのサイズを指定します。デフォルト値は **1** です。

esp.publish 関数は、イベントをキューに入れて非同期に処理します。戻り値には、コードフィールドとメッセージフィールドが含まれます。

イベントのクエリ

このトピックは esp パッケージに関連しています。イベントをクエリするには、esp.query 関数を使用します:

```
patients = esp.query(window="patients")
vitals = esp.query(window="vitals")
```

表 9.4 esp.query 関数のパラメーター

パラメーター	説明
window	イベントを受信するウィンドウを指定します。
output	次のエントリを使用して Python オブジェクトを指定します: ■ fields - イベント配信に含めるフィールドのカンマ区切りリストを指定します。 ■ exclude - で指定された項目が有効かどうかを示すブール値。fields 含めるか除外する必要があります。デフォルト値は **false** です。 ■ encode_binary - イベント配信用にバイナリデータを Base64 エンコードするかどうかを示すブール値です。デフォルト値は **false** です。
matches	照合するイベントフィールドを含む Python オブジェクトを指定します。次に例を示します: data = esp.query(window="measurement",matches={"patient":"Patient 1"})
sort	イベントの並べ替えに使用される Python オブジェクトを指定します。 ■ fields - 並べ替えるフィールドのカンマ区切りリストを指定します。 ■ dir - ソート方向を**昇順**または**降順**で指定します。デフォルトは **descending** です。

パラメーター	説明
	■ maxevents - maxevents 返されるイベントの最大数を指定します。デフォルトでは、すべてのイベントが返されます。

株式取引を処理するプロジェクトありによる上位 10 件の取引を数量順に取得したいとします。次のコードは、esp.query を使用して trades という名前のウィンドウからイベントを取得し、そしてそれに応じて出力を並べ替えます。

```
events = esp.query(window="p/cq/trades",

output={"fields":"symbol,quantity,price"},

sort={"fields":"quantity","maxevents":10})
print("events\n" + str(events))
```

HTTP リクエストの送信と応答の受信

このトピックは esp パッケージに関連しています。HTTP リクエストを送信してから、esp.sendHttp(*fields*)関数を使用して、応答を Python ベースウィンドウの Python コードに戻します。リクエストオブジェクトをフォーマットし、次の *fields* を使用して関数に渡します。

表 9.5　リクエストオブジェクトフィールド

フィールド	説明
url	リクエストの URL を指定します。
method	使用する HTTP メソッドを指定します(**GET**、**PUT**、**POST**、**DELETE**)。デフォルト値は **GET** です。
headers	サーバーに送信する HTTP リクエストヘッダーを指定します。
body	**PUT** および **POST** の HTTP リクエストの本文を指定します。
topython	HTTP 応答が JSON または XML の場合、ESP サーバーが応答を返す前に応答を Python オブジェクトに変換するために、このフィールドを **true** に設定できます。
connect-timeout	サーバーとの接続が正常に確立されるまでのタイムアウトを秒数で指定します。デフォルト値は **0** です(タイムアウトなし)。
data-timeout	サーバーからデータを正常に受信するまでのタイムアウトを秒数で指定します。デフォルト値は **0** です(タイムアウトなし)。

次のコードを考えてみましょう。

```
import esp

def create(data,context):
   url = "http://api.openweathermap.org/data/2.5/group?
id=4487042&APPID=d0b38229a8f66dac4fbce8bf76be97b4&units=imperial"
   response = esp.sendHttp(url=url,method="GET",headers={"accept":"application/
json"},topython=True)

   print(str(response))
```

実行すると Python ベースのウィンドウ内で上記のコードを実行すると、天気 API から次の応答が返されるはずです。

```
status=200 response={"cnt":1,"list":[{"coord":
{"lon":-78.6386,"lat":35.7721},"sys":
{"country":"US","timezone":-14400,"sunrise":1650450904,"sunset":1650498685},"weath
er":[{"id":801,"main":"Clouds","description":"few clouds","icon":"02d"}],"main":
{"temp":40.91,"feels_like":40.91,"temp_min":35.08,"temp_max":46.42,"pressure":1031
,"humidity":89},"visibility":10000,"wind":{"speed":0,"deg":0},"clouds":
{"all":20},"dt":1650456592,"id":4487042,"name":"Raleigh"}]}
```

デバッグ

概要

診断出力を Python 関数に直接追加できます。この出力を表示するには 2 つの方法があります。

- コンソールを介して
- ESP サーバーログを介して

コンソールでのデバッグ

Python の print()関数を使用して、コンソールに情報を表示します。

たとえば、パターンウィンドウ内で株式取引情報のシーケンスを表示しているとします。次のコードは、関数の実行時に現在のイベントとコンテキストをダンプします。関数の最後に、戻り値がダンプされます。

```
def f3(event,context):
   print("in function e3")
   print(str(event))
   print(str(context))

   code = event["broker"] == "George"
         and event["price"] < context["events"]["john"]["price"]
         and event["price"] < context["events"]["paul"]["price"]
```

```
    print("returning: " + str(code) + " from f3")

    return code
```

コードを実行すると、次の例のような出力が生成されます:

```
in function e3

           {'price':50.0, 'quantity':50, 'broker':'George', 'id':6,
'symbol':'IBM'}

           name':'george'
           window':1,
           events':{'john': { 'price':97.34 'quantity':500 'broker':'John'
'id':0 'symbol':'IBM' }},
                    {'paul': { 'price':57.34 'quantity':500 'broker':'Paul'
'id':3 'symbol':'IBM' }}
           }

       returning: true from f3
```

イベントとコンテキストをダンプすると、パターンに従って、特定のイベントが特定のパターンをトリガーする理由を判断できます。

ESP サーバーログのデバッグ

このトピックは esp パッケージに関連しています。esp.logMessage 関数を使用して、Python から ESP サーバーログにメッセージを記録できます。esp.logMessage 関数は、次のパラメーターを取ります。

表 9.6　esp_logMessage 関数のパラメーター

パラメーター	説明
logcontext	ESP サーバーでコンテキストオブジェクトを指定します(**DF.ESP** など)。このパラメーターは必須です。コンテキストが存在しない場合、ESP サーバーはコンテキストを作成します。
message	ログに記録するメッセージを指定します。このパラメーターは必須です。
level	使用するログレベルを指定します。指定されたログコンテキストがそのログレベル以上に設定されている場合、メッセージはログに記録されます。それ以外の場合、メッセージは破棄されます。 level の可能な値は次のとおりです。 ■ **trace** ■ **error**

パラメーター	説明
	■ **warn** ■ **fatal** ■ **info** ■ **debug** このパラメーターはオプションで、デフォルトは **info** です。

line	メッセージが記録される Lua コードの行番号を指定します。この値は次の Python 呼び出しで生成されます。
	`import inspect`
	`inspect.getframeinfo(inspect.currentframe()).lineno`
	line の値を指定しない場合、Python の行番号はログに記録されません。

イベントを受け取り、データを少し変更してイベントを生成する次の関数を考えてみましょう。mylogger ログコンテキスが **debug** 以上に設定されている場合に、コードは受信イベントと生成されたイベントをログに記録します。

```
import inspect
import esp

   eventId = 1

function create(data,context)
   global eventId

   esp.logMessage(logcontext="mylogger",message="creating event from" +
str(data),level="debug",line=inspect.getframeinfo(inspect.currentframe()).lineno)

   e = {}

   e["id"] = str(eventId)
   e["new_string"] = "\"" + data["string"] + "\""
   e["new_number"] = data["number"] * 3
   e["new_array"] = []
   for value in data.array:
      e["new_array"].append(value * 2)

   eventId += 1

   esp.logMessage(logcontext="mylogger",message="created event " +
str(e),level="debug",line=inspect.getframeinfo(inspect.currentframe()).lineno)

   return(e)
```

ログ出力は、次の例のようになります。

```
_timestamp=2021-08-23 19:19:48
logName=mylogger
logLevel=Debug
messageContent={81dec492-c3f2-42ba-9cb9-1793ed7fedf7}[XMLServer0001] creating
event from
    id=1
    string=string data
    array=
        1=11
        2=24
        3=55
    number=33
 (Python line 7)
messageFile=./src/dfESPwindow_python.cpp
messageLine=774

_timestamp=2021-08-23 19:19:48
logName=mylogger
logLevel=Debug
messageContent={81dec492-c3f2-42ba-9cb9-1793ed7fedf7}[XMLServer0001] created event
    id=1
    new_string="string data"
    new_array=
        1=22
        2=48
        3=110
    new_number=99
 (Python line 21)
messageFile=./src/dfESPwindow_python.cpp
messageLine=774
```

次のコードは、パターンマッチング関数のサーバーロギングを示しています:

```
def f3(event,context):

    message = "in function e3"
    message = message + "\n" + str(event)
    message = message + "\n" + str(context)

    code = event["broker"] == "George"
          and event["price"] < context["events"]["john"]["price"]
          and event["price"] < context["events"]["paul"]["price"]

    message = message + "\n" + "returning: " + str(code) + " from f3"

    esp.logMessage(logcontext="mypatternlogger",
                message=message,
                level="debug",
                line=inspect.getframeinfo(inspect.currentframe()).lineno)

    return code
```

もし mypatternlogger ロギングコンテキストが適切に設定されている場合、ESP サーバーログに次の例のような内容が表示されるはずです。

```
{
      "_timestamp": "2021-12-16T12:36:43.312000-05:00",
      "logger": "mypatternlogger",
      "loggerLevel": "DEBUG",
      "messageContent": "{f3b6119e-e65c-4ad9-a09b-36075fc43339}[XMLServer0001] in
function e3\n\n    symbol=IBM\n    quantity=50\n    id=6\n    broker=George\n
price=50.0\n\n\n    window=1\n    name=george\n    events=\n        paul=
\n            symbol=IBM\n            quantity=500\n            id=3\n
broker=Paul\n            price=57.34\n        john=\n            symbol=IBM
\n            quantity=500\n            id=0\n            broker=John
\n            price=97.34\n\nreturning: true from f3 (Python line 23)",
      "messageFile": "dfESPwindow_pythonbase.cpp",
      "messageLine": "130",
      "thread": "00000034"
}
```


esp.getLogLevel()関数を使用して、ESP サーバー内のロガーの現在のログレベルを取得します。この関数は、名前が指定されていない場合、個々のログコンテキストまたはすべてのコンテキストに関する情報を返します。

数値レベルとレベル名の両方がデータに返されます。

表 9.7　数値レベルとログレベル名

数値	ログレベル
2	Trace
3	Debug
4	Info
5	Warn
6	Error
7	Fatal
8	Off

個々のロガーを取得するには、次のコマンドを入力します。

```
logger = esp.getLogLevel("DF.ESP")
```

データは次のようになります。


```
{
    "level": 6,
    "levelname": "error",
    "name": "DF.ESP"
}
```


すべてのロガーを取得するには、次のコマンドを入力します。

```
logger = esp.getLogLevel()
```

データは次のようになります。

```
[
  {
    "level": 6,
    "levelname": "error",
    "name": "DF.ESP"
  },
  {
    "level": 4,
    "levelname": "info",
    "name": "DF.ESP.AUTH"
  },
  {
    "level": 4,
    "levelname": "info",
    "name": "common.http"
  },
  ...

]
```

Python プロパティの操作

概要

次のトピックは、esp パッケージに関連しています。Python プロパティを使用して、SAS Event Stream Processing プロジェクト内の Python エンティティ 間でコミュニケーションできます。Python プロパティは、ESP サーバーに格納される名前と値のペアです。これらのペアは Python コンテキストに保存されます。通常、デフォルトのコンテキストは ESP サーバーによって自動的に作成されるため、プロパティにアクセスするときにコンテキストを指定する必要はありません。ただし、Python プロパティへのアクセスをより 効率的にできると思われる場合はいつでも、複数のコンテキストを作成できます。

次のコードは、2 つの異なるコンテキストで myproperty の値を設定します。

```
esp.setProperty(name="myproperty",value="foo")
esp.setProperty(name="myproperty",value="foo",context="newcontext")
```

最初の呼び出しでは、デフォルトの Python コンテキストに myproperty が設定されます。2 番目の呼び出しでは、newcontext という名前のコンテキストにプロパティを設定します。どちらの場合も、呼び出しによりプロパティの値がハードコードされた文字列 **one** に設定されます。変数の値を参照してプロパティの値を設定するには、二重引用符を省略します。

すべての Python エンティティは、プロパティ値を読み取ることができます。

ESP サーバーは、Python オブジェクトに関連付けられたプロパティをロックする必要がありますが、プロパティを取得するには READ ロックのみが必要です。プロパティを所有する Python オブジェクトがプロパティを変更するたびに、プロパティ

は WRITE ロックでロックされます。ロックにより、プロパティの操作効率が向上します。

Python プロパティの値は、ESP サーバーに保存される Python オブジェクト(文字列、数値、テーブルなど)です。

プロパティの設定

プロジェクト内の任意の Python コードから Python プロパティを設定できます。esp.setProperty 関数を使用して、Python プロパティを設定します。

表 9.8 esp_setProperty 関数

構文	説明
esp.setProperty(name=*name*,value=*value*[,field=*field*][,context=*context*][,persist=true | false])	name には、Python プロパティの名前を指定します。 value には、Python プロパティ(任意の Python オブジェクト)の値を指定します。 field パラメーターを使用して、Python プロパティのフィールドを設定します。 context パラメーターを使用して、プロパティを設定する異なる Python コンテキストを指定します。 プロパティを永続的にするには、persist=true のように設定します。永続的なプロパティは、設定時にファイルシステムに保存されます。ESP サーバーは、これらの保存された値を使用して、起動時にプロパティ値を初期化します。

プロパティ内のフィールドを指定するには、ドット表記を使用します。たとえば、次の従業員情報がプロパティとして格納されているとします:

```
employees = {}

employees["Joe"] = {"department":"Marketing","position":"Marketer","info":
{"vacation_days":30,"years_of_service":8}}
employees["Sally"] = {"department":"Research and
Development","position":"Developer","info":{"vacation_days":30,"years_of_service":10}}
employees["Mark"]={"department":"Research and Development","position":"Tester","info":
{"vacation_days":10,"years_of_service":3}}
employees["Helen"]={"department":"Support","position":"Customer Support
Engineer","info":{"vacation_days":24,"years_of_service":30}}

esp.setProperty(name="employee_data",value=employees)
```

次のコードは、Joe の休暇日を設定します:

```
esp.setProperty(name="employee_data",value=25,field="Joe.info.vacation_days")
```

次の Python ウィンドウは、ハートビートを使用してプロパティを設定します:

```
<window-python name="python" events="create" heartbeat="hb">
 <schema>
  ...
 </schema>
  <code><![CDATA[

     def create(data,context):
        ...

     def hb():
        info = {}
        info["foo"] = "bar"
        info["num"] = 100
        esp.setProperty(name="info",value=info)

  ]]></code>
</window-python>
```

esp.setProperty 関数は、プロパティが正常に設定された場合は **true** を返し、プロパティの設定中にサーバーでエラーが発生した場合は **false,err** を返します。通常、エラーをキャッチするための呼び出しは次の例のようになります。

```
code,err = esp.setProperty(name="product",value="Esp")

if (code == False):
   print("cannot set property: "..err)
```

プロパティの取得

すべての Python エンティティは、esp.getProperty 関数を使用してサーバーから Python プロパティを取得できます。

表 9.9　esp_setProperty 関数

構文	説明
esp.getProperty(name=*name*, [field=*field*][,context=*context*])	name には、取得するプロパティの名前を指定します。field パラメーターを使用して、取得するフィールドを指定します。context パラメーターを使用して、プロパティを取得する異なる Python コンテキストを指定します。

この関数はプロパティの値を返すか、プロパティが見つからない場合は **nil** を返します。エラーが発生した場合(たとえば、ESP サーバーがエンティティを 見つけられない場合)、2 番目のパラメーターとして **nil** が返されます。

```
data = esp.getProperty(name="employee_data")
print(str(data))
```

前のトピックで提供された従業員情報の Python プロパティ全体を取得したいとします。また、Python エンティティの名前は test だとします。

次の出力が生成されます。

```
{'Helen': {'department': 'Support', 'info': {'vacation_days': 24.0,
'years_of_service': 30.0}, 'position': 'Customer Support Engineer'},
    'Joe': {'department': 'Marketing', 'info': {'vacation_days': 30.0,
'years_of_service': 8.0}, 'position': 'Marketer'},
    'Mark': {'department': 'Research and Development', 'info': {'vacation_days':
10.0, 'years_of_service': 3.0}, 'position': 'Tester'},
    'Sally': {'department': 'Research and Development', 'info': {'vacation_days':
30.0, 'years_of_service': 10.0}, 'position': 'Developer'}}
```

Sally 専用に info フィールドが必要だとします。次のコードを考えてみましょう。

```
data = {"name": "employee_data", "field": "Sally.in"}
err = esp.getProperty(data)
```

次の出力が生成されます。

```
{'vacation_days': 30.0, 'years_of_service': 10.0}
```

注: マルチバイト文字を含む文字列はバイナリデータとして保存されます。プロパティの値がバイナリデータ(Python バイト配列)であり、それを文字列として取得したい場合は、decode=True を設定する必要があります。

```
data = esp.getProperty(name="中 employee_data 文",decode=True)
```

プロパティのクリア

esp.clearProperty は、Python コンテキストからプロパティを削除します。

表 9.10 esp.clearProperty 関数

構文	説明
esp.clearProperty(name=*name*[,field=*field*][,context=*context*][,persist=true | false])	name を使用して、クリアするプロパティの名前を指定します。field パラメーターを使用して、クリアするプロパティのフィールドを指定します。context を使用して、プロパティをクリアにする異なる Python コンテキストを指定します。persist を使用して、ディスクからプロパティ値を削除するかどうかを指定します。

esp.clearProperty 関数は、プロパティが正常にクリアされた場合は **true** を返し、プロパティをクリアしようとしたときに ESP サーバーでエラーが発生した場合は **false** を返します。

永続的なプロパティ

永続的なプロパティは、設定時にファイルシステムに保存されます。ESP サーバーは、これらの保存された値を使用して、起動時にプロパティ値を初期化します。

永続プロパティを有効にするには、 esp.setProperty または esp.clearProperty 関数の persist プロパティを設定します。

```
esp.setProperty(name="myproperty",value="foo",persist=True)

esp.clearProperty(name="myproperty",persist=True)
```

プロパティは、プロジェクトの Lua のルートディレクトリから派生したディレクトリに保存されます。

画像の操作

SAS Event Stream Processing は、Python コード内で画像データを操作する関数を提供します。

これらの関数を使用するには、esp_utils パッケージをインポートする必要があります。

```
import esp_utils
```

関数	説明
sas_wide_image_to_opencv_image	画像を SAS ワイド形式から OpenCV で使用できる形式に変換します。 デフォルトでは、copy パラメーターが **True** に設定されているため、画像の処理が遅くなり、画像の編集や注釈が可能になります。画像をより迅速に処理するために、copy=False を設定します。
my_opencv_image = esp_utils.image_conversion.sas_wide_image_to_opencv_image(data['my_sas_wide_image'], copy= True | False)	
opencv_image_to_sas_wide_image	OpenCV で作成された画像を SAS ワイド形式に変換します。 **注:** 画像圧縮のオーバーヘッドを回避するため、特定のシナリオでパフォーマンスを向上させるために SAS ワイド形式を使用することをお勧めします。

関数	説明
my_sas_wide_image = esp_utils.image_conversion.opencv_image_to_sas_wide_image(data['my_opencv_image'])	
blob_image_to_opencv_image	JPEG または PNG 画像を OpenCV で使用できる形式に変換します。
my_opencv_image = esp_utils.image_conversion.blob_image_to_opencv_image(data['my_blob_image'])	
opencv_image_to_blob_image	OpenCV で作成された画像を、JPEG(デフォルト)または PNG 画像に変換します。 出力する type に、**.jpeg**(デフォルト)または**.png** を指定します。1 秒間に送信するフレーム数に対応する整数 quality 値に、**0** から **100** を指定します。これにより、ファイルサイズと品質を制御できます。デフォルト値は **85** です。
my_blob_image = esp_utils.image_conversion.opencv_image_to_blob_image(data['my_opencv_image'],type=".jpeg | .png", quality=u	

注: Grafana には JPEG および PNG 画像が必要です。

テンソルの操作

SAS Event Stream Processing は、Python コード内でテンソルを操作する関数を提供します。このトピックは esp_utils パッケージに関連しています。テンソルの詳細については、*"スコアウィンドウでのテンソルの表現" (SAS Event Stream Processing: ストリーミング分析の適用)*を参照してください

np_array_to_tensor 関数を使用して、NumPy 配列を ESP テンソルに変換します。ONNX モデルスコアリング用にデータを前処理する予定がない場合は、代わりにこの関数を使用できます。NumPy を使用してデータを準備し、この関数を使用してデータをテンソルに変換できます。

```
my_tensor = np_array_to_tensor(data['my_numpy_array'])
```

tensor_to_np_array 関数を使用して、テンソル出力を NumPy array 配列に変換します。結果として得られる NumPy 配列は、ONNX モデルの出力をデコード(後処理)するのに役立ちます。

```
my_numpy_array = tensor_to_np_array(data['my_tensor'])
```

後処理

このトピックは esp_utils パッケージに関連しています。

bbox_letterbox_to_original 関数を使用して、レターボックスイメージの x,y,w,h 境界ボックスを元のイメージの座標に変換します。この関数は、レターボックスのサイズ変更されたイメージで Computer Vision オブジェクト検出モデルを使用し、検出されたオブジェクトを元の(サイズ変更されていない)フレームで視覚化する場合に役立ちます。

xy_letterbox_to_original を使用して、レターボックス画像内の(x,y)位置を元の画像内の座標に変換します。この関数は、Computer Vision のキーポイント検出モデルを使用する場合に便利で、元のフレーム内のキーポイントを視覚化できます。

Python スニペットからの関数の呼び出し

このトピックは esp パッケージに関連しています。esp.call 関数を使用して、Python スニペットで定義された関数を呼び出すことができます。まず、呼び出された関数内でデータとして使用するパラメータを指定します。次に、_pyname_パラメータは参照するスニペットを識別します。二番目の _pyfunc_パラメータは、そのスニペット内で実行する関数を識別します。サーバーはこれらのパラメーターを削除し、他のすべてのパラメーターを 指定された関数に渡します。

```
response = esp.call(parameter,parameter,_pyname_="snippet_name",_pyfunc_="func_name")
```

戻り値は、次のフィールドを含む Python オブジェクトです。

- code - 関数が正常に実行されたかどうかを示すブール値。
- results - 呼び出しが正常に実行された場合の関数からの戻り値の配列。
- error - 通話が失敗した場合のエラーの説明。

次のスニペットを考えてみましょう:

```
<python-snippet name="math">
 <code><![CDATA[

    import math

    def divide(dividend,divisor):
      value = dividend / divisor
      return value

    ]]></code>
</python-snippet>
```

divide 関数を使用して、次のように呼び出すことができます:

```
import esp

def create(data,context):
```

```
e = []
dividend = data["dividend"]
divisor = data["divisor"]

response = esp.call(dividend,divisor,_pyname_="snippet",_pyfunc_="divide")

if (response["code"] == True):
   e["quotient"] = response["results"][0]
else:
   e["error"] = response["error"]

return e
```

0 による除算がある場合、応答の code フィールドは **false** となり、応答の error フィールドはエラーを示します。それ以外の場合、コードフィールドは **true** となり、結果フィールドには商を提供する単一のエントリが含まれます。

詳細については、“Python スニペットの使用”を参照してください。

その他の関数

これらの関数を使用するには、esp パッケージをインポートする必要があります。

```
import esp
```

表 9.11 その他の関数

関数	説明

esp.getServerProperty(name)	ESP サーバーからサーバーレベルプロパティの値を取得します。これらは、サーバーコマンドライン、esp-properties.yml、または環境で指定された値です。たとえば、REST 要求を送信して、実行しているサーバーから ESP プロジェクトとスキーマを取得する場合:
	`host = esp.getServerProperty("HOST")`
	`port = esp.getServerProperty("http")`
	`url = "http://" + host + ":" + port + "/eventStreamProcessing/v2/projects?schema=true"`
	
	`request = {}`
	`request["url"] = url`
	`request["method"] = "GET"`
	`request["headers"] = {"accept":"application/json"}`
	`request["topython"] = True`
	`response = esp.sendHttp(request)`
	`print(str(response))`

esp.getProjectProperty(name, [project])	プロジェクトレベルプロパティの値を取得します。プロジェクトパラメーターはオプションです。プロジェクト名を指定せず、サーバー内にプロジェクトが 1 つしかない場合は、そのプロジェクトが使用されます。これらの値は、プロジェクト XML で定義されています。
	`<project>`
	`  <properties>`

関数	説明
	<property name="property_1">value 1</property> <property name="property_2">value 2</property> </properties> ... </project>

Python コンポーネントからのリレーショナルデータベースへのアクセス

概要

ODBC ドライバーを介して、Python コンポーネントからリレーショナルデータベースにアクセスできます。

ODBC ドライバーを使用する場合は、システムデータソース名(DSN)を介してデータベースへの接続を確立します。この DSN および関連するデータベースユーザー資格情報は、必須の構成パラメーターです。DSN の一般的な形式は次のとおりです。

DSN=*data_source*[;*attribute*=*value*[;*attribute*=*value*]...]

データベース接続に使用する目的のワイヤプロトコルを指定する *data_source* を指定する必要があります。指定したデータソースに対して、1 つ以上の関連属性を列挙します。ほとんどのワイヤプロトコルでは、データベースにアクセスするために *ユーザー ID* および *パスワード* を指定する必要があります。多くのワイヤプロトコルでは、データベースの場所を識別する *ホスト名* を指定することも必要です。

data_source に指定した各属性は、SAS Event Stream Processing で提供される odbc.ini ファイルに設定された値をオーバーライドします。

詳細については、“Using DataDirect ODBC Drivers” (*SAS Event Stream Processing: Connectors and Adapters*)を参照してください。

データベース接続は Python コンテキストに保存されます。ほとんどの場合、デフォルトのコンテキストは ESP サーバーによって作成されるため、接続を作成するときにコンテキストを指定する必要はありません。

重要　Python コンポーネントによる次のリレーショナルデータベースへのアクセスがサポートされています。

- Db2
- Gleenplum
- MySQL

- Oracle
- PostgreSQL

サポートされていないデータベースはご自身の判断で使用してください。

データベースへの接続

データベースに接続するには、esp.sqlConnect 関数を使用します。次に例を示します:

```
code,err = esp.sqlConnect(options)
```

この関数の **options** は、次のフィールドを含む Python オブジェクトです。

- name は、SQL 接続の名前です。
- connstr は、ここ説明されている DSN です。
- context は、接続を作成する Python コンテキストを指定するオプションの値です。

接続の名前は、後続のデータベース呼び出しで使用されます。

esp_sqlConnect 関数は 2 つの値を返します。最初に返される値は、接続が成功したかどうかを示すブール値です。接続が失敗した場合は、2 番目の値を通じて失敗の理由が返されます。

たとえば、次のコードは、PostgreSQL データベースに接続する方法を示しています。

```
connstr = "DSN=PostgreSQL Wire
Protocol;UID=postgres;PWD=mypass123;servername=myserver;port=4444;database=espd
emo"
code,err = esp.sqlConnect(name="db",connstr=connstr)
```

接続が完了したら、esp.sqlDisconnect 関数を使用して接続を削除できます。

```
code,err = esp.sqlDisconnect(options)
```

この関数の **options** は、次のフィールドを含む Python オブジェクトです。

- name は、切断する SQL 接続の名前です。
- context は、接続が存在する Python コンテキストを指定するオプションの値です。

esp.sqlDisconnect 関数は 2 つの値を返します。返される最初の値は、切断が成功したかどうかを示すブール値です。切断が失敗した場合は、2 番目の値を通じて失敗の理由が返されます。

次のコードは、Postgres 接続を切断する方法を示しています。

```
code,err = esp.sqlDisconnect(name="db")
```

データの取得

データベースからデータを取得するには 2 つの方法があります。1 つ目の方法は、SQL select ステートメントを ESP サーバーに送信することによって、SQL クエリからすべての 結果を取得します。

```
resp = esp.sqlSelect(options)
```

このステートメントの **options** は、次のフィールドを含む Python オブジェクトです。

- conn は、使用する SQL 接続の名前です。
- query は、SQL の選択ステートメントです。
- context は、接続が存在する Python コンテキストを指定するオプションの値です。

戻り値は、次のフィールドを含む Python オブジェクトです。

- code は、クエリが成功したかどうかを示すブール値です。
- data は、データベースのデータです。
- message は、クエリが失敗したときに送信される情報メッセージです。

esp.sqlSelect は、中程度のサイズのクエリで使用できます。たとえば、何百万もの株式取引が取引テーブルに保存されているとします。次のコードは、非常に大規模な取引(100 万株以上)を取得する方法を示しています。

```
resp = esp.sqlSelect(conn="db",query="select * from trades where quant > 1000000")

print(str(resp))
```

```
{'code': True,
   'data': (
      {'broker': 101667.0, 'buyer': 1012223.0, 'buysellflg': 0.0, 'currency': 87236.0, 'id':
2669875.0, 'msecs': 418.0, 'price': 4.15, 'quant': 2063400.0, 'seller': 8382739.0, 'symbol': 'C',
'udate': 1280928609000000.0, 'venue': 55555.0},
      {'broker': 101667.0, 'buyer': 0.0, 'buysellflg': 1.0, 'currency': 87236.0, 'id': 18564845.0,
'msecs': 724.0, 'price': 21.7, 'quant': 1286200.0, 'seller': 8123541.0, 'symbol': 'XEL', 'udate':
1280928728000000.0, 'venue': 55555.0}
)}
```

ただし、1 回のパスですべての取引を選択すると、メモリを大量に消費します。データベースから大量のデータを取得する必要がある場合は、データを段階的に取得することをお勧めします。次の関数を使用してクエリを開き、データを段階的に取得します。

- esp_sqlOpen - クエリを開きます。
- esp_sqlNext - 結果から指定された数の行を取得します。
- esp_sqlClose - クエリを閉じます。

esp.sqlOpen 関数を次のように使用します。

```
handle = esp.sqlOpen(options)
```

関数において、**options** には次のフィールドが含まれます。

- conn は、ESP サーバーへの SQL 接続の名前です。
- query は、SQL の選択ステートメントです。
- context は、接続が存在する Python コンテキストを指定するオプションの値です。

戻り値は、次のフィールドを含む Python オブジェクトです。

- code は、クエリが成功したかどうかを示すブール値です。
- message は、クエリが失敗した場合に送信される情報メッセージです。

クエリを正常に開くことができたら、esp.sqlNext 関数を使用してデータを取得できます。

```
data,count = esp.sqlNext(handle,numrows)
```

- handle は、esp.sqlOpen から返される値です。
- numrows は、返される行数です。

numrows パラメータを指定しない場合は、結果の残りの行が返されます。

この関数は 2 つの値を返します。最初の値はクエリ結果のデータ、またはエラーが発生した場合は **nil** です。2 番目の値は返されるレコードの数で、エラーが発生した場合、またはデータが使い果たされた場合は **nil** になります。関数がカウント 0 を返すたびに、データは使い果たされます。その場合、サーバーはクエリを閉じるため esp.sqlClose を呼び出す必要はありません。

クエリが終了したら、esp.sqlClose 関数を使用してサーバーからクエリを削除します。

```
code,err = esp.sqlClose(handle)
```

この関数は 2 つの値を返します。最初の値は、クローズが成功したかどうかを示すブール値です。2 番目の値には、失敗の原因に関する情報を提供するエラーメッセージが含まれています。

前述したように、すべての行を取得したら、ESP サーバーがクエリを閉じる必要はありません。

以下は、Postgres から大規模な株取引(一度に 100 件の取引)を読み取り、シンボルとブローカーを出力する例です。

```
handle = esp.sqlOpen(conn="db",query="select * from trades")

if (handle["code"] == True):
   while True:
      trades,count = esp.sqlNext(handle,100)

      if (trades == None):
         break

      for trade in trades:
         print("symbol=" + trade["symbol"] + ", broker="..trade["broker"])
```

データの挿入

esp.sqlInsert 関数を使用して、データベースにレコードを挿入します。

```
resp = esp.sqlInsert(options)
```

関数において、**options** には次のフィールドが含まれます。

- conn は、使用する SQL 接続の名前です。
- table は、データを挿入するデータベーステーブルです。
- data には、挿入するレコードを含む 1 つ以上の Python オブジェクトが含まれます。
- context は、接続が存在する Python コンテキストを指定するオプションの値です。

株式取引を処理している場合は、ソースウィンドウで Python サブスクライバーコネクタを使用して受信取引を取得し、データベースに書き込むことができます。

```
<window-source name="trades" insert-only="true">
  <schema>
    <fields>
      <field key="true" name="id" type="int64"/>
      <field name="symbol" type="string"/>
      <field name="currency" type="int32"/>
      <field name="time" type="int64"/>
      <field name="msecs" type="int32"/>
      <field name="price" type="double"/>
      <field name="quant" type="int32"/>
      <field name="venue" type="int32"/>
      <field name="broker" type="int32"/>
      <field name="buyer" type="int32"/>
      <field name="seller" type="int32"/>
      <field name="buysellflg" type="int32"/>
    </fields>
  </schema>
  <connectors>
    <connector class="python" name="sub">
      <properties>
        <property name="type">sub</property>
        <property name="code"><![CDATA[
          def subscribe(data):
            response = esp.sqlInsert(conn="db",table="trades",data=data)
            if (response.code == False):
              print(str(response))
        ]]></property>
      </properties>
    </connector>
  </connectors>
</window-source>
```

esp.sqlInsert からの戻り値は、次のフィールドを含む Python オブジェクトです。

- code は挿入のステータスです: 成功の場合は **true** 、失敗の場合は **false** です。

- message には、障害発生時の障害に関連する情報が含まれています。
- object は、障害が発生したときの、障害が発生したレコードのデータです。

データの更新

esp.sqlUpdate 関数を使用して、データベースに更新します。

```
resp = esp.sqlUpdate(options)
```

Python コンポーネントを通じてデータベースを更新するには 2 つの方法があります。1 つ目の方法は、更新するフィールドを含むデータのみを送信することです。このアプローチを使用する場合、次のフィールドが options パラメータで受け入れられます:

- conn は、使用する SQL 接続の名前です。
- table は、更新するデータベーステーブルです。
- data には、更新を含む 1 つ以上の Python オブジェクトが含まれます。
- upsert は、レコードが存在しない場合にレコードの挿入を試みるオプションのブール値です。デフォルト値は **false** です。
- context は、接続が存在する Python コンテキストを指定するオプションの値です。

サーバーはデータからデータベースキーを導出するため、データには各オブジェクトの少なくともすべてのキーフィールドが 含まれている必要があります。この更新メソッドは、関数呼び出しで複数の Python オブジェクトを受け入れることができます。各オブジェクトはデータベースに対して個別の更新リクエストを生成します。

たとえば、株式取引を処理していて、各シンボルの最低価格と最高価格を保存したいとします。次のテーブルを作成します:

```
create table trades_data
(
   symbol text,
   min_price float,
   max_price float,
   range float,
   primary key (symbol)
);
```

次のコードストリームは、各シンボルの現在の最低価格と最高価格を取引して維持します。

```
resp = esp.sqlUpdate(conn="db",
                     table="trades_data",
                     upsert=True,
                     data=[
                      {"symbol":"ABC","min_price":100,"max_price":120,"range":20},
                      {"symbol":"XYZ","min_price":10,"max_price":20,"range":10}
                     ])
```

upsert の値が **true** であることに注意してください。その結果、レコードが存在しない場合、サーバーはレコードを追加します。

更新を実行する 2 番目の方法は、更新するデータベースレコードを識別する SQL WHERE 句を含む単一の Python オブジェクトを追加することです。次のコードでは、リクエストのデータが WHERE 句に一致するすべてのレコードに適用されます。

```
resp = esp.sqlUpdate(conn="db",
                     table="trades_data",
                     data={"min_price":0,"max_price":0,"range":0},
                     where="1=1")
```

データ内のキーフィールドは無視されます。前の例の trades_data テーブルが初期化され、以前の既存の値がゼロに設定されます。

esp.sqlUpdate からの戻り値は、次のフィールドを含む Python オブジェクトです。

- code は更新のステータスです: 成功の場合は **true** 、失敗の場合は **false** です。
- message は、障害が発生した場合にその障害に関連する情報です。
- object は、障害が発生したときの、障害が発生したレコードのデータです。

データの削除

esp.sqlDelete 関数を使用して、データベースからレコードを削除します。

```
resp = esp.sqlDelete(options)
```

この関数において、**options** は削除プロパティを含む Python テーブルです。

Python コンポーネントを使用してデータベースからレコードを削除するには、2 つの方法があります。1 つ目の方法は、削除するレコードのキーフィールドデータを含む 1 つ以上の Python オブジェクトを送信することです。このアプローチを使用すると、options パラメータは次のフィールドを受け入れます。

- conn は、使用する SQL 接続の名前です。
- table は、レコードを削除するデータベーステーブルです。
- data は、削除するレコードのキーフィールドを含む 1 つ以上の Python オブジェクトが含まれます。
- context は、接続が存在する Python コンテキストを指定するオプションの値です。

次のコードは、更新セクションの trades_data テーブルからいくつかのレコードを削除します。

```
resp = esp.sqlDelete(conn="db",
                     table="trades_data",
                     data=[{"symbol":"ABC"},{"symbol":"XYZ"}])
```

レコードを削除する 2 番目の方法は、削除するデータベースレコードを指定する SQL WHERE 句を追加することです。次のコードは、特定の条件(max_price > 10)を満たすすべてのレコードを trades_data から削除します。

```
resp = esp.sqlDelete(conn="db",table="trades_data",where="max_price > 10")
```

esp.sqlDelete からの戻り値は、次のフィールドを含む Python オブジェクトです。

- code は削除のステータスです: 成功の場合は **true** 、失敗の場合は **false** です。
- message は、障害が発生した場合にその障害に関連する情報です。
- object は、障害が発生したときのデータです。

Python モジュールの使用

Python モジュールは、プロジェクトレベルで存在する Python コードの自己完結型ブロックです。モジュール名を使用して、これらのモジュールをプロジェクト内の任意の Python エンティティにインポートできます。

python-module 要素を使用して、スニペットを定義します。詳細については、"python-module" (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

code 要素を使用して、スニペットのコードを指定します。

次のモジュールを考えてみましょう:

```
<project ...>
 <python-modules>
  <python-module name="module">
    <code><![CDATA[
      import esp
      class Module
      # Python code
    ]]>
    </code>
  </python-module>
 </python-modules>
  ...
 <contqueries>
   <contquery>
     ...
     <window-python...>
      ...
       <code><![CDATA[
            import esp
            from module import Module
            # Python code
       ]]>
       </code>
     </window-python>
    ...
   </contquery>
 </contqueries>
</project>
```

ESP サーバーのログをカプセル化して、任意の Python エンティティで利用できるようにするクラスが必要だとします:

```
<python-module name="logger">
  <code><![CDATA[

import esp

class Logger:

  def __init__(self,context):
    self._context = context if context != None else "mylogger"

  def error(self,msg,line = None):
    esp.logMessage(logcontext=self._context,message=msg,level="error",line=line)

  def warn(self,msg,line = None):
    esp.logMessage(logcontext=self._context,message=msg,level="warn",line=line)

  def fatal(self,msg,line = None):
    esp.logMessage(logcontext=self._context,message=msg,level="fatal",line=line)

  def info(self,msg,line = None):
    esp.logMessage(logcontext=self._context,message=msg,level="info",line=line)

  def debug(self,msg,line = None):
    esp.logMessage(logcontext=self._context,message=msg,level="debug",line=line)

  def trace(self,msg,line = None):
    esp.logMessage(logcontext=self._context,message=msg,level="trace",line=line)

  ]]></code>
</python-module>
```

イベントに入ってくる数値を分割する Python ウィンドウがあるとします。除数が 0 の場合、このモジュールの Logger クラスを使用してサーバーに ERROR メッセージを記録する必要があります。

```
<window-python name="python" events="create">
 <schema>
  <fields>
   <field name="id" type="string" key="true"/>
   <field name="dividend" type="int32"/>
   <field name="divisor" type="int32"/>
   <field name="quotient" type="double"/>
  </fields>
 </schema>
 <copy>id,dividend,divisor</copy>
 <use>dividend,divisor</use>
 <code><![CDATA[
  from logger import logger

  def create(data):

    e = {}
    dividend = data["dividend"]
    divisor = data["divisor"]

    if (divisor == 0):
      logger = Logger("modules.example")
```

```
        logger.error("division by 0, " + str(dividend) + "/" + str(divisor))
        return None

    e["quotient"] = dividend / divisor

    return(e)
 ]]></code>
</window-python>
```

Python ウィンドウは、ロガーモジュールから Logger クラスをインポートします。0 による除算が発生すると、サーバーログにエラーが表示されます。

```
{
    "level":"ERROR",
    "message":"{6c00732f-5d9d-4203-9c42-731d4e442b30}[XMLServer0001] division by
0, 31\/0",
    "properties":
    {
        "caller":"Python.cpp:1910",
        "logger":"modules.example",
        "thread":"00000033"
    },
    "timeStamp":"2023-09-13T16:19:04.891000-04:00"
}
```

Python スニペットの使用

Python スニペットは、プロジェクトレベルで存在する Python コードの自己完結型ブロックです。他の Python インスタンスは、Python スニペット内で指定されたプロパティにアクセスできます。

python-snippet 要素を使用して、スニペットを定義します。詳細については、*"python-snippet" (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*を参照してください。

code 要素を使用して、スニペットのコードを指定します。

コードはインラインで記述することも、外部ファイルに記述することもできます。外部ファイルにある場合は、ファイル属性を使用してそのファイルを参照できます。

```
<code file="mycode.py" />
```

次のスニペットを考えてみましょう:

```
<project ...>
 <python-snippets>
  <python-snippet name="access_token" init="initfunc" destroy="cleanup">
    <code><![CDATA[

        import esp
        from pathlib import Path

        def initfunc():
```

```
        print("initializing snippet")

      def cleanup():
        print("cleaning up snippet")

      def load():
        token = Path("token.txt").read_text()
        esp.setProperty(name="token",value=token)

      load()

    ]]></code>
  </python-snippet>
 </python-snippets>
  ...
</project>
```

スニペットは初期化関数とクリーンアップ関数を実行した後、**token.txt** という名前のファイルを開き、そこからトークンを読み取ります。次に、スニペットは ESP サーバーに token と呼ばれるプロパティを設定します。このプロパティには、他の Python エンティティからアクセスできます。

注: 関数の内部にないコードは、Python エンティティが作成されて読み込まれるときに実行されます。

次の Python コネクタを考えてみましょう。Python スニペットによって設定されたトークンを使用して REST 要求を作成します。

```
<connector class="python" name="pub">
 <properties>
  <property name="type">pub</property>
  <property name="interval">5 seconds</property>
  <property name="code"><![CDATA[

    token = esp.getProperty(name="token")

    def publish():
      # use token to make REST request, parse response, and get events
      return {"events":data, "done": False}

  ]]></property>
 </properties>
</connector>
```

注: esp.getProperty 関数は、Python プロパティの名前を取得します。

Python スニペットを使用すると、指定した関数を指定した間隔で呼び出すスレッドを開始することもできます。次のコードでは、一定期間後にトークンの有効期限が切れると、スニペットが 5 分ごとに新しいトークンを取得してプロパティを設定するように、前のスニペットを改訂しています。

```
<project ...>
 <python-snippets>
  <python-snippet name="largetrades">
    <python-thread func="load" interval="5 minutes" />
```

```
        <code><![CDATA[

          def load():
            from pathlib import Path
            contents = Path("token.txt").read_text()
            esp.setProperty(name="token",value=token)
            return True

          load()

        ]]></code>
      </python-snippet>
    </python-snippets>
    ...
  </project>
```

関数からの戻りコードは、スレッドを実行し続けるかどうかを ESP サーバーに伝えるブール変数です。**true** を返すと、スレッドは引き続き実行されます。**false** を返すと、スレッドは終了します。

これで、Python コネクタは発行するたびにトークンを取得して、リクエストに対応する新しいトークンを確実に取得できるようになります。

```
  <connector class="python" name="pub">
    <properties>
      <property name="type">pub</property>
      <property name="code"><![CDATA[

        def publish():
          local   token = esp.getProperty(name="token")
          # use token to make REST request, parse response, and get events
          return {"events":data, "done": False}

      ]]></property>
    </properties>
  </connector>
```

Python スニペットのコードは、XML でスニペットが宣言されている順序でロードされます。これは、別のスニペットの前に宣言されたスニペットには、コードのロード時に使用できる変数がない可能性があることを意味します。

次のコードを考えてみましょう。

```
  <python-snippets>
    <python-snippet name="thisSnippet">
      <code><![CDATA[

        esp.setProperty(name="theproperty",value="hello, world")

      ]]></code>
    </python-snippet>
    <python-snippet name="thatSnippet">
      <code><![CDATA[

        value = esp.getProperty(name="theproperty")
        print("VALUE IS: " + str(value))

      ]]></code>
```

```
    </python-snippet>
  </python-snippets>
```

これらのスニペットが指定どおりに含まれている場合、コードは ESP ログに次の出力を返します。

```
VALUE IS: hello, world
```

スニペットが転置されているとします:

```
<python-snippets>
  <python-snippet name="thatSnippet">
    <code><![CDATA[

      value = esp.getProperty(name="theproperty")
      print("VALUE IS: " + str(value))

    ]]></code>
  </python-snippet>
  <python-snippet name="thisSnippet">
    <code><![CDATA[

      esp.setProperty(name="theproperty",value="hello, world")

    ]]></code>
  </python-snippet>
</python-snippets>
```

thatSnippet のコードが thisSnippet のコードより前に実行されるため、次の出力が ESP ログに返されます。

```
VALUE IS: nil
```

Python コードの例外の処理

Python コードは例外が発生する可能性があります。init 関数からの例外により、プロジェクトの読み込みに失敗します。create 関数、heartbeat 関数、およびスニペット(スレッド化された場合)からの例外は、project 要素の on-script-exception 属性に基づいて処理されます。詳細については、*"project" (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*を参照してください。

10

式

式の概要

Expression Engine Language (EEL)の式を使用して、次を定義できます。

- フィルターウィンドウのフィルター条件
- 計算、集計、および結合ウィンドウでの非キーフィールド計算
- パターンウィンドウで関心があるイベント(EOI)のパターンロジックに一致
- ウィンドウ出力スプリッター/スロットの計算(たとえば、式を使用して生成されたイベントを送信する場所を評価します)

注: EEL コードでは、正規表現の中の特殊文字は、\\ でエスケープしなければなりません。

計算、集計、結合ウィンドウでの非キーフィールド計算を除くすべてのケースで、式の代わりに Lua または Python で記述された関数を使用できます。

パターンマッチングを除くすべての場合で、式のかわりに C 言語で記述されたユーザー定義関数を使用できます。

式の指定方法については、*式言語: リファレンスガイド*を参照してください。

それぞれのウィンドウを使用した式の作成と登録は、C で同等のユーザー定義関数を記述するよりも簡単です。ただし、式は関数よりも低速で実行されます。低待機時間のアプリケーションでは、ユーザー定義関数して、式の解析と処理のオーバーヘッドを最小限に抑えることができます。

可能であれば、プロトタイプ式を使用してください。結果に基づいて、必要に応じて最適化したり、関数に変換してください。ほとんどのアプリケーションでは、関数の代わりに式を使用しますが、パフォーマンスの向上が重要な場合は関数を使用できます。

各式ウィンドウとウィンドウスプリッターには、独自の式エンジンインスタンスがあります。式エンジンインスタンスは、ウィンドウによって処理される各イベントのウィンドウ式とスプリッター式を実行します。エクスプレッションエンジンが式ウィンドウまたはウィンドウスプリッターで使用される前に(つまり、それらのウィンドウにイベントストリームが入る前に)式エンジンを初期化できます。式エンジンの初期化は、式ウィンドウまたはウィンドウスプリッター式で使用される式エンジン変数の宣言および初期化に役立ちます。また、式で使用される正規表現を宣言するのにも役立ちます。

式ウィンドウとウィンドウスプリッターの式エンジンを初期化するには、XML コードの<expr-initialize>エレメントを使用します。たとえば、次の XML コードはスプリッターのユーザー定義関数を初期化します。

```
<expr-initialize>
    <udfs>
     <udf name='udf1' type='int32'>
      <![CDATA[private integer p
            p = parameter(1);
            return p%2]]>
     </udf>
    </udfs>
   </expr-initialize>
```

式エンジン言語の使用(EEL)

データ型マッピングについて

SAS Event Stream Processing API でサポートされているデータ型と式エンジン言語でサポートされているデータ型の間には、正確なデータ型マッピングは存在しません。

次の表に、サポートされているデータ型のマッピングを示します。

表 10.1　式データ型マッピングテーブル

イベントストリーム処理式	式	メモと制限
文字列(utf8)	文字列(utf8)	式に渡される文字列は、1024 バイトに切り捨てられることがあります。ウィンドウ固有の属性 exp-max-string= '*N*'または C++window method

イベントストリーム処理式	式	メモと制限
		dfESPwindow::set_EXP_strMax(int N)を使用してウィンドウの式文字列の最大サイズを設定します。
日付(2 番目の粒度)	日付(2 番目の粒度)	秒粒度
タイムスタンプ(マイクロ秒粒度)	日付(マイクロ秒の粒度)	dfExpressions の定数ミリ秒はサポートされていません。
INT32(32 ビット)	整数(64 ビット)	dfExpressions の 64 ビット変換
INT32(64 ビット)	整数(64 ビット)	64 ビット、変換なし
Double(64 ビット IEEE)	Real(192 ビット固定小数)	Real192 ビット固定小数点、Double64 ビット float
金額(192 ビット固定小数点)	Real(192 ビット固定小数)	192 ビット固定小数点、変換なし

式でのイベントメタデータの使用

SAS Event Stream Processing では、イベントのメタデータにアクセスするために使用できる予約語のセットが提供されます。これらの予約語は、フィルター式、計算式、および結合ウィンドウ式およびウィンドウ出力スプリッター式で使用できます。パターンウィンドウは挿入専用なので、メタデータはパターンウィンドウ式では使用できません。

表 10.2 イベントのメタデータにアクセスするための予約語

予約語	演算コード
ESP_OPCODE 指定されたウィンドウでイベントの演算コードを取得するには、この予約語を使用します。	I-挿入 U-更新 P-アップサート D-削除 SD-安全な削除

予約語	演算コード
	安全な削除では“キーが見つかりません“という ERROR は生成されません。 **注:** 値は大文字と小文字が区別されます。
ESP_FLAGS 指定されたウィンドウ内のイベントのフラグを取得するには、この予約語を式で使用します。	N-標準 P-部分 R-保持の削除

式エンジン言語のグローバル関数の使用

式エンジン言語では、ユーザー定義関数(UDF)とも呼ばれるグローバル関数がサポートされています。それらをグローバル関数として登録し、任意の式ウィンドウまたはウィンドウスプリッター式から参照することができます。

グローバル関数を登録できる 2 つの SAS Event Stream Processing 関数があります。

- dfESPexpression_window::regWindowExpUDF(*udfString*, *udfName*, *udfRetType*)
- dfESPwindow::regSplitterExpUDF(*udfString*, *udfName*, *udfRetType*)

ウィンドウスプリッターまたは式ウィンドウのグローバル関数を登録した後、スプリッター式またはウィンドウ式で *udfName* を参照できます。イベントが処理される際に *udfName* は *udfString* に置き換えられます。

フィルター、計算、結合、パターンの式ウィンドウでは、グローバル関数の使用がサポートされています。集計ウィンドウはグローバル関数をサポートしていません。なぜなら、それらの出力フィールドは集計関数を使用した作成専用であるからです。すべてのウィンドウは、指定されたウィンドウで出力スプリッターのグローバル関数をサポートします。

SAS Data Quality 関数の使用

イベントストリーム処理式は、SAS Data Quality 関数の使用をサポートしています。次の関数については、このドキュメントの後半で詳しく説明します。

- dq.case
- dq.extract
- dq.gender
- dq.getlasterror
- dq.identify

- dq.initialize
- dq.loadqkb
- dq.matchcode
- dq.matchscore
- dq.parse
- dq.pattern
- dq.standardizeL
- dq.token
- dq.tokenvalue
- dq.value

SAS Event Stream Processing でこれらの関数を使用するには、次のように 2 つの環境変数を設定する必要があります。

- DFESP_QKB を SAS Data Quality インストール配下の共有フォルダーに設定します。Linux システムに SAS Data Quality をインストールした後、この共有フォルダーは***/QKB_root*/data/ci/*qkb_version_number_no_dots*** です(たとえば**/QKB/data/ci/22**)。
- DFESP_QKB_LIC を SAS Data Quality ライセンスファイルへの完全パス名に設定に変更します。

SAS Event Stream Processing 用に SAS Data Quality を設定した後、これらの関数をイベントストリーム処理式に含めることができます。これらの関数は、通常、計算ウィンドウの非キーフィールド計算式のイベントフィールドを正規化するために使用されます。

> 重要　SAS Data Quality ソフトウェアを Kubernetes 環境の SAS Event Stream Processing で使用するには、SAS Event Stream Processing が利用できる永続ボリュームにソフトウェアをインストールする必要があります。

11

日時

時間の節約と処理方法

SAS Event Stream Processing では、UNIX エポック時間(1970 年 1 月 1 日 00:00 UTC)を基準にして、すべての時間の値が保存されます。スタンプデータは 1 マイクロ秒単位で処理されます。日付データは 1 秒単位で処理されます。時間は int64 値として保存されます。

日付と時間の処理方法

コネクタとアダプター

コネクタとアダプターは、人間が読み取れるタイムスタンプ(01/01/2019 09:30:30 AM など)と UNIX エポック時間との間で変換できます。時間を変換できるコネクタとアダプターの場合、日付または時刻の値のフォーマットを記述する dateformat キーが存在します。C ++で記述されたコネクタおよびアダプターの場合、このフォーマットは strftime にあります。Java で記述されたアダプターの場合、このフォーマットは Java 8 SimpleDateFormat にあります。

timestamp として定義されているフィールドの場合、マイクロ秒の精度がサポートされています。strftime と SimpleDateFormat のどちらも、フォーマット文字列でマイクロ秒はサポートされていません。ピリオドを含む秒の値を持つ時間の場合、ピリオドの後の値はマイクロ秒に変換されます。

たとえば、次のタイムスタンプを考えてみましょう。

10/16/2019 16:17:30.123456

C ++で記述されたコネクタまたはアダプターの場合、dateformat 文字列は次のとおりです。

“%m/%d/%Y %H:%M:%S”

Java で記述されたアダプターの場合、dateformat 文字列は次のとおりです。

“mm/DD/yyyy HH:mm:ss”

タイムスタンプ 10/16/2019 16:17:30.123456 は、マイクロ秒値 1571242650123456 に変換されます。

すべての 日時データは、ソースでどのようにタイムスタンプが付けられたかどうかに関係なく、協定世界時(UTC)に変換されます。UTC はグリニッジ標準時(GMT)の後継です。

ウィンドウ

ウィンドウが時間データを含むイベントを処理するときは常に、生データは UNIX エポック UTC 形式です。どのウィンドウでも、選択したとおりに生データをフォーマットできます。

式エンジン言語

式エンジン言語(EEL)は異なる時間形式を使用し、値を数値として保存します。数値は 1899 年 12 月 30 日からの日数を表します。時間、分、秒、ミリ秒は分数に変換され、1 時間 = 1/4 単位、1 分 = 1/(24*60)単位のようになります。

SAS Event Stream Processing では、日付または時刻の値が使用されている場合、UNIX エポック形式と式エンジン言語形式の間で変換されます。式エンジン言語で値を変更するには、式エンジン言語形式を使用します。

たとえば、次の式は、入力日付から 1 時間を減算します。

return (input_date - (1.0/24.0))

SAS Event Stream Processing の時間と整合性を保つには、式エンジン言語で日付フィールドを作成するときに、today()関数の代わりに todayGMT()関数を使用します。

SAS Micro Analytic Service モジュール

SAS Micro Analytic Service モジュールを DS2 や Python で使用した場合、日付や時刻の値がネイティブ形式に変換されません。代わりに、値は 1960 年 1 月 1 日から始まる SAS エポック形式に変換されます。値は DS2 では BIGINT として、Python では long として渡されます。詳細については、“データ型マッピングについて”を参照してください。

UNIX エポック時間と SAS エポック時間の違いは次のとおりです。

- EPOCH_DIFF_SECONDS: 315619200
- EPOCH_DIFF_MICROSECONDS: 315619200000000

したがって、入力日付の値から UNIX エポック時間を取得する Python プログラムでは、その値から 315619200 を減算します。

関数ウィンドウおよび通知ウィンドウで、データや時間データにアクセスしたり、操作するには、次の一般的な関数を使用することができます。

表 11.1 時間関数

関数	説明
EVENTTIMESTAMP	現在のイベントのタイムスタンプを返します
TIMECURRENT	現在の時刻を返します。
TIMEDAYOFMONTH	現在または指定された時刻の日付を返します。
TIMEDAYOFWEEK	現在または指定された時刻の曜日を返します。
TIMEDAYOFYEAR	現在または指定された時刻の通日(DOY)を返します。
TIMEGMTTOLOCAL	引数に指定されている GMT を現地時間に変換します。
TIMEGMTSTRING	最初の引数で表される GMT 時間を出力します。
TIMEHOUR	現在または指定された時刻の時刻を返します。
TIMEMICRO	指定された引数のマイクロ秒数、または引数が指定されていない場合は 1970 年 1 月 1 日からのマイクロ秒数を返します。
TIMEMILLI	指定された引数のミリ秒数、または引数が指定されていない場合は 1970 年 1 月 1 日からのマイクロ秒数を返します。

関数	説明
TIMEMINUTE	現在または指定された時刻の分を返します。
TIMEMINUTEOFDAY	現在または指定された時刻の、その日に入ってからの分数を返します。
TIMEPARSE	最初の引数で指定された時刻を表す文字列を返します。
TIMESECOND	現在または指定された時刻の秒を返します。
TIMESTAMP	現在の時刻を文字列で返します。
TIMESTRING	最初の引数で表される時刻を返します。
TIMESECONDOFDAY	現在または指定された時刻の、その日に入ってからの秒数を返します。
TIMETODAY	現地時間を基準にして、現在の日の最初の秒を表す値を返します。
TIMEYEAR	1900 年から現在または指定された時刻までの年数を返します。

12

配列関数

配列関数

式エンジン言語(EEL)では、文字列、整数、日付、ブール値、実数などの単純な型の配列を作成することができます。現在、配列型に適用される関数は 3 つあります。DIM、GET、SET です。

ディクショナリ

DIM 関数

配列の作成、サイズ変更、またはサイズの決定を行います。パラメーターを指定すると、配列のサイズ変更または作成が行われます。新しいサイズが返されます。

カテゴリ: 配列

返されるデータの種類: 整数

構文

arrayName.**DIM**<(*newsize*)>

必須引数

arrayName
プロセスの中で先に宣言した配列の名前です。

オプション引数

newsize
配列のオプションの数値サイズ(ディメンション)です。これは、数値定数、フィールド名、または式で指定することができます。

詳細

配列のサイズ設定やサイズ変更には DIM 関数を使用します。配列の作成、サイズ変更、またはサイズの決定を行います。パラメーターを指定すると、配列の作成またはサイズ変更が行われます。サポートされている配列タイプは次の通りです。

- 文字列
- 整数
- 日付
- ブール値
- 実数

例

```
// declare the string array
string array string_list
// Set the dimension of the String_List array to a size of 5
rc = string_list.dim(5) // outputs 5
// <omitted code to perform some actions on the array>
// Re-size the array size to 10
rc = string_list.dim(10) // outputs 10
// Query the current size
re = string_list.dim() // outputs 10
```

GET 関数

配列内の指定されたアイテムの値を取得します。返される値は、配列の値です。

カテゴリ: 配列

返されるデータの種類: 整数

構文

array name.**GET**(<*n*>)

必須引数

array name
プロセスの中で先に宣言した配列の名前です。

オプション引数

n
コンテンツを取得する配列エレメントのインデックスです。これは、数値定数、フィールド名、または式で指定することができます。

詳細

GET 関数は、配列の特定のエレメントの値を返します。

例

例 1

```
// Declare the string array "string_list" and Integer "i"
string array string_list
integer i

// Set the dimension of string_list array to 5 and initialize the counter (i) to 1
string_list.dim(5)
i=1

// Set and print each entry in the array, incrementing the counter by 1
while(i<=5)
begin
  string_list.set(i,"Hello")
  print(string_list.get(i))
  i=i+1
end
```

例 2

```
string array string_list
integer i

// set the dimension
string_list.dim(5)
i=1

// set and print each entry in the array
while(i<=5)
begin
    string_list.set(i,"Hello")
    print(string_list.get(i))
    i=i+1
end

// resize the array to 10
string_list.dim(10)
while(i<=10)
begin
    string_list.set(i,"Goodbye")
    print(string_list.get(i))
    i=i+1
end
```

SET 関数

配列内のアイテムの値を設定します。返される値は、配列内の指定された要素の古い値です。

カテゴリ:	配列
返されるデータの種類:	整数

構文

array name.**SET**(<*n*,*"string"*>)

必須引数

array name
プロセスの中で先に宣言した配列の名前です。

オプション引数

n
値を設定するディメンションの番号で、これは数値定数、フィールド名、または式で指定できます。

"string"
配列エレメントに配置したい値です。これは文字列定数、フィールド名、または式で指定できます。

詳細

SET 関数は、配列のエントリの値を設定します。

例

例 1

```
// Declare the string array "string_list"
// Set the dimension of string_list array to 5
string array string_list
string_list.dim(5)

// Set the first string element in the array to "Hello"
string_list.set(1,"Hello")
```

例 2

```
string array string_list
string_list.dim(5)
// sets the first string element in the array to hello
string_list.set(1,"hello")
```

13

データ品質関数

データ品質関数

式エンジン言語 (EEL)は、データ品質オブジェクトをサポートしています。データ品質を利用して、EEL ノード内からリストアップされた関数(オブジェクトメソッド)を実行できます。EEL 内でデータ品質関数を使用する利点としては、一致定義を動的に変更したり、別の列から一致定義を読み取ったり、異なる定義を設定したりすることが挙げられます。

ディクショナリ

DQ.CASE 関数

大文字と小文字のルール (大文字、小文字、または先頭の一文字のみ大文字) を文字列に適用します。この関数は、SAS Quality Knowledge Base (QKB)の大文字小文字定義を使用して、コンテキスト特有の大文字小文字ロジックも適用します。

カテゴリ:	データ品質
返されるデータの種類:	整数
注:	返される値は、1=成功、0=エラーのブール値です。

構文

DQ.CASE(case_def, casing_type, input, result)

必須引数

casing_type
　適用される大文字小文字の種類を指定する整数の数値定数[1=大文字、2=小文字、3=先頭の一文字のみ大文字]。

input
　入力値または入力フィールド名を表す文字列です。

result
　出力フィールド名を表す文字列です。

オプション引数

case_def
　QKB の大文字小文字定義の名前を表す文字列。デフォルトの大文字小文字アルゴリズムを使用するために空の文字列を渡します。

詳細

DQ.CASE 関数は、入力文字列に大文字と小文字のルールを適用し、その結果をフィールドに出力します。

この関数は、データ品質クラスのメンバーです。データ品質オブジェクトを変数として宣言し、関数 DQ_INITIALIZE を呼び出して初期化する必要があります。

大文字、小文字、先頭の一文字のみ大文字の 3 種類を指定できます。大文字または小文字を指定した場合、この関数は Unicode の大文字または小文字のマッピングを入力文字列の文字に適用します。propercasing が指定されている場合、この関数は、各単語の最初の文字に大文字のマッピングを適用し、残りの文字に小文字のマッピングを適用します。

呼び出し元は、大文字小文字定義の使用を呼び出すことができます。大文字小文字定義は、コンテキスト固有の大文字小文字ロジックを含む QKB のオブジェクトです。たとえば、名前データの先頭一文字を大文字にする目的で実装された大文字小文字定義を使用して、文字列"Mcdonald"を"McDonald"に変換できます。QKB で利用可能な大文字小文字定義については、QKB のドキュメントを参照してください。大文字小文字定義を使用したくない場合は、大文字小文字定義パラメーターに空文字列を入力することで、大文字小文字定義名を省略できます。この場合、先に説明したように、汎用 Unicode 大文字小文字のマッピングが入力文字列に適用されます。

注: 大文字小文字定義を使用したい場合は、DQ.CASE を呼び出す前に DQ.LOADQKB を呼び出す必要があります。関数 DQ.LOADQKB は、QKB の内容をメモリにロードし、その QKB をデータ品質オブジェクトにリンクします。これにより、DQ.CASE は指定する大文字小文字定義にアクセスできるようになります。

例

```
dq dataq
 string output
 dataq = dq_initialize()
 dataq.case("", 1, "ronald mcdonald", output)
 // outputs "RONALD MCDONALD"

dataq.case("", 3, "ronald mcdonald", output)
 // outputs "Ronald Mcdonald"

dataq.loadqkb("ENUSA")
 dataq.case("Proper (Name)", 3, "ronald mcdonald", output)
 // outputs "Ronald McDonald"
```

DQ.EXTRACT 関数

文字列から属性を抽出します。

カテゴリ:	データ品質
返されるデータの種類:	文字
注:	戻り値は、extract 関数からの値、トークン、またはトークン値です。

構文

DQ.EXTRACT*definition, string*

必須引数

definition
QKB 内の抽出定義の名前を表す文字列。

string
抽出する必要のある属性を表す文字列。

詳細

DQ.EXTRACT 関数は、属性を文字列からトークンに抽出します。最初のパラメーターは、QKB 抽出定義の名前です。2 つ目は属性が抽出された文字列です。この関数は、作成されたトークンの数を返します。失敗した場合は 0 を返します。

例

```
dq dataq
string output
integer o
integer i

/* Initialize DQ */
dataq = dq_initialize()
dataq.loadqkb("EN")

/* Extract using the "Product Attributes" extraction definition (using QKB PD 2012A) */
o = dataq.extract("Product Attributes", "DOOR RANCHERO WOOD 16X8 WHT")

/* print all of the tokens we got */
print (o & " tokens filled")
for i = 1 to o
begin
    dataq.token(i, output)
    print ("token #" & i & " = " & output)
    dataq.value(i, output)
    print ("value #" & i & " = " & output)
end
/* to get a token's value by its name... */
dataq.tokenvalue("Colors", output)
print ("Colors = " & output)
```

DQ.GENDER 関数

QKB の性別分析定義を使用して、個人の名前の性別を決定します。

カテゴリ: データ品質

返されるデータの種類: 整数

注: 返される値は、1=成功、0=エラーのブール値です。

構文

DQ.GENDER(*gender_def, input, result*)

必須引数

gender_def
　QKB の性別分析定義の名前を表す文字列です。

input
　入力値または入力フィールド名を表す文字列です。

result
　出力フィールド名を表す文字列です。

詳細

DQ.GENDER 関数は、個人の名前を表す文字列を解析し、名前の性別を決定します。

この関数は、データ品質クラスのメンバーです。データ品質オブジェクトを変数として宣言し、関数 DQ_INITIALIZE を呼び出して初期化する必要があります。その後、メンバー関数 DQ.LOADQKB を呼び出して、QKB の内容をメモリにロードし、その QKB をデータ品質オブジェクトにリンクする必要があります。そして、データ品質オブジェクトは、QKB ロケール設定と QKB ロケール設定の情報を保持します。

DQ.GENDER を呼び出す際には、性別分析定義の名前を指定する必要があります。性別分析定義は、入力された名前文字列の性別を決定するために使用される参照データとロジックを含む QKB 内のオブジェクトです。QKB で利用可能な性別分析の定義については、QKB のドキュメントを参照してください。

例

```
dq dataq
 string output
```

```
dataq = dq_initialize()
dataq.loadqkb("ENUSA")
dataq.gender("Name", "John Smith", output)
 // outputs "M"

 dataq.gender("Name", "Jane Smith", output)
 // outputs "F"

 dataq.gender("Name", "J. Smith", output)
 // outputs "U" (unknown)
```

DQ.GETLASTERROR 関数

データ品質オブジェクトで発生した最新のエラーを説明する文字列を返します。

カテゴリ:	データ品質
返されるデータの種類:	文字
注:	戻り値は、エラーメッセージを含む文字列です。

構文

DQ.GETLASTERROR(<>)

詳細

DQ.GETLASTERROR 関数は、データ品質クラスのメンバーです。データ品質オブジェクトが遭遇した最新のエラーを記述したエラーメッセージを返します。他のデータ品質メンバー関数の呼び出し中にエラーが発生した可能性があります。

プログラマにとってのベストプラクティスは、データ品質の呼び出しごとに結果コードを確認することです。結果コードが失敗を示す場合は、DQ.GETLASTERROR を使用して関連するエラーメッセージを取得します。

例

```
dq dataq
 integer rc
 string errmsg
dataq = dq_initialize()
 rc = dataq.loadqkb("XXXXX")
 // an invalid locale name -- this will cause an error

 if (rc == 0) then
```

```
    errmsg = dq.getlasterror()
// returns an error message
```

DQ.IDENTIFY 関数

QKB の ID 分析定義を使用して、文字列のコンテキストを識別します。

カテゴリ:	データ品質
返されるデータの種類:	整数
注:	返される値は、1=成功、0=エラーのブール値です。

構文

DQ.IDENTIFY(*ident_def, input, result*)

必須引数

ident_def
　QKB 内の識別分析定義の名前を表す文字列。

input
　入力値または入力フィールド名を表す文字列です。

result
　出力フィールド名を表す文字列です。

詳細

DQ.IDENTIFY 関数は文字列を解析し、文字列のコンテキストを決定します。コンテキストは、名前、住所、電話などの論理型データを指します。

この関数は、データ品質クラスのメンバーです。データ品質オブジェクトを変数として宣言し、関数 DQ_INITIALIZE を呼び出して初期化する必要があります。その後、メンバー関数 DQ.LOADQKB を呼び出して、QKB の内容をメモリにロードし、その QKB をデータ品質オブジェクトにリンクする必要があります。そして、データ品質オブジェクトは、QKB ロケール設定と QKB ロケール設定の情報を保持します。

DQ.IDENTIFY を呼び出す際には、識別分析定義の名前を指定する必要があります。識別分析定義は、入力文字列のコンテキストを識別するために使用される参照データとロジックを含む QKB 内のオブジェクトです。QKB で利用可能な識別分析定義については、QKB のドキュメントを参照してください。

注: 各識別分析定義に対して、出力される可能性のあるコンテキストの小規模なセットがあります。その定義がどのコンテキストを識別できるかについては、QKB ドキュメントの識別分析定義の説明を参照してください。

例

```
dq dataq
 string output
 dataq = dq_initialize()
dataq.loadqkb("ENUSA")
dataq.identify("Individual/Organization", "John Smith", output)
 // outputs "INDIVIDUAL"

dataq.identify("Individual/Organization", "DataFlux Corp", output)
 // outputs "ORGANIZATION"
```

DQ_INITIALIZE 関数

データ品質オブジェクトをインスタンス化して初期化します。

カテゴリ:	データ品質
返されるデータの種類:	文字
注:	戻り値は、データ品質オブジェクトの初期化されたインスタンスです。

構文

DQ_INITIALIZE(<>)

詳細

DQ_INITIALIZE は、データ品質オブジェクトをインスタンス化して初期化します。このオブジェクトは、データ品質クラス関数を呼び出す場合に使用できます。

例

```
dq dataq
dataq = dq_initialize()
```

DQ.LOADQKB 関数

定義を QKB からメモリにロードし、それらの定義をデータ品質オブジェクトにリンクします。

カテゴリ:	データ品質
返されるデータの種類:	整数
注:	返される値は、1=成功、0=エラーのブール値です。

構文

DQ.LOADQKB(*locale*)

必須引数

locale
QKB がサポートするロケールを表す 5 文字のロケールコード名です。

詳細

関数 DQ.LOADQKB は、データ品質クラスのメンバーです。データ品質オブジェクトを変数として宣言し、関数 DQ_INITIALIZE を呼び出して初期化する必要があります。関数 DQ.LOADQKB は、初期化後に呼び出すことができます。

DQ.LOADQKB 関数は、QKB から定義をメモリにロードし、それらの定義をデータ品質オブジェクトにリンクします。定義は、コンテキスト依存のロジックと参照データを使用して文字列の分析や変換を行う呼び出し可能なオブジェクトです。定義は他の dq 関数のパラメーターとして使用されます。

DQ.LOADQKB を呼び出す際に、ロケールコードを指定する必要があります。このロケールコードは、そのロケールの言語や国の ISO コードを表す 5 文字の文字列です。QKB でサポートされているロケールのコードのリストについては、QKB のドキュメントを参照してください。

注: DQ.LOADQKB を呼び出すたびに指定できるロケールコードは 1 つだけです。そのロケールに関連付けられた定義のみがメモリにロードされます。これは、一度に 1 つのロケールのみのサポートが、データ品質オブジェクトで使用するためにロードできることを意味します。複数のロケールの QKB 定義を使用するには、データ品質クラスの複数のインスタンスを使用するか、同じインスタンスに対して DQ.LOADQKB を複数回呼び出して、呼び出しごとに異なるロケールを指定する必要があります。

例

```
dq dataq_en
 // we instantiate two dq objects

dq dataq_fr
 string output_en
 string output_fr
dataq_en = dq_initialize()
dataq_fr = dq_initialize()

 dataq_en.loadqkb("ENUSA"
 // loads QKB support for locale English, US

 dataq_fr.loadqkb("FRFRA")
 // loads QKB support for locale French, France

 dataq_en.gender("Name", "Jean LaFleur", output_en)
 // output is 'U'

dataq_fr.gender("Name", "Jean LaFleur", output_fr)
 // output is 'M'
```

DQ.MATCHCODE 関数

QKB の一致定義を使用して、文字列のマッチコードを生成します。

カテゴリ:	データ品質
返されるデータの種類:	整数
注:	返される値は、1=成功、0=エラーのブール値です。

構文

DQ.MATCHCODE(*match_def, sensitivity, input, result*)

必須引数

match_def
: QKB の一致定義の名前を表す文字列です。

sensitivity
: マッチコードを生成する際に使用する感度レベルを指定する整数の数値定数[指定可能な値は 50〜95]。

input
: 入力値または入力フィールド名を表す文字列です。

result
出力フィールド名を表す文字列です。

詳細

DQ.MATCHCODE 関数は、入力文字列のマッチコードを生成し、マッチコードをフィールドに出力します。マッチコードは、入力文字列のファジー表現です。入力文字列と他の文字列とのファジー比較を行うために使用できます。

この関数は、データ品質クラスのメンバーです。データ品質オブジェクトを変数として宣言し、関数 DQ_INITIALIZE を呼び出して初期化する必要があります。その後、メンバー関数 DQ.LOADQKB を呼び出して、QKB の内容をメモリにロードし、その QKB をデータ品質オブジェクトにリンクする必要があります。そして、データ品質オブジェクトは、QKB ロケール設定と QKB ロケール設定の情報を保持します。

DQ.MATCHCODE を呼び出す際には、一致定義の名前を指定する必要があります。一致定義は、入力文字列のマッチコードを生成するために使用されるコンテキスト固有の参照データとロジックを含む QKB 内のオブジェクトです。QKB で利用可能な一致定義については、QKB のドキュメントを参照してください。

感度のレベルも指定する必要があります。感度は、マッチコードを生成する際に使用される曖昧さのレベルを示します。感度が高いということは、一致コードの曖昧さが少ないことを意味します(比較すると、偽陽性が少なく、偽陰性が多くなります)。感度が低いということは、一致コードの曖昧さが多いことを意味します(比較すると、偽陰性が少なく、偽陽性が多くなります)。感度パラメーターの有効範囲は 50～95 です。

例

```
dq dataq
 string output
dataq = dq_initialize()
 dataq.loadqkb("ENUSA")
 dataq.matchcode("Name", 85, "John Smith", output)
 // Outputs match code "4B~2$$$$$$$C@P$$$$$$"

dataq.matchcode("Name", 85, "Johnny Smith", output)
 // Outputs match code "4B~2$$$$$$$C@P$$$$$$"
```

DQ.MATCHSCORE 関数

2 つの入力文字列と一致定義の名前を処理し、一致する文字列の感度を出力します。

カテゴリ: データ品質

構文

DQ.MATCHSCORE(*definition_name, input1, input2, use_wildcards*)

必須引数

definition_name
　使用するマッチ定義の名前です。

input1
　チェックする最初の入力文字列です。

input2
　チェックする 2 番目の入力文字列です。

returns
　感度値を指定します。

use_wildcards
　スコアリングの目的で、生成されたマッチコードのワイルドカードを考慮する必要がある場合は true。

詳細

DQ.MATCHSCORE 関数は、2 つの入力文字列が同じマッチコードを生成する場合の最高感度値を決定します。

例

```
dq dataq;
dataq = dq_initialize();
dataq.loadqkb("ENUSA");

integer x;
x = dataq.matchscore("Name", "John Jones", "John J. Jones", false);
```

DQ.PARSE 関数

文字列を解析します。

カテゴリ:　データ品質

構文

DQ.PARSE(*definition, string*)

必須引数

definition
　解析されたデータを表す文字列です。

returns
　トークンの数です。

string
　トークンに解析する入力文字列です。

詳細

DQ.PARSE 関数は、入力文字列をトークンに解析します。最初のパラメーターは、QKB 解析定義の名前です。2 番目のパラメーターは、トークンを解析する文字列です。これは、作成されたトークンの数を返します。失敗した場合は 0 を返します。

例

```
/* Parse (using QKB CI 2013A) */
o = dataq.parse("Name", "Mr. John Q Public Sr")

/* print all of the tokens available */
print (o & " tokens filled")
for i = 1 to o
begin
   dataq.token(i, output)
   print ("token #" & i & " = " & output)
   dataq.value(i, output)
   print ("value #" & i & " = " & output)
end

/* Get a token value by the name. */
dataq.tokenvalue("Given Name", output)
print ("Given Name= " & output)
```

DQ.PATTERN 関数

QKB のパターン分析定義を使用して、文字列のパターンを生成します。

カテゴリ:	データ品質
返されるデータの種類:	整数
注:	返される値は、1=成功、0=エラーのブール値です。

構文

DQ.PATTERN(*pattern_def, input, result*)

必須引数

pattern_def
　QKB の一致定義の名前を表す文字列です。

input
　入力値または入力フィールド名を表す文字列です。

result
　出力フィールド名を表す文字列です。

詳細

DQ.PATTERN 関数は、入力文字列のパターンを生成し、パターンをフィールドに出力します。パターンは、入力文字列の中の文字を簡単に表現したものです。このようなパターンを使用して、テキスト文字列の集合に対してパターン頻度解析を行うことができます。

この関数は、データ品質クラスのメンバーです。データ品質オブジェクトを変数として宣言し、関数 DQ_INITIALIZE を呼び出して初期化する必要があります。その後、メンバー関数 DQ.LOADQKB を呼び出して、QKB の内容をメモリにロードし、その QKB をデータ品質オブジェクトにリンクする必要があります。そして、データ品質オブジェクトは、QKB ロケール設定と QKB ロケール設定の情報を保持します。

DQ.PATTERN を呼び出す際には、パターン解析定義の名前を指定する必要があります。パターン解析定義は、入力文字列のパターンを生成するために使用されるロジックを含む QKB 内のオブジェクトです。QKB で利用可能なパターン解析定義については、QKB のドキュメントを参照してください。

例

```
dq dataq
 string output
dataq = dq_initialize()
 dataq.loadqkb("ENUSA")
 dataq.pattern("Character", "abc123", output)
 // Outputs "aaa999"
```

DQ.STANDARDIZE 関数

QKB の標準化定義を使用して、文字列の標準を生成します。

カテゴリ:　データ品質

返されるデータの種類: 整数

注: 返される値は、1=成功、0=エラーのブール値です。

構文

DQ.STANDARDIZE(*stand_def, input, result*)

必須引数

stand_def
QKB 内の標準化定義の名前を表す文字列です。

input
入力値または入力フィールド名を表す文字列です。

result
出力フィールド名を表す文字列です。

詳細

DQ.STANDARDIZE 関数は、入力文字列に対して正規化された標準を生成し、標準をフィールドに出力します。

この関数は、データ品質クラスのメンバーです。データ品質オブジェクトを変数として宣言し、関数 DQ_INITIALIZE を呼び出して初期化する必要があります。その後、メンバー関数 DQ.LOADQKB を呼び出して、QKB の内容をメモリにロードし、その QKB をデータ品質オブジェクトにリンクする必要があります。そして、データ品質オブジェクトは、QKB ロケール設定と QKB ロケール設定の情報を保持します。

DQ.STANDARDIZE を呼び出す際は、標準化定義の名前を指定する必要があります。標準化定義は、入力文字列の標準を生成するために使用されるコンテキスト固有の参照データとロジックを含む QKB 内のオブジェクトです。QKB で利用可能な標準化定義については、QKB のドキュメントを参照してください。

例

```
dq dataq
 string output
 dataq = dq_initialize()
dataq.loadqkb("ENUSA")
dataq.standardize("Name", "mcdonald, mister ronald", output)
 // Outputs "Mr Ronald McDonald"
```

DQ.TOKEN 関数

解析関数または抽出関数からインデックスのトークン名を取得します。

カテゴリ: データ品質

返されるデータの種類: 文字

注: この関数は、成功した場合は true を返し、インデックスが範囲外の場合は false を返します。トークン名は、解析または抽出関数からの 2 番目のパラメーターで返されます。

構文

DQ.TOKEN(integer, string)

必須引数

integer
名前が必要なトークンのインデックスです。

string
トークン名を受け取る出力文字列です。

詳細

DQ.TOKEN 関数は、インデックスの抽出または解析トークン名を取得するために使用されます。この関数は、解析または抽出関数の後に続きます。

例

例 1

```
dq dataq
string output
integer o
integer i

/* Initialize DQ */
dataq = dq_initialize()
dataq.loadqkb("EN")
```

```
/* Extract using the "Product Attributes" extraction definition (using QKB PD 2012A) */
o = dataq.extract("Product Attributes", "DOOR RANCHERO WOOD 16X8 WHT")

/* print all of the tokens we got */
print (o & " tokens filled")
for i = 1 to o
begin
    dataq.token(i, output)
    print ("token #" & i & " = " & output)
    dataq.value(i, output)
    print ("value #" & i & " = " & output)
end
/* to get a token's value by its name... */
dataq.tokenvalue("Colors", output)
print ("Colors = " & output)
```

例 2

```
/* Parse (using QKB CI 2013A) */
o = dataq.parse("Name", "Mr. John Q Public Sr")

/* print all of the tokens available */
print (o & " tokens filled")
for i = 1 to o
begin
    dataq.token(i, output)
    print ("token #" & i & " = " & output)
    dataq.value(i, output)
    print ("value #" & i & " = " & output)
end

/* Get a token value by the name. */
dataq.tokenvalue("Given Name", output)
print ("Given Name= " & output)
```

DQ.TOKENVALUE 関数

最後の解析関数または抽出関数から属性を取得します。

カテゴリ:	データ品質
返されるデータの種類:	文字
注:	戻り値は、解析または抽出関数からのトークン値です。トークン値が見つかった場合は true、見つからなかった場合は false を返します。

構文

DQ.TOKENVALUE(<*token string, output string*>)

必須引数

token string
　トークンが見つかった場合は true、見つからなかった場合は false を返します。

output string
　そのトークンの値を表す文字列です。

詳細

DQ.TOKENVALUE 関数は、抽出または解析結果値を取得するために使用されます。最初のパラメーターはトークン名です。2 番目のパラメーターでそのトークンに保存されている属性を返します。成功した場合は true、失敗した場合は false を返します(たとえば、トークンが見つからなかった場合など)。この関数は、解析または抽出関数の後に続きます。

例

例 1

```
dq dataq
string output
integer o
integer i

/* Initialize DQ */
dataq = dq_initialize()
dataq.loadqkb("EN")

/* Extract using the "Product Attributes" extraction definition (using QKB PD 2012A) */
o = dataq.extract("Product Attributes", "DOOR RANCHERO WOOD 16X8 WHT")

/* print all of the tokens we got */
print (o & " tokens filled")
for i = 1 to o
begin
    dataq.token(i, output)
    print ("token #" & i & " = " & output)
    dataq.value(i, output)
    print ("value #" & i & " = " & output)
end
/* to get a token's value by its name... */
dataq.tokenvalue("Colors", output)
```

```
print ("Colors = " & output)
```

例 2

```
/* Parse (using QKB CI 2013A) */
o = dataq.parse("Name", "Mr. John Q Public Sr")

/* print all of the tokens available */
print (o & " tokens filled")
for i = 1 to o
begin
    dataq.token(i, output)
    print ("token #" & i & " = " & output)
    dataq.value(i, output)
    print ("value #" & i & " = " & output)
end

/* Get a token value by the name. */
dataq.tokenvalue("Given Name", output)
print ("Given Name= " & output)
```

DQ.VALUE 関数

最後の解析関数または抽出関数から値を取得します。

カテゴリ:	データ品質
返されるデータの種類:	整数
注:	返される値は、インデックスが有効な場合は true です。

構文

DQ.VALUE(*integer, string*)

必須引数

integer
　トークンのインデックスを表します。

string
　出力値を表す文字列です。

詳細

DQ.VALUE 関数は、抽出または解析結果を取得するために使用されます。最初のパラメーターはトークンのインデックスです。2 番目のパラメーターは、トークンに保存されている属性を取得します。この関数は、トークン値を取得できる場合に true を返します。失敗した場合は false を返します。この関数は、解析または抽出関数の後に続きます。

例

例 1

```
dq dataq
string output
integer o
integer i

/* Initialize DQ */
dataq = dq_initialize()
dataq.loadqkb("EN")

/* Extract using the "Product Attributes" extraction definition (using QKB PD 2012A) */
o = dataq.extract("Product Attributes", "DOOR RANCHERO WOOD 16X8 WHT")

/* print all of the tokens we got */
print (o & " tokens filled")
for i = 1 to o
begin
    dataq.token(i, output)
    print ("token #" & i & " = " & output)
    dataq.value(i, output)
    print ("value #" & i & " = " & output)
end
/* to get a token's value by its name... */
dataq.tokenvalue("Colors", output)
print ("Colors = " & output)
```

例 2

```
/* Parse (using QKB CI 2013A) */
o = dataq.parse("Name", "Mr. John Q Public Sr")

/* print all of the tokens available */
print (o & " tokens filled")
for i = 1 to o
begin
    dataq.token(i, output)
```

```
    print ("token #" & i & " = " & output)
    dataq.value(i, output)
    print ("value #" & i & " = " & output)
end

/* Get a token value by the name. */
dataq.tokenvalue("Given Name", output)
print ("Given Name= " & output)
```

14

日付と時間の関数

概要

日付は、整数、実数、ブール値、文字列とともに、EEL では基本的なデータ型と考えられています。EEL は他の基本データ型と同様に、日付に対する演算を行う関数を提供しています。

ディクショナリ

FORMATDATE 関数

文字列としてフォーマットされた日付を返します。

カテゴリ:	日時
返されるデータの種類:	文字列
注:	戻り値は、日付が文字列としてフォーマットされた文字列です。

構文

FORMATDATE(<datevar>,<format>)

必須引数

<**datevar**>
フォーマットする必要がある日付。これはフィールド名で指定できます。

<**format**>
適用する必要があるフォーマットを表す文字列。これは固定文字列、フィールド名、または式で指定できます。

詳細

format パラメーターには任意の文字列を含めることができますが、次の文字列は指定した値に置き換えられます。

- YYYY: 4 桁の年
- YY: 2 桁の年
- MMMM: 先頭一文字のみ大文字の 1 か月間
- MMM: 3 文字に略した月
- MM: 2 桁の月
- DD: 2 桁の日
- hh: 時
- mm: 分
- ss: 秒
- ms: ミリ秒

注: format パラメーターは大文字と小文字を区別します。

FORMATDATE 関数の日付は、曖昧さを避けるために ISO 8601 で指定されたフォーマット(YYYY-MM-DD hh:mm:ss:ms)でなければなりません。日付定数は#記号で始まり、#記号で終わることを覚えておいてください(例: #12-February-2010#)。

例

例 1

```
// Declare a date variable and initialize
// it to a value
```

```
date dateval
dateval = #2010-02-12#
// Declare the formatted date variable
string fmtdate
fmtdate = formatdate(dteval,"MM/DD/YY")
Results: 02/12/10
```

例 2

```
// Declare a date variable and initialize
// it to a value
date dateval
dateval = #2010-02-12#
// Declare the formatted date variable
string fmtdate
fmtdate = formatdate(dateval,"DD MMM YYYY")
Results: 12 Feb 2010
```

例 3

```
// Declare a date variable and initialize
// it to a value
date dateval
dateval = #2010-02-12#
// Declare the formatted date variable
string fmtdate
fmtdate = formatdate(dateval,"MMMM DD,YYYY")
Results: February 12, 2010
```

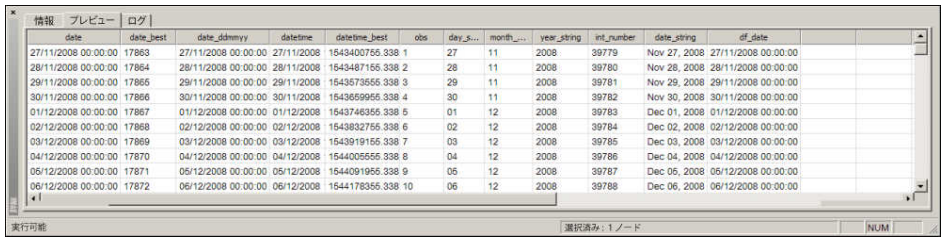
例 4

```
day_string=formatdate('date',"DD");
month_string=formatdate('date',"MM");
year_string=formatdate('date',"YYYY");

int_number='date';
date_string=formatdate(int_number,"MMM DD,YYYY");

df_date='date';
```

アウトプット 14.1　結果



TODAY 関数

現在の日時を返します。この関数は、現地のタイムゾーンを基準にしています。

カテゴリ:　日時

返されるデータの種類:　文字

注:　返される値は、現在の日時の値を表す日付です。

構文

TODAY(<>)

詳細

TODAY 関数は、現在の日時の値を返します。たとえば、2010 年 2 月 12 日の午後 4 時の時点で、この関数は”02/12/10 4:00:00 PM”という値を返します。文字列として表現されていますが、実際の値は日付値です。詳細については、日付式を参照してください。

この関数を使用する前に、まず、日付/時刻の値を格納する日付変数を宣言する必要があります。

例

```
// declare the date variable to contain the date and time
date currentdate
// Use the TODAY function to populate the date variable with the current date and time
currentdate = TODAY()
```

TODAYGMT 関数

現在の日時を返します。この関数はグリニッジ平均時(GMT)に基づいています。

カテゴリ:	日時
返されるデータの種類:	文字
注:	返される値は、現在の GMT の日時の値を表す日時の組み合わせです。

構文

TODAYGMT(<>)

詳細

TODAYGMT 関数は、現在の GMT の日時の値を返します。たとえば、2010 年 2 月 12 日の午後 4 時の時点で、この関数は”02/12/10 4:00:00 PM”という値を返します。文字列として表現されていますが、実際の値は日付値です。

この関数を使用する前に、まず、日付/時刻値を格納する日付変数を宣言する必要があります。

例

```
// declare the date variable to contain the date and time
date currentdate
// Use the today function to populate the date variable with the current date and time
currentdate = todaygmt()
```

15

ファイル関数

概要

外部ファイルオブジェクトを使用して、式エンジン言語 (EEL)のファイルを操作します。ファイルオブジェクトでは読み取りおよび書き込み操作がサポートされており、ファイルを操作して作業するための関数が追加されています。

ファイルオブジェクトの概要

ファイルオブジェクトを使用して、ファイルを開いたり、読み取ったり、書き込んだりできます。ファイルオブジェクトを使用してファイルを開きます。次に例を示します。

```
File f
f.open("c:\filename.txt","r")
```

この例では、OPEN()関数は filename.txt を開きます。ファイルを開くモードを読み取ります。その他のモードとしては、"a"(ファイルの末尾に追加)と"w"(書き込み)があります。これらのスイッチを組み合わせて使用することができます。

プログラムとファイルコマンドの実行

プログラムを実行するには、EXECUTE()関数を使用します。

```
execute(string)
```

たとえば、次のコードは、Data Job Editor で作成されたテキストファイルのデフォルトのパーミッションを変更します。

Microsoft Windows でコマンドを実行するには、次を入力します。

```
execute("/bin/chmod", "777", "file.txt")
```

または、UNIX シェルから実行するには、次を入力します。

```
execute("/bin/sh", "-c", "chmod 777 file.txt")
```

Execute 関数を使用したバッチファイルの実行

MS-DOS のコマンドプロンプトを呼び出すには、cmd.exe ファイルを呼び出します。次に例を示します。

```
//Expression
execute("cmd.exe", "/q" ,"/c", C:\BatchJobs.bat");
```

次のパラメーターを宣言することができます。

- /q — エコーをオフにします
- /c — MS-DOS プロンプトで指定されたコマンドを実行し、プロンプトを閉じます

注: 式エンジンはバックスラッシュ文字を別の方法で扱います。エスケープする必要はありません。

例: "C:\\Program Files\DataFlux"は、"C:\Program Files\DataFlux"と入力してください

ディクショナリ

CLOSE 関数

開いているファイルを閉じます。

カテゴリ: 外部ファイル

返されるデータの種類: 整数

注: 返される値は、1=成功、0=エラーのブール値です。

構文

fileobject.**CLOSE**

詳細

CLOSE メソッドは、現在開いているファイル(fileobject.OPEN 呼び出しを使って開いたファイル)を閉じます。

例

```
file myfile
if ( myfile.open("data.txt") ) then ...
rc = myfile.close()
```

COPYFILE 関数

ファイルをコピーします。

カテゴリ: 外部ファイル

返されるデータの種類: 整数

注: 返される値は、1=成功、0=エラーのブール値です。

構文

COPYFILE(<source_file, target_file>)

必須引数

source_file
: コピーするファイルの名前を表す文字列。これは固定文字列、フィールド名、または式で指定できます。

target_file
: 書き込まれるファイルの名前を表す文字列。これは固定文字列、フィールド名、または式で指定できます。

詳細

COPYFILE 関数はファイルをコピーします。対象ファイルが存在する場合、COPYFILE 関数は対象ファイルを上書きします。

例

```
string source_file
string target_file
boolean rc_ok

source_file="C:\mydata.txt"
target_file="C:\mydata_copy.txt"

rc_ok = copyfile(source_file, target_file)
```

DELETEFILE 関数

指定のファイルを削除します。

カテゴリ: 外部ファイル

返されるデータの種類: 整数

注: 返される値は、1=成功、0=エラーのブール値です。

構文

DELETEFILE(<filename>)

必須引数

filename
削除するファイルの名前を表す文字列。これは固定文字列、フィールド名、または式で指定できます。

詳細

DELETEFILE 関数は、ディスクからファイルを削除します。ファイルが存在しなかった場合、リターンコードは false に設定されます。

例

```
string filename
boolean rc_ok

filename="C:\mydata_copy.txt"

rc_ok = deletefile(filename)
```

EXECUTE 関数

指定のプログラムを実行します。

カテゴリ:	外部ファイル
返されるデータの種類:	整数
注:	戻り値は、プログラムの既存のステータスを表します。プログラムが見つからないなどのエラーが発生した場合は-1 を返します。

構文

EXECUTE(<filename<option1, option2,..., <option N>>

必須引数

filename
: 実行するファイル(またはコマンド)を表す文字列。これは、固定文字列、フィールド名、または式で指定できます。

オプション引数

option1...N
: 実行しようとしているファイル(またはコマンド)に渡されるオプションを表す(オプション)文字列。これは固定文字列、フィールド名、または式で指定できます。

詳細

EXECUTE 関数は、ファイル(またはオペレーティングシステムコマンド)を呼び出します。

注: 次の例に示すように、スペースが埋め込まれているパラメーターには単一引用符を使用します。

```
execute('cmd.exe','/C',
    '"C:\Progra~1\SASHome\DataFluxDataManagementStudio\2.9\bin\dmpexec
-j C:\ProgramData\DataFlux\DMStudio\studio1\MyRepo\FILEST~1\batch_jobs\sample.djf
-l c:\temp\JobLogs\sample.log.txt"')
```

注: EXECUTE 関数の最初のパラメーターは、実行ファイル、スクリプト、またはバッチファイルへのフルパスでなければなりません。引数を引用したり、含めたりしないでください。後続のパラメーターの引数を EXECUTE 関数に渡します。

```
execute('path\to\executable.exe')
execute('executable.exe','argument')
```

例

```
integer rc

// Windows example
rc = execute("cmd.exe", "/Q", "/C", "C:\mybatchjob.bat")
// /Q turns echo off
// /C executes the command specified by filename and then closes the prompt

// Unix example
rc = execute("/bin/sh", "-c", "chmod 777 file.txt")
```

FILEEXISTS 関数

指定のファイルが存在しているか確認します。

カテゴリ: 外部ファイル

返されるデータの種類: 文字

注: 返される値は、ファイルが存在する場合は true です。

構文

FILEEXISTS(<filename>)

必須引数

filename
ファイルが存在するかどうかを調べるために確認中のファイルの名前。

例

```
string filename
boolean rc_ok

filename="C:\doesexist.txt"

rc_ok = fileexists(filename) // outputs "true" if file exists
```

MKDIR 関数

ディレクトリを作成します。

カテゴリ: 文字列

返されるデータの種類: ブール値

構文

MKDIR(*string*<, *create-intermediary-directories*>)

必須引数

string
作成するディレクトリを含むテキスト文字列を指定します。

オプション引数

create-intermediary-directories
これらの値を使用して、存在しない場合に中間ディレクトリを作成するかどうかを指定します。

TRUE　中間ディレクトリを作成するように指定します。

FALSE　中間ディレクトリを作成しないように指定します。

例

例 1

```
// Declare a string variable to contain the path to the directory to be created
string dir
dir="C:\DataQuality\my_data"

// Declare a Boolean variable for the MKDIR function call
boolean d

// Use the MKDIR function to create the C:\DataQuality\my_data directory
d mkdir(dir)
```

例 2

```
// Declare a string variable to contain the path to the directory to be created
string dir
dir="C:\DataQuality\my_data"

// Declare Boolean variables for the MKDIR function call and the optional condition
boolean b
boolean d

// Set the condition "true" to create an intermediary directory if it does not exist
b=true

// Use the MKDIR function to create the new directory, including the intermediary
// directory DatQuality, if it does not exist.
d mkdir(dir,b)
```

MOVEFILE 関数

指定のファイルの移動または名前の変更をします。

カテゴリ: 外部ファイル

返されるデータの種類: 整数

注: 返される値は、1=成功、0=エラーのブール値です。

構文

MOVEFILE(<old_file, new_file_name>)

必須引数

old_file_name
移動するファイルの名前を表す文字列。これは固定文字列、フィールド名、または式で指定できます。

new_file_name
ファイルが移動される場所の名前(位置を含む)を表す文字列。これは固定文字列、フィールド名、または式で指定できます。

詳細

MOVEFILE 関数はファイルを移動します。この関数がファイルを新しい場所に移動させるためには、ディレクトリ構造がすでに整っている必要があります。対象ファイルが既に存在する場合は、ファイルを移動せず、false を返します。

例

```
string old_file_name
string new_file_name
boolean rc_ok

old_file_name = "C:\mydata_copy.txt"
new_file_name = "C:\TEMP\mydata_copy.txt"

rc_ok = movefile(old_file_name, new_file_name)
```

OPEN 関数

指定のファイルを開きます。

カテゴリ:	外部ファイル
返されるデータの種類:	整数
注:	返される値は、1=成功、0=エラーのブール値です。

構文

fileobject.**OPEN**(<filename, openmode>)

必須引数

filename
: 開くファイルの名前を表す文字列。ファイルが存在しない場合は作成されます。このパラメーターは、固定文字列、フィールド名、または式で指定することができます。

openmode
: 使用する OPENMODE を表す(オプション)文字列。これは、固定文字列、フィールド名、または式で指定できます[a = append、r = read、w = write、rw = read and write]。

詳細

open メソッドは、filename パラメーターで指定されたファイルを開きます。ファイルが存在せず、"a"または"w"を含む OPENMODE が指定された場合、ファイルが作成されます。OPENMODE が指定されていない場合、値 false を返します。

WRITEBYTES メソッドと WRITELINE メソッドがファイルの末尾に書き込む場合、ファイル内の位置を調整するために SEEKBEGIN、SEEKCURRENT、または SEEKEND メソッドを使用してファイルの位置を調整しない限り、情報はファイルの現在位置に書き込まれます。

OPENMODE の"w"を使用した場合、WRITEBYTES メソッドと WRITELINE メソッドは、ファイルの現在位置で書き込みを行い、ファイルの既存の情報を上書きする可能性があります。

例

```
file myfile
```

```
if ( myfile.open("data.txt") ) then ...
```

POSITION 関数

ファイルの現在のカーソル位置を返します。ファイルの開始からのバイト数になります。

カテゴリ:	外部ファイル
返されるデータの種類:	整数
注:	戻り値は、ファイルの先頭からの現在位置(オフセット)を表す整数です。

構文

<fileobject>.**POSITION**(<>)

詳細

position メソッドは、ファイル内のカーソルの現在位置を返します。SEEKEND()メソッドと組み合わせることで、ファイルのサイズを決定する際に使用できます。

例

```
file f
integer byte_size

f.open("C:\filename.txt", "r")
f.seekend(0) // position cursor at end of file

// or if you want to test return codes for the method calls
// boolean rc_ok
// rc_ok = f.open("C:\filename.txt", "r")
// rc_ok = f.seekend(0) // position cursor at end of file

// The integer variable byte_size will have
// the size of the file in bytes
byte_size = f.position()

f.close()
```

READBYTES 関数

ファイルから特定のバイト数を読み取ります。

カテゴリ: 外部ファイル

返されるデータの種類: 整数

注: 戻り値は実際に読み取ったバイト数で、失敗した場合は 0 です。

構文

<fileobject>.**READBYTES**(<number_of_bytes, buffer>)

必須引数

number_of_bytes
: ファイルから読み取る必要があるバイト数を指定する整数。このパラメーターは、数値、フィールド名、または式で指定することができます。

buffer
: 読み取られたバイトを含む文字列。このパラメーターは、固定文字列、フィールド名、または式で指定することができます。

詳細

READBYTES メソッドは、ファイルポインターの現在位置から始まるファイルから、指定されたバイト数を読み取ります。ファイルポインターの位置は、最後に読み取られたバイトの後になります。バッファーが小さすぎる場合は、ファイルの最初のバイトだけがバッファーに入れられます。

このメソッドは通常、バイナリファイルの読み取りに使用されます。各種フォーマット関数を使用して、読み取ったバイナリ情報を変換できます。

このメソッドは EOL 文字も読み取ることに注意してください。こうした Windows のテキストファイルを読み取る場合:

```
C:\filename.txt
```

```
abc
```

```
def
```

READBYTES(7, buffer)ステートメントにより、フィールドバッファーは次の値"abc \n de"を含みます。値は、1 行目の全ての情報(3 バイト)の後に CR 文字と LF 文字(Windows では 2 バイト、UNIX では 1 バイト)が続き、"\n"で表されます。これらの後に、2 行目から最初の 2 バイトが続きます。テキストファイルを読み取りるには、READLINE()メソッドを使用します。

例

```
string input
file f

f.open("C:\filename.txt", "r")
f.readbytes(7, input)

// or if you want to test return codes for the method calls
// boolean rc_ok
// rc_ok = f.open("C:\filename.txt", "r")
// rc_ok = f.readbytes(7, input)
```

READLINE 関数

開いているファイルから次の行を読み込みます。

カテゴリ: 外部ファイル

返されるデータの種類: 文字

注: 返される値は、ファイルから読み込まれた行を含む文字列です。

構文

fileobject.**READLINE**(<>)

詳細

READLINE メソッドは、開いているファイルから次の行のデータを読み込みます。最大 1024 バイトまで読み込まれます。テキストを返します。ファイルの終了などの条件があった場合は Null を返します。

例

```
file f
```

```
string input

f.open("C:\filename.txt", "r")
input=f.readline()
f.close()
```

RMDIR 関数

空のディレクトリの場合削除します。

カテゴリ: 文字列

返されるデータの種類: ブール値

構文

RMDIR(*directory*)

必須引数

directory
空の場合に削除するディレクトリを指定します。

例

```
// Declare a string variable to contain the path to the directory to be created
string dir
dir="C:\DataQuality\my_data"

// Declare a Boolean variable for the MKDIR function call
boolean d

// Use the MKDIR function to create the C:\DataQuality\my_data directory
d rmdir(dir)
```

SEEKBEGIN 関数

ファイルの先頭から始まる位置にファイルポインターを設定します。成功すると true を返し、そうでない場合は false を返します。パラメーターは位置を指定します。

カテゴリ: 外部ファイル

返されるデータの種類: 整数

注: 返される値は、1=成功、0=エラーのブール値です。

構文

fileobject.**SEEKBEGIN**(<position>)

必須引数

position
ファイルの先頭から前進させる必要があるバイト数を指定する整数。0 を指定すると、ファイルの開始を意味します。このパラメーターは、数値、フィールド名、または式で指定することができます。

詳細

SEEKBEGIN メソッドは、ファイルポインターをファイルの指定された位置に移動します。ここで、0 はファイルの開始を示します。成功すると true を返します。それ以外の場合は、false が返されます。0 を指定すると、ファイルの最初の位置の後から読み込みが開始されることを意味します。

例

```
file f
string input

f.open("C:\filename.txt", "r")

input = f.readline()

// return the pointer to the beginning of the file
// and read the first line again
f.seekbegin(0)
f.readline()

f.close()
```

SEEKCURRENT 関数

ファイル内の現在位置を基準にして、ファイル内の位置にファイルポインターを設定します。

カテゴリ: 外部ファイル

返されるデータの種類: 整数

注: 返される値は、1=成功、0=エラーのブール値です。

構文

fileobject.**SEEKCURRENT**(<position>)

必須引数

position
ファイル内の現在位置から移動する必要があるバイト数を指定する整数。正の値は前進するバイト数を指定し、負の値は後退するバイト数を指定します。このパラメーターは、数値、フィールド名、または式で指定することができます。

詳細

SEEKCURRENT メソッドは、ファイルの現在位置からファイルポインターを移動します。このメソッドは、オフセットを含むバイナリファイルを読み込む際に、関連情報がファイル内のどこにあるかを示すのに便利です。

例

```
file f
string input

f.open("C:\filename.txt", "r")

input = f.readline()

// The file contains 3 bytes per record followed by a CR and LF character
// So move the pointer 3+2=5 positions back to read the beginning of
// the first line and read it again.
f.seekcurrent(-5)
f.readline()

f.close()
```

SEEKEND 関数

ファイルの最後からカウントされたファイルの位置にファイルポインターを設定します。

カテゴリ: 外部ファイル

返されるデータの種類: 整数

注: 返される値は、1=成功、0=エラーのブール値です。

構文

fileobject.**SEEKEND**(<position>)

必須引数

position
ファイルの最後から戻る必要があるバイト数を指定する整数。0 を指定すると、ファイルの最後を意味します。このパラメーターは、数値、フィールド名、または式で指定することができます。

詳細

SEEKEND メソッドは、指定されたバイト数だけファイルポインターを戻します。ここで、0 はファイルの最後を示します。成功すると true を返し、そうでない場合は false を返します。

例

```
file f
f.open("C:\filename.txt", "rw")

// write information to the end of the file
f.seekend(0)
f.writeline("This is the end ")
f.close()
```

WRITEBYTES 関数

ファイルに特定のバイト数書き込みます。

カテゴリ: 外部ファイル

返されるデータの種類: 整数

注: 返される値は、書き込まれたバイト数を表す整数です。

構文

fileobject.**WRITEBYTES**(<number_of_bytes, buffer>)

必須引数

number_of_bytes
: ファイルに書き込まれるバイト数を指定する整数。このパラメーターは、数値、フィールド名、または式で指定することができます。

buffer
: 書き込む必要のあるバイトを含む文字列。このパラメーターは、固定文字列、フィールド名、または式で指定することができます。

詳細

WRITEBYTES メソッドは、指定したバイト数をファイルの現在位置から始まるファイルに書き込みます。このメソッドは、ファイル内の現在の位置に存在するデータを上書きします。ファイル内の現在位置に書き込まれるバイト数を加えた値が現在のファイルサイズよりも大きい場合、ファイルサイズが大きくなります。

バッファーが指定された number_of_bytes より大きい場合は、バッファーの最初の number_of_bytes のみが書き込まれます。このメソッドを使用するには、ファイルを Write または Append モードで開く必要があります。このメソッドは、実際に書き込まれたバイト数を返します。

このメソッドは通常、バイナリファイルの書き込みに使用されます。テキストファイルを書き込むには、WRITELINE()メソッドを使用できます。

例

```
string input
file f

string = "this is longer than it needs to be"
f.open("C:\filename.txt", "rw")
// This will write to the beginning of the file
// Only the first 10 bytes from the string will be written
// If the file was smaller than 10 bytes it will be automatically
// appended
f.writebytes(10, input)

f.close()
```

WRITELINE 関数

ファイルに 1 行書き込みます。

カテゴリ:	外部ファイル
返されるデータの種類:	整数

注: 返される値は、1=成功、0=エラーのブール値です。

構文

fileobject.**WRITELINE**(<string>)

必須引数

string
ファイルに書き込む必要のある情報を指定する文字列。このパラメーターは、固定文字列、フィールド名、または式で指定することができます。

詳細

WRITELINE()メソッドは、ファイル内の現在位置に文字列を書き込みます。このメソッドは、ファイル内の現在の位置に存在するデータを上書きします。ファイル内の現在位置に文字列の長さを加えた値が現在のファイルサイズよりも大きい場合、ファイルサイズが大きくなります。

このメソッドを使用するには、ファイルを Write または Append モードで開く必要があります。

例

```
file f

f.open("C:\filename.txt", "a")
f.writeline("This text will be appended to the file")

f.seekbegin(0)
f.writeline("Using seekbegin(0) and Append will
still cause the info to be written at the start of the file")

f.close()
```

16

情報/変換関数

概要

式エンジン言語(EEL)には、次の情報関数と変換関数があります。

ディクショナリ

DETERMINE_TYPE 関数

入力文字列が示すデータ型を返します。

カテゴリ:	日付
返されるデータの種類:	文字列

構文

DETERMINE_TYPE

必須引数

string
　データの文字列です。

returns
　入力文字列が示すデータ型です。

詳細

この関数は、文字列を解析して、それが文字列、整数、ブール値、日付、実数のいずれかであるかを判断します。

例

例 1

　1000
　Results: integer

例 2

　1000.5
　Results: real

ISALPHA 関数

式が英文字のみ含む文字列の場合は true 値を返します。

カテゴリ:	情報と変換
返されるデータの種類:	整数
注:	返される値はブール値で、"in_string"にアルファベット文字のみが含まれていれば true です。それ以外の場合、値は false です。

構文

ISALPHA(<in_string>)

必須引数

in string
　任意のアルファベット文字を検索する文字列。

詳細

ISALPHA 関数は、"in_string"がアルファベット文字のみを含む文字列であると判定された場合、true を返します。

例

例 1

```
// Expression
string letters
letters="lmnop"
string mixed
mixed="1a2b3c"

string alphatype
alphatype=isalpha(letters) // returns true
string mixedtype
mixedtype=isalpha(mixed) // returns false
```

例 2

```
string all_Alpha
all_Alpha="abcdefghijklmnoprstuvyz"

string non_Alpha
non_Alpha="%&)#@*0123456789"

string error_message1
string error_message2

if (NOT isalpha(all_Alpha))
  error_message1 ="all_Alpha string contains alpha numeric characters"
else
  error_message1 ="all_Alpha string contains alpha numeric characters"
```

```
if(isalpha(non_Alpha))
  error_message2= "non_Alpha string contains alpha numeric characters"
else
  error_message2= "non_Alpha string does not contain alpha numeric characters"
```

例 3

```
string all_Alpha
string error_message
all_Alpha="abcdefghijklmnopqrstuvwxyz"
if (isalpha(all_Alpha))
  begin
     error_message= "alpha strings were identified as alpha"
  end
```

ISBLANK 関数

引数に空白、空の値が含まれているかどうかを確認します。引数値が空白、NULL、または制御文字(chr(09)、chr(10)、chr(11)、chr(12)、chr(13)を含む)の場合、この関数は true を返します。それ以外の場合は false を返します。

カテゴリ:	情報と変換
返されるデータの種類:	整数
注:	返される値はブール値です。

構文

<boolean>**ISBLANK**(<argvalue>)

必須引数

argvalue
　文字列です。

詳細

ISBLANK 関数の引数型は文字列になります。

例

例 1

```
string x
string y
date z
string error_message1
string error_message2

x="Hello"

if(isblank(x) )
  error_message1 = "x is blank"
else
  error_message1= "x is not blank"

if( isblank(y) )
  error_message2 =" String y value is blank"
else
  error_message2 =" String y value is not blank"
```

例 2

```
string toCheck
boolean result
toCheck = " "

result = isblank (toCheck)
//or
result = isblank(" ")
//both return True
```

ISNULL 関数

引数値に NULL 値が含まれているかどうかを確認します。引数の値が NULL の場合、関数は true を返します。それ以外の場合は false を返します。

カテゴリ:	情報と変換
返されるデータの種類:	整数
注:	返される値はブール値です。

構文

<boolean>**ISNULL**(<argvalue>)

必須引数

argvalue
　文字列、日付、整数、実数、ブール値。

詳細

ISNULL 関数は、引数型として文字列、日付整数、実数、ブール値を使用します。

例

例 1

```
// Expression
if State <> "NC" OR isnull(State)
  return true
else
  return false
```

例 2

```
integer x
string y
string error_message1
string error_message2

y="Hello"

if(isnull(x) )
  error_message1 = "Integer x is null"
else
  error_message1= "Integer x is not null"

if( isnull(y) )
  error_message2 =" String y value is null"
else
  error_message2 =" String y value is not null"
```

ISNUMBER 関数

引数の値に数値が含まれているかどうかを確認します。引数の値が数字の場合、関数は true を返します。それ以外の場合は false を返します。

カテゴリ: 情報と変換

返されるデータの種類: 整数

注: 返される値はブール値です。

構文

argvaluebooleanISNUMBER()

必須引数

argvalue
文字列

詳細

ISNUMBER 関数の引数型は文字列になります。

例

```
string x
string y
string z
string error_message1
string error_message2
string error_message3

x ="5"
y="Hello"
z="01/01/10"

if(isnumber(x) )
  error_message1 = "String x is a number"
else
  error_message1= "String x is not a number"

if( isnumber(y) )
  error_message2 =" String y value is a number"
```

```
else
  error_message2 =" String y value is not a number"

if( isnumber(z) )
  error_message3 = "String z value is a number"
else
  error_message3 = "String z value is not a number"
```

LOCALE 関数

ロケールを設定します。

カテゴリ: 情報と変換

返されるデータの種類: 文字列

注: 返される値は、現在のロケール設定の文字列です。
この関数は、変換や日付操作など一部の操作に影響を与えます。この関数は、前回のロケールを返します。パラメーターを渡さない場合は、現在のロケールを返します。ロケール設定はグローバル設定です。

構文

<string>**LOCALE**(<>)

<string>**LOCALE**(<“locale_string”>)

詳細

パラメーターが指定されている場合は、そのパラメーターが設定されます。それ以外の場合、パラメーターを取得します。設定すると、古いロケールを取得します。

LOCALE()関数では、次の値を設定することができます。

- アメリカと英国では 2 文字の略語を使用できます。
- 国によっては、GER、FRA、DEU のように 3 文字の略語を使用できます。

ここでは、例をいくつか紹介します。

```
my_locale = locale("DEU")
my_locale = locale("German")
my_locale = locale("German_Germany")
my_locale = locale("German_Germany.1252")
```

LOCALE()関数は、現在の設定を Country_Language.codepage 表記で返します。

注: ("#iso8601")を使用している場合、日付は ISO8601 形式で設定されます。

例

```
string currentSetting
string newSetting

currentSetting = locale();

newSetting = locale("FRA");
```

TOBOOLEAN 関数

引数をブール値に変換します。

カテゴリ: 情報と変換

返されるデータの種類: 整数

注: 返される値は、引数値をブール値に変換できる場合に返されるブール値です。

構文

boolean**TOBOOLEAN**(value<>)

必須引数

value
実数、整数、文字列、日付のいずれかで渡されます。

例

```
boolean convertedValue
integer result
result = 1
convertedValue = toboolean(result)
Print (convertedValue)
```

TYPEOF 関数

渡された値のデータタイプを識別します。

カテゴリ: 情報と変換

返されるデータの種類: 文字

注: 返り値は、ブール変数はブール値を返す、整数変数は整数を返す、実変数は実数を返す、文字列変数は文字列を返す、のいずれかです。

構文

<string> **TYPEOF**(in_value)

オプション引数

in value
評価される変数です。

例

例 1

```
// Expression
string hello
hello="hello"

boolean error
error=false

// variable that will contain the type
string type
type=typeof(hello)

// type should be string
if(type<>"string") then
  error=true
```

例 2

```
string content
content = "Today is sunny"

hidden integer one
one =1

hidden real pi
pi=3.1415962

hidden boolean test
test=false
```

```
hidden string type

type= typeof(content);

if (type == "string")
  begin
     error_message="The data type for variable 'Content' is string"
  end

type=typeof(one)
if (type == "integer")
  begin
     error_message="The data type for variable 'one' is integer"
  end
type= typeof(pi);

if (type == "real")
  begin
     error_message="The data type for variable 'real' was real"
  end

type= typeof(test);

if (type == "boolean")
  begin
     error_message="The data type for variable 'test' was boolean"
  end
```

17

数学関数

概要

式エンジン言語(EEL)では、次のような数学関数が利用できます。

ディクショナリ

ABS 関数

数値の絶対値を返します。

カテゴリ: 数学

返されるデータの種類: 実数

注: ABS 関数は、引数の大きさと等しい大きさの非負の数を返します。

構文

ABS(*argument*)

必須引数

argument
実データ型を持つ値を指定します。これは、数値定数、フィールド名、または式で指定できます。

例

例 1

ステートメント	結果
x = abs(3.5)	// outputs 3.5
x = abs(-7)	// outputs 7
x = abs(-3*1.5)	// outputs 4.5

例 2

```
real seconds_diff
seconds_diff = abs((date1 - date2) * 86400)
// The number 86400 represents the total number of seconds in a day
```

CEIL 関数

引数以上の最小の整数を返します。

カテゴリ: 数学

返されるデータの種類: 実数

構文

CEIL(*argument*)

必須引数

argument
実データ型を持つ値を指定します。これは、数値定数、フィールド名、または式で指定できます。

詳細

これは切り上げ(天井)とも呼ばれています。

例

```
x = ceil(3.5)
// outputs 4

x = ceil(-3.5)
// outputs -3

x = ceil(-3)
// outputs -3

x = ceil(-3*1.5)
// outputs -4
```

FLOOR 関数

引数以下の最大の整数を返します。

カテゴリ:	数学
返されるデータの種類:	実数

構文

FLOOR(*argument*)

必須引数

argument
データ型が実数の値を指定します。これは、数値定数、フィールド名、または式で指定できます。

詳細

これは切り捨てとも呼ばれています。

例

```
x = floor(3.5)
// outputs 3

x = floor(-3.5)
// outputs -4

x = floor(-3)
// outputs -3

x = floor(-3*1.5)
// outputs -5
```

MAX 関数

一連の値の最大値を返します。

カテゴリ:	数学
返されるデータの種類:	実数

構文

MAX(*argument1*< , *argument2*, ...>)

必須引数

argument1
実データ型を持つ値を指定します。これは、数値定数、フィールド名、または式で指定できます。

オプション引数

argument2, ...
実データ型を持つ 1 つ以上の値を指定します。これらの値は、数値定数、フィールド名、または式で指定できます。

詳細

この関数は、すべての値が NULL の場合は NULL を返します。

例

```
x = max(1, 3, -2)
 // outputs 3

x = max(1, null, 3)
 // outputs 3

x = max(-3)
// outputs -3

x = max(4, -3*1.5)
 // outputs 4
```

MIN 関数

一連の値の最小値を返します。

カテゴリ:	数学
返されるデータの種類:	実数

構文

MIN(*argument1*<, *argument2*, ...>)

必須引数

argument1
データ型が実数の値を指定します。これは、数値定数、フィールド名、または式で指定できます。

オプション引数

argument2, ...
実データ型を持つ 1 つ以上の値を指定します。これらの値は、数値定数、フィールド名、または式で指定できます。

詳細

この関数は、すべての値が NULL の場合は NULL を返します。

例

```
x = min(1, 3, -2) // outputs -2
x = min(1, null, 3) // outputs 1
x = min(-3) // outputs -3
x = min(4, -3*1.5) // outputs -4.5
```

POW 関数

数値の指定された累乗を計算します。

カテゴリ:	数学
返されるデータの種類:	実数

構文

POW(*x*, *y*)

必須引数

x
データ型が実数の値を指定し、乗する基数を示します。これは、数値定数、フィールド名、または式で指定できます。

y
データ型が実数の値を指定し、乗する累乗を示します。これは、数値定数、フィールド名、または式で指定できます。

詳細

POW 関数は *x* をを累乗 *y*、x^y に乗します。

例

ステートメント	結果
x = pow(5,2)	// outputs 25
x = pow(5,-2)	// outputs 0.04
x = pow(16,0.5)	// outputs 4

ROUND 関数

数値を指定された小数点以下の桁数で最近傍の数値に丸めます。

カテゴリ: 数学

返されるデータの種類: 実数

構文

ROUND(<*argument*><*decimals*>)

必須引数

argument
数値定数、フィールド名、または式として指定できるデータ型が実数の値。

オプション引数

decimals
数値定数、フィールド名、または式を指定し、丸め処理の結果提供する小数点以下の桁数を示します。小数点の右へ丸めには小数点以下の正の値を使用します。小数点以下の左には負の値を使用します。

デフォルト 0

例

```
x = round(1.2345,1)
 // outputs 1.2

x = round(1.449,2)
 // outputs 1.45

x = round(9.8765,1)
 // outputs 9.9

x = round(9.8765)
 // outputs 10
```

18

正規表現関数

概要

正規表現(regex)オブジェクトを使用すると、式エンジン言語(EEL)で文字列の正規表現検索を行うことができます。

ディクショナリ

COMPILE 関数

指定されたエンコーディングを使用して有効な正規表現をコンパイルします。

カテゴリ: 正規表現

返されるデータの種類: 整数

構文

r.**COMPILE**(*regex* <,*encoding*>)

必須引数

regex
PCRE 互換の正規表現を指定します。

オプション引数

encoding
エンコーディングで示されるエンコーディング定数を定義する文字列を指定します。エンコーディング列の値を使用します。

デフォルト オペレーティングシステムのデフォルトのエンコーディング。

詳細

コンパイル関数は正規表現オブジェクトとともに使用します。COMPILE( )を使用して PCRE 互換の正規表現をコンパイルするには、まず正規表現オブジェクトを変数として定義する必要があります。正規表現は、高度なパターンマッチング(場合によってはパターン置換)を行うために使用できます。下記の他の関数を使用して、指定された文字列(変数の場合もあります)でパターンを見つけたり、一致するパターンの長さを決定したり、パターンを置換したりできます。

正規表現のコンパイルに成功した場合、この関数は 1 を返します。それ以外の場合は 0 を返します。失敗の原因は、正規表現の書式が正しくないか、あるいは無効なエンコーディング定数である可能性があります。

パフォーマンス上の理由から、正規表現は**式**ノードの前処理ステップでコンパイルするのが最適です。つまり、データ行が**式**ノードによって処理される前に、正規表現は一度だけコンパイルされます。

注: このセクションのサンプルコードでは、通常、わかりやすくするために、COMPILE()関数を**式**タブに残りの式コードと一緒に配置しています。

場合によっては、すべての行が評価される前に正規表現をコンパイルしなければならない場合があります。たとえば、変数を使ってコンパイルしたい正規表現を定義できます。この変数はデータ行自体から発生している可能性があり、パターン検索が正しく動作するように各行の正規表現を再コンパイルする必要があります。

PCRE 互換の正規表現の構文は、PCREPattern の Man Page(https://www.pcre.org/original/doc/html/pcrepattern.html)に記載されています。PCRE 互換の正規表現では、区切り文字として"/"文字を使用しません。内部オプションは式の"/"で囲まれ

ます。たとえば、Perl 式をコンパイルすると、"/index/i" は成功です。この式は文字列/index/i"のみに一致します。なぜなら、/文字には特別な意味がないためです。Perl 式"/index/i"は、PCRE 互換形式では"(?i)index"と表現されます。PCRE 内部オプションの詳細については、https://www.pcre.org/original/doc/html/pcrepattern.html#SEC13 を参照してください。

検索するパターンのみに一致する正規表現を設計するように注意してください。正規表現コードの書き方が悪いと、多くの追加処理が必要となり、パフォーマンスに悪影響を及ぼす可能性があります。

例

注: この例は、**親が指定されていないときに行を生成する**オプションが選択されている場合、スタンドアロンの式ノードで実行できます。このノードにデータを渡す場合は、この設定をオフにして SETEOF()関数を削除します。特に指定がない限り、表示されているすべてのコードは、**式**ノードの**式**タブに入力する必要があります。

```
//You must define a regex object
regex r
//Then compile your regular expression
//This example will match any single digit in an input string
r.compile ("[0-9]","ISO-8859-1")
// Terminate the expression node processing
seteof()
```

関連項目

- “FINDFIRST 関数” (417 ページ)
- “FINDNEXT 関数” (419 ページ)
- “MATCHLENGTH 関数” (420 ページ)
- “MATCHSTART 関数” (422 ページ)
- “REPLACE 関数” (423 ページ)
- “SUBSTRINGCOUNT 関数” (425 ページ)
- “SUBSTRINGLENGTH 関数” (427 ページ)
- “SUBSTRINGSTART 関数” (429 ページ)

FINDFIRST 関数

コンパイルされた正規表現を使用して、パターンマッチを得るために指定された文字列を検索します。

カテゴリ: 正規表現

返されるデータの種類: ブール値

構文

r.**FINDFIRST**(*input*)

必須引数

input
コンパイルした正規表現で定義されたパターンを検索する文字列の値を指定します。これは明示的な文字列("MyValue")にすることができます。これは、式コードで既に定義されている変数、または前回のノード(MyValue または"My Value")の列として式ノードに渡されている変数にできます。

要件 *input* は NULL または空白であってはなりません。

詳細

FINDFIRST 関数は、入力に 1 つ以上のパターンマッチが見つかったかどうかを示します。この関数を使用すると、入力文字列から一致した一連のパターンを取り出す論理ループに入ることができます。この関数は、パターンマッチが見つかった場合に TRUE を返します。FALSE 戻り値は、一致するものが見つからず、処理を続行できることを示します。

例

注: この例は、**親が指定されていないときに行を生成する**オプションが選択されている場合、スタンドアロンの式ノードで実行できます。このノードにデータを渡す場合は、この設定をオフにして SETEOF()関数を削除します。特に指定がない限り、表示されているすべてのコードは、**式**ノードの**式**タブに入力する必要があります。

```
//You must define a regex object
regex r
//Then compile your regular expression. This one will match any single
// uppercase letter
r.compile("[A-Z]")
// If a pattern match is found this will evaluate to 1 (TRUE)
if r.findfirst("Abc")
// Print the output to the statistics file. You must run
// this job for stats to be written. A preview will not generate
// a message in the log.
print("Found match starting at " & r.matchstart() & " length " & r.matchlength())
// Terminate the expression node processing
seteof()
```

FINDNEXT 関数

FINDNEXT()関数の使用後、次に一致するものがないか文字列の検索を続行します。

カテゴリ: 正規表現

返されるデータの種類: ブール値

構文

r.**FINDNEXT**(*input*)

必須引数

input
コンパイルした正規表現で定義されたパターンを検索する文字列の値を指定します。これは明示的な文字列("MyValue")にすることができます。これは、式コードで既に定義されている変数であってもよいですし、前のノード(MyValue または"My Value")の列として式ノードに渡すこともできます。

要件 *input* は NULL または空白であってはなりません。

詳細

FINDNEXT 関数は、FINDFIRST()が使用された後に別のパターンマッチが見つかったことを示します。"While"ステートメントループを使用すると、戻り値が true である限り、この関数を使用して潜在的なすべてのパターンマッチを 繰り返し処理できます。

この関数は、パターンマッチが見つかった場合に TRUE を返します。それ以外の場合は FALSE を返します。

例

注: この例は、**親が指定されていないときに行を生成する**オプションが選択されている場合、スタンドアロンの式ノードで実行できます。このノードにデータを渡す場合は、この設定をオフにして SETEOF()関数を削除します。データ値をデータ行としてノードに渡す場合は、PUSHROW ステートメントも不要です。特に指定がない限り、表示されているすべてのコードは、**式**ノードの**式**タブに入力する必要があります。

```
// Define some string variables
string MyString
string MySubString

// Set one to some sample input
MyString = "DwAwTxAxFyLyUzXz"

//You must define a regex object
regex r
// Then compile your regular expression
// This one will match any single uppercase letter
r.compile("[A-Z]")
// Find the first pattern match
if r.findfirst(MyString)
  begin
  // Pull the pattern from MyString and place it into MySubString
  MySubString = mid(MyString, r.matchstart(),r.matchlength())
  // Use pushrow to create new rows - this is purely for the sake of
  // clarity in the example
  pushrow()
  // Create a while loop that continues to look for matches
  while r.findnext(MyString)
    begin
    // Pull the pattern from MyString and place it into MySubString agin
    MySubString = mid( MyString, r.matchstart(),r.matchlength())
    // Just for display again
    pushrow()
    end
  end
// Terminate the expression node processing
seteof(true)
// Prevent the last pushrow() from showing up twice
return false
```

MATCHLENGTH 関数

最後に見つかったパターンマッチの長さを返します。

カテゴリ:	正規表現
操作:	この関数は、FINDFIRST( )または FINDNEXT( )を使用して見つかったパターンマッチの部分文字列に対して動作します。
返されるデータの種類:	整数

構文

r.**MATCHLENGTH**( )

引数なし

この関数には引数があります。

詳細

MATCHLENGTH 関数を使用して、FINDFIRST( )または FINDNEXT( )で見つかった現在一致しているパターンの文字の長さを決定します。MATCHSTART( )と一緒に使用すると、この関数を使用して一致する部分文字列を見つけたり、変数を式コードに入力できます。

この関数は、正規表現のパターンマッチとなる文字数を表す正の整数値を返します。現在検討中の部分文字列がなく、したがって返す長さがない場合は NULL が返されます。

例

注: この例は、**親が指定されていないときに行を生成する**オプションが選択されている場合、スタンドアロンの式ノードで実行できます。かわりにこのノードにデータを渡す場合は、この設定をオフにして SETEOF()関数を削除します。特に指定がない限り、表示されているすべてのコードは、**式**ノードの**式**タブに入力する必要があります。

```
// Define some variables
integer i
string MyString

//Supply some values for the variables
i = 0
MyString = "DataFlux"
// Uncomment the line below to see the value of variable i change
//MyString = "Data_Management_Studio"

//You must define a regex object
regex r
//Then compile your regular expression.
// This expression will match as many "word" characters as it can
// (alphanumerics and undescore)
r.compile("\w*")
// If a pattern match is found then set i to show the length of
// the captured substring
if r.findfirst(MyString) then i = r.matchlength()
// Terminate the expression node processing
seteof()
```

MATCHSTART 関数

最後に見つかったパターンマッチの場所を返します。

カテゴリ: 正規表現

返されるデータの種類: 整数

構文

r.**MATCHSTART**(*input*)

必須引数

input
コンパイルした正規表現で定義されたパターンを検索する文字列の値を指定します。

要件 *input* は NULL または空白であってはなりません。

詳細

MATCHSTART 関数は、正規表現に一致した部分文字列の開始文字位置を返します。現在検討中の部分文字列がなく、したがって返す長さがない場合は NULL が返されます。論理ループを使用して、一致するすべての部分文字列を繰り返し処理できます。MATCHLENGTH( )関数を MATCHSTART( )と組み合わせて使用することで、他の値や他の変数に保存された値と比較できるように、一致する部分文字列を取り出すことができます。

例

注: この例は、**親が指定されていないときに行を生成する**オプションが選択されている場合、スタンドアロンの式ノードで実行できます。かわりにこのノードにデータを渡す場合は、この設定をオフにして SETEOF()関数を削除します。データ値をデータ行としてノードに渡す場合は、PUSHROW ステートメントも不要です。特に指定がない限り、表示されているすべてのコードは、**式**ノードの**式**タブに入力する必要があります。

```
// Define some string variables
string MyString
string MySubString
```

```
integer StartLocation

// Set one to some sample input
MyString = "00AA111BBB2222CCCC"
// Will hold the starting location of matched patterns
StartLocation = 0

//You must define a regex object
regex r
// Then compile your regular expression
// This one will match any single uppercase letter
r.compile("[A-Z]+")
// Find the first pattern match
if r.findfirst(MyString)
  begin
  // Pull the pattern from MyString and place it into MySubString
  MySubString = mid(MyString, r.matchstart(),r.matchlength())
  // Use pushrow to create new rows - this is purely for the sake of
  // clarity in the example
  pushrow()
  // Create a while loop that continues to look for matches
  while r.findnext(MyString)
    begin
    // Pull the pattern from MyString and place it into MySubString
    again
    MySubString = mid( MyString, r.matchstart(),r.matchlength())
    // Set StartLocation to the starting point of each pattern found
    StartLocation = r.matchstart()
    // Just for display again
    pushrow()
    end
  end
// Terminate the expression node processing
seteof(true)
// Prevent the last pushrow() from showing up twice
return false
```

REPLACE 関数

最初の文字列を検索し、それを 2 番目の文字列に置き換えます。これは、regex オブジェクトの外で使用される REPLACE()関数とは異なります。

カテゴリ:	正規表現
返されるデータの種類:	文字列

構文

r.**REPLACE**(*input-string*, *replacement–value*)

必須引数

input-string
コンパイルした正規表現で定義されたパターンで検索して置換する文字列の値を指定します。これは明示的な文字列("MyValue")にすることができます。これは、式コードで既に定義されている変数であってもよいですし、前のノード(MyValue または"My Value")の列として式ノードに渡すこともできます。

要件 *input-string* は NULL または空白であってはなりません。

replacement–value
コンパイルされた正規表現により一致した *input-string* を置き換える文字列を指定します。

詳細

REPLACE 関数は、単に正規表現に一致するパターンを見つけるだけのものから、一致している部分文字列を新しい値に置き換えるものまで、regex オブジェクトの機能を拡張します。たとえば、(-A-, -B-)の間にある任意の文字を持つ 2 つのハイフンのパターンに一致するすべての部分文字列を一致させ、1 つの文字(Z)と置き換えたい場合、ハイフン/文字/ハイフンのパターンを見つけるための正規表現をコンパイルすることになります。次に、REPLACE()関数の置換値として"Z"を使用して、変数や文字列の値を入力に渡すことになります。

戻り値は、正規表現が動作し、値が入力用に与えられているという条件で実際に置換が行われた場合に、行われた置換を持つ文字列の値となります。置換ができなかった場合は、入力の元の値が返されます。

この機能には制限があります。一致した部分文字列を、その部分文字列の"捕捉された"部分に簡単に置き換えることはできません。先ほどの例では、FINDFIRST( )や FINDNEXT( )を使用して一致した部分文字列が見つかった後に解析し、その操作に基づいて置換値を作成する必要があります。しかし、一致したパターンは可変長の場合があるため、部分文字列の部分の位置を推測するのは厄介です。

これを、標準化定義の一部として正規表現を使用して提供される類似の関数と比較してください。正規表現を使用する QKB 定義の場合、正規表現エンジンでは捕捉された部分文字列を置換値で使用できるため、よりスマートな置換が可能になります。

例

注: この例は、**親が指定されていないときに行を生成する**オプションが選択されている場合、スタンドアロンの式ノードで実行できます。このノードにデータを渡す場合は、この設定をオフにして SETEOF()関数を削除します。特に指定がない限り、表示されているすべてのコードは、**式**ノードの**式**タブに入力する必要があります。

```
//Define two string variables
string MyString
string MyNewString
```

```
// Provide a value for MyString
MyString = "12Flux"

// Defined a regular expression object variable
regex r
// Compile a regular expression that will look for a series of digits
// either 2 or 3 digits long
r. compile("\d{2,3}")
// Use the replace function to place "Data" in place of the found
// pattern and save that in a new string variable.
// If you change MyString to 1234 or 12345 you can see the
// difference in how the pattern is found
MyNewString = r.replace(MyString,"Data")
// Terminate the expression node processing
seteof()
```

SUBSTRINGCOUNT 関数

コンパイルされた正規表現で指定されたパターンに一致することが見つかったサブパターンの数を返します。

カテゴリ:	正規表現
要件	正規表現には、パターンを一致させるために使用できるサブパターンが含まれていなければなりません。
返されるデータの種類:	整数

構文

r.**SUBSTRINGCOUNT**( )

引数なし

この関数には引数はありません。

詳細

SUBSTRINGCOUNT 関数を使用して、正規表現に一致したサブパターンの総数を見つけます。通常、単純な正規表現は"1"に評価されますが、サブパターンを使って正規表現を設計した場合、この関数は見つかった数を返します。

この関数は、正規表現に一致したと認められる部分文字列の数を指定する正の整数を返します。部分文字列が見つからない場合は"0"を返します。

サブパターンを使用するための構文は、開括弧と閉括弧です。次に例を示します。

```
(Mr|Mrs) Smith
```

この例では、サブパターンは"(Mr|Mrs)"となっています。この関数を使用すると、文字列全体が最初のサブパターンと見なされるため、部分文字列のカウントに"2"を返します。括弧内の部分が第 2 のサブパターンです。

この関数は、FOR コマンドを使用して論理ループの上位番号を提供することができるため、コードは一致したサブパターンを繰り返して他の値と比較できます。

例

注: この例は、**親が指定されていないときに行を生成する**オプションが選択されている場合、スタンドアロンの式ノードで実行できます。このノードにデータを渡す場合は、この設定をオフにして SETEOF()関数を削除します。データ値をデータ行としてノードに渡す場合は、PUSHROW ステートメントも不要です。特に指定がない限り、表示されているすべてのコードは、**式**ノードの**式**タブに入力する必要があります。

```
//Define some variables
string MyString
string MyString2
integer i
integer SSC
integer SSS
integer SSL

// Set initial values for variables
i = 0
SSS = 0
SSL = 0
SSC = 0

// Sample inpit string
MyString = "DataFlux Data Management Studio"

// Define a regular expression object
regex r
// Then compile it - notice the use of ( and )
r. compile("(DataFlux|DF) Data Management (Studio|Platform)")
// Find the first substring
if r.findfirst(MyString)
  begin
    // Use the "substring" functions to find the number of substrings
    SSC = r.substringcount()
    // Loop through substrings
    for i = 1 to SSC
    begin
    // Then pull out substrings
    SSS = r.substringstart(i)
    SSL = r.substringlength(i)
    MyString2 = mid(MyString,SSS,SSL)
    // Place the substrings in a data row
    pushrow()
    end
```

```
  end
// Terminate the expression node processing
seteof(true)
// Prevent the last pushrow() from showing up twice
return false
```

SUBSTRINGLENGTH 関数

n 個のキャプチャされたサブパターンの長さを返します。

カテゴリ:	正規表現
要件	正規表現には、パターンを一致させるために使用できるサブパターンが含まれていなければなりません。
返されるデータの種類:	整数

構文

r.**SUBSTRINGLENGTH**(*n*)

必須引数

n
返す長さの部分文字列を示す正の整数を指定します。

要件 *n* は NULL または空白であってはなりません。

詳細

SUBSTRINGLENGTH 関数は、正規表現のサブパターンマッチとなる文字数を表す正の整数値を返します。現在検討中の部分文字列がなく、したがって返す長さがない場合は NULL が返されます。サブパターンの操作についての詳細は、“SUBSTRINGCOUNT 関数” (425 ページ)の「詳細」セクションを参照してください。

単純な正規表現のほとんどはサブパターンを持たないため、この関数は MATCHLENGTH( )と同様の動作をします。しかし、正規表現がサブパターンを使用している場合、この関数を使用して、全体的に一致したパターン内で見つかった個々に捕捉されたサブパターンの長さを見つけることができます。

例

注: この例は、**親が指定されていないときに行を生成する**オプションが選択されている場合、スタンドアロンの式ノードで実行できます。かわりにこのノードにデータを渡す場合は、この設定をオフにして SETEOF()関数を削除します。データ値をデータ行としてノードに渡す場合は、PUSHROW ステートメントも不要です。特に指定がない限り、表示されているすべてのコードは、**式**ノードの**式**タブに入力する必要があります。

```
//Define some variables
string MyString
string MyString2
integer i
integer SSC
integer SSS
integer SSL

// Set initial values for variables
i = 0
SSS = 0
SSL = 0
SSC = 0

// Sample inpit string
MyString = "DataFlux Data Management Studio"

// Define a regular expression object
regex r
// Then compile it - notice the use of ( and )
r. compile("(DataFlux|DF) Data Management (Studio|Platform)")
// Find the first substring
if r.findfirst(MyString)
  begin
    // Use the "substring" functions to find the number of substrings
    SSC = r.substringcount()
    // Loop through substrings
    for i = 1 to SSC
    begin
    // Then pull out substrings
    SSS = r.substringstart(i)
    SSL = r.substringlength(i)
    MyString2 = mid(MyString,SSS,SSL)
    // Place the substrings in a data row
    pushrow()
    end
  end
// Terminate the expression node processing
seteof(true)
// Prevent the last pushrow() from showing up twice
return false
```

SUBSTRINGSTART 関数

n 個のキャプチャされたサブパターンの開始位置を返します。

カテゴリ:	正規表現
要件	正規表現には、パターンを一致させるために使用できるサブパターンが含まれていなければなりません。
返されるデータの種類:	整数

構文

r.**SUBSTRINGSTART**(*n*)

必須引数

n
返す開始位置のサブパターンを示す正の整数を指定します。

要件 *n* は NULL または空白であってはなりません。

詳細

SUBSTRINGSTART 関数は、提供する入力整数 *n* を使用して、その入力整数で表されるサブパターンの開始位置を返します。現在検討中の部分文字列がなく、したがって返す位置がない場合は NULL が返されます。

SUBSTRINGCOUNT( )を使用して、検討中のサブパターンの数を決定します。この関数で SUBSTRINGLENGTH( )を使用して、一致したサブパターンを取り出し、式コードの評価ロジックで使用します。

単純な正規表現のほとんどはサブパターンを持たないため、この関数は MATCHSTART()と同様の動作をします。しかし、正規表現がサブパターンを使用している場合、この関数を使用して、全体的に一致したパターン内で見つかった個々に捕捉されたサブパターンの始点を見つけることができます。

例

注: この例は、**親が指定されていないときに行を生成する**オプションが選択されている場合、スタンドアロンの式ノードで実行できます。かわりにこのノードにデータを渡す場合は、この設定をオフにして SETEOF()関数を削除します。データ値をデータ行としてノードに渡す場合は、PUSHROW ステートメントも不要です。特に指定

がない限り、表示されているすべてのコードは、**式**ノードの**式**タブに入力する必要があります。

```
//Define some variables
string MyString
string MyString2
integer i
integer SSC
integer SSS
integer SSL

// Set initial values for variables
i = 0
SSS = 0
SSL = 0
SSC = 0

// Sample inpit string
MyString = "DataFlux Data Management Studio"

// Define a regular expression object
regex r
// Then compile it - notice the use of ( and )
r. compile("(DataFlux|DF) Data Management (Studio|Platform)")
// Find the first substring
if r.findfirst(MyString)
  begin
    // Use the "substring" functions to find the number of substrings
    SSC = r.substringcount()
    // Loop through substrings
    for i = 1 to SSC
    begin
    // Then pull out substrings
    SSS = r.substringstart(i)
    SSL = r.substringlength(i)
    MyString2 = mid(MyString,SSS,SSL)
    // Place the substrings in a data row
    pushrow()
    end
  end
// Terminate the expression node processing
seteof(true)
// Prevent the last pushrow() from showing up twice
return false
```

19

検索関数

概要

表現エンジン言語(EEL)については、次の検索関数があります。

ディクショナリ

INLIST 関数

ターゲットパラメーターが値パラメーターのいずれかに一致する場合、TRUE を返します。

カテゴリ: 検索

返されるデータの種類: ブール値

構文

INLIST(*target_parameter*, *value_parameter1* <, *value_parameter2*, ...>)

必須引数

target_parameter
検索する文字列、整数、日付を指定します。

value_parameter1
target_parameter で検索する文字列、整数、または日付値を指定します。

オプション引数

value_parameter2, ...
target_parameter で検索する 1 つ以上の文字列、整数、または日付値を指定します。

詳細

target_parameter は各 *value_parameter* と比較されます。一致するものが見つかった場合は TRUE を返します。それ以外の場合は、FALSE が返されます。

例

```
string error_message

integer a
a=5
integer b
b=5

if (inlist(a,3,5)<>true)
  error_message="integer 5 not found in argument list of 3,5 "
else
  error_message="integer 5 was found in argument list of 3,5 "

print(error_message,false)
```

20

文字列関数

概要

式エンジン言語 (EEL) には、組み込みの文字列データ型に影響を与えるいくつかの関数があります。

ディクショナリ

APARSE 関数

文字列をワードに解析し、見つかったワードの数を返し、ワードを配列に配置します。

カテゴリ: 文字列

返されるデータの種類: 整数

構文

APARSE(*string*, *delimiter*, *word_list*)

必須引数

string
単語に区切る必要のある文字列を指定します。これは固定文字列、フィールド名、または式で指定できます。

制限事項 *string* は NULL であってはなりません。実行時エラーが発生します。

注 *string* が空("")の場合は 1 の値が返され、*word_list* には空の文字列を含むエレメントが 1 つあります。

delimiter
文字列を単語に区切る際に区切り文字として使用する文字を含む文字列を指定します。これは固定文字列、フィールド名、または式で指定できます。

制限事項 複数の文字を指定した場合は、最後の文字のみ使用されます。

word_list
解析中に見つかった単語を含む文字列配列を指定します。これはフィールド名で指定できます。

比較

parse 関数も同じです。文字列配列の代わりに個々の文字列フィールドを返します。文字列フィールドは、関数呼び出しの一部として指定する必要があります。APARSE 関数にはこの制限がないため、最大語数があらかじめわからない場合に使用できます。

例

```
string = "one:two:three"
delimiter = ":"
nwords = aparse(string, delimiter, word_list) // outputs 3
first_word = word_list.get(1) // outputs "one"
last_word = word_list.get(nwords) // outputs "three"
```

ASC 関数

ASCII 照合シーケンスの文字の位置を返します。

カテゴリ: 文字列

返されるデータの種類: 整数

構文

ASC(*string*)

必須引数

string
ASCII 照合シーケンスで見つける必要がある文字を指定します。これは、文字列定数、フィールド名、または式として指定できます。

制限事項 複数の文字を指定した場合は、最初の文字のみ使用されます。

詳細

ASCII 値の完全なリストについては、付録 A: ASCII 値を参照してください。

例

```
ascii_value = asc("a") // outputs 97
character_content = chr(97) // outputs the letter "a"
```

CHR 関数

ASCII コードの ASCII 文字を返します。

カテゴリ: 文字列

返されるデータの種類: 文字列

構文

CHR(<*n*>)

必須引数

n
特定の ASCII 文字を表す整数を指定します。これは数値定数、フィールド名、または式で指定できます。

詳細

CHR 関数は、ASCII 照合シーケンスの n番目の文字を返します。

注: CHR 関数は、Unicode コードポイントを渡した場合に、任意の Unicode 文字を返すために使用することもできます。

ASCII 値の完全なリストについては、付録 A: ASCII 値を参照してください。

例

例 1

```
character_content = chr(97) // outputs the letter "a"
ascii_value = asc("a") // outputs 97
```

例 2

次の例では、Unicode コードポイントをサポートしています。

```
string input_string // this is a string that could contain greek characters
string(1) character
boolean greek_capital
for i=1 to len(input_string)
begin
  character=mid(input_string,i,1)
  if chr(character)>=913 and chr(character)<=939 then
    greek_capital=true
end
```

COMPARE 関数

2 つの文字列を比較した結果を返します。

カテゴリ:	文字列
返されるデータの種類:	整数

構文

COMPARE(*string1*, *string2*<,*case-insensitive*>)

必須引数

string1
比較に使用する文字列を指定します。これは、文字列定数、フィールド名、または式として指定できます。

string2
比較に使用する文字列を指定します。これは、文字列定数、フィールド名、または式として指定できます。

オプション引数

case-insensitive
大文字小文字を区別しない文字列を比較するかどうかを示すブール文字列を指定します。これは、文字列定数、フィールド名、または式で指定できます。

TRUE　文字列比較で大文字小文字を区別しないことを指定します。

FALSE　比較で大文字と小文字を区別することを指定します。

デフォルト　FALSE

詳細

MATCH_STRING 関数は、2 つの文字列を辞書編集上の観点から比較し、ワイルドカードを使用して文字列比較を行うことができます。

戻り値は、2 つの文字列の辞書編集上の観点からの比較の結果を表す整数です。

[-1 = string1 < string 2,
0 = string1 equals string2,
1 = string1 > string2]

2 つの文字列が等しいかどうかを調べるには、たとえば==演算子を使用するのがより効率的です。

```
if string1 == string2 then match=true
```

例

```
// hallo comes before hello when alphabetically sorted
rc = compare("hello", "hallo" ) // outputs 1

// Hello comes before hello when alphabetically sorted
rc = compare("Hello", "hello" ) // outputs -1

modifier = null
rc = compare("Hello", "hello", modifier ) // outputs -1
rc = compare("Hello", "hello", true ) // outputs 0
```

EDIT_DISTANCE 関数

文字列間の変換に適用する必要がある修正の数を返します。

カテゴリ:　文字列

返されるデータの種類:　整数

構文

EDIT_DISTANCE(*"string1"*, *"string2"*)

必須引数

"string1"
比較に使用する文字列を指定します。これは、文字列定数、フィールド名、または式として指定できます。

"string2"
比較に使用する文字列を指定します。これは、文字列定数、フィールド名、または式として指定できます。

詳細

EDIT_DISTANCE 関数は、*"string1"*を*"string2"*に変換するために適用する必要がある補正の数を返します。

例

```
distance = edit_distance("hello", "hllo" )
 // outputs 1

distance = edit_distance("hello", "hlelo" )
 // outputs 2

distance = edit_distance("hello", "hey" )
 // outputs 3
```

HAS_CONTROL_CHARS 関数

文字列に ASCII 制御文字が含まれているかを特定します。

カテゴリ: 文字列

返されるデータの種類: ブール値

構文

HAS_CONTROL_CHAR(*string*)

必須引数

string
ASCII 制御文字の有無を検索する文字列を指定します。

詳細

HAS_CONTROL_CHARS 関数は、ASCII 文字テーブルにある印刷不可能な ASCII 制御文字の識別に使用できます。戻り値が TRUE の場合は、制御文字が見つかったことを示します。戻り値が FALSE の場合は、制御文字が見つからなかったことを示します。

注: HAS_CONTROL_CHARS 関数が検出しない唯一の制御文字は 0(ヌル文字)です。

制御文字のリストについては、付録 B: ASCII 制御文字を参照してください。

例

```
boolean result_1

string error_message1

string test
test="Contol character: "&chr(13)
result_1=has_control_chars(test)
if(result_1)
  error_message1 = "test string contains control character"
else
  error_message1 = "test string does not contain control character"
```

INSTR 関数

別の文字列内に含まれる 1 つの文字列の位置を返します。

カテゴリ: 文字列

返されるデータの種類: 整数

構文

INSTR(*source*, *excerpt*<, *count*>)

必須引数

source
検索する文字列を指定します。これは、文字列定数、フィールド名、または式で指定できます。

excerpt
source 内で検索する文字列を指定します。これは文字列定数、フィールド名、または式で指定できます。

オプション引数

count
検索対象の *excerpt* の出現を示す整数を指定します。これは、数値定数、フィールド名、または式で指定できます。たとえば、値が 2 の場合は、*source* 中の *excerpt* の 2 回目の出現を検索することを示します。

詳細

INSTR 関数は、左から右に向かって *source* を検索し、*excerpt* の *count* 回目の出現を求めます。*excerpt* が *source* にない場合、この関数は 0 を返します。

例

```
source = "This is a simple sentence."
excerpt = "is"

position = instr(source, excerpt, 1) // outputs 3
position = instr(source, excerpt, 2) // outputs 6
```

LEFT 関数

文字列の左端の文字を返します。

カテゴリ:	文字列
返されるデータの種類:	文字列

構文

LEFT(*source*, *count*)

必須引数

source
検索する文字列を指定します。これは、文字列定数、フィールド名、または式で指定できます。

注 source が NULL の場合、この関数は NULL 値を返します。

count
返す文字数を示す整数を指定します。これは、数値定数、フィールド名、または式で指定できます。

注 0 以下のカウントを指定した場合は空文字列を返します。

例

```
source = "abcdefg"
result = left(source, 4) // outputs the string "abcd"
```

LEN 関数

文字列の長さを返します。

カテゴリ: 文字列

返されるデータの種類: 整数

構文

LEN(*source*)

必須引数

source
長さを決定する必要がある文字列を指定します; これは文字列定数、フィールド名、または式で指定できます。

注 空の文字列("")の長さは 0 です。*source* が NULL の場合、この関数は NULL 値を返します。

ヒント 前後のブランクを削除するには、トリム関数を使用します。

例

例 1

```
string(30) source
source = "abcdefg"
length_string = len(source) // outputs 7

source = " abcdefg "
length_string = len(source) // outputs 11

source = " " // source contains a blank
length_string = len(source) // outputs 1
```

例 2

```
function nvlString
return string len
//!params string inval string defaultVal

if isnull(parameter(1)) then return parameter(2)
else return parameter(1)
end function
```

LOWER 関数

文字列を小文字に変換します。

カテゴリ:	文字列
返されるデータの種類:	文字列

構文

LOWER(*source*)

必須引数

source
文字列を指定します。これは、文字列定数、フィールド名、または式で指定することができます。

注 *source* が NULL の場合、この関数は NULL 値を返します。

例

```
source = "MÜNCHEN in Germany"
lowcase_string = lower(source) // outputs "münchen in germany"
```

MATCH_STRING 関数

最初の文字列が 2 番目の文字列と一致するかを特定します。ワイルドカードを使用できます。

カテゴリ: 文字列

返されるデータの種類: ブール値

構文

MATCH_STRING(*string1*, *string2*)

必須引数

string1
検索する文字列を指定します。これは、文字列定数、フィールド名、または式で指定できます。

注 source が NULL の場合この関数は NULL 値を返します。

string2
検索パターンを表す文字列を指定します。これは、文字列定数、フィールド名、または式で指定できます。

詳細

MATCH_STRING 関数は、*string2* で指定された検索パターンを用いて *string1* を検索します。一致するものが見つかった場合は、true を返します。それ以外の場合は、false が返されます。

検索文字列には、先頭(*ABC)と末尾(ABC*)の位置にワイルドカードを含めることができます。文字列内のワイルドカードは無効です(A*BC)。

疑問符はワイルドカードとして使用できますが、文字にのみ一致します。たとえば、AB?は AB ではなく ABC と一致します。

ワイルドカードとして使用される文字の検索を実行するには、その文字の前にバックスラッシュを付けます。これは、その文字がワイルドカードとしてではなく、文字通りに使用されるべきであることを示します。有効な検索文字列は、*BCD*、*B?D*、*BCDE、*BC?E、*BCD?、ABCD*、AB?D*、?BCD*、*B??*、*B\?*になります(リテラル文字列 AB?\E)。無効な例は、AB*DE です。

より複雑な検索を行う場合は、MATCH_STRING()関数の代わりに正規表現を使用します。

例

```
string1 = "Monday is sunny, Tuesday is rainy & Wednesday is windy"
string2 = "Tuesday is"
match = match_string(string1, string2) // outputs false
string2 = "*Tuesday is*"
match = match_string(string1, string2) // outputs true
```

MID 関数

引数から部分文字列を抽出します。

カテゴリ: 文字列

返されるデータの種類: 文字列

構文

MID(*source*, *position*<, *length*>)

必須引数

source
検索する文字列を指定します。これは、文字列定数、フィールド名、または式で指定できます。

position
先頭文字の位置である整数を指定します。これは数値定数、フィールド名、式のいずれかで指定できます。

オプション引数

length
抽出する部分文字列の長さである整数を指定します。これは、数値定数、フィールド名、または式で指定できます。

デフォルト 長さが省略されている場合は、文字列の残りが抽出されます。

注 長さが NULL、ゼロ、または位置の後にソースに残っている式の長さよりも大きい場合、式の残りが返されます。

例

```
source = "06MAY15"

result = mid(source, 3, 3) // outputs "MAY"
result = mid(source, 3) // outputs "MAY15"
```

OCCURS 関数

1 つの文字列が別の文字列内で見つかった回数を返します。

カテゴリ: 文字列

返されるデータの種類: 整数

構文

OCCURS(*string1*, *string2*)

必須引数

string1
検索する文字列を指定します。

string2
カウントする部分文字列を表します。

詳細

OCCURS 関数は、文字列の中で部分文字列が出現した回数を返します。

例

```
string s
integer o
s="one:two:three"
o=occurs(s,":")
//the value o should contain 2 at this point
```

PARSE 関数

文字列をワードに解析し、見つかったワードの数と見つかったワードを返します。

カテゴリ: 文字列

返されるデータの種類: 整数

構文

PARSE(*string*, *delimiter*, *word1*<, *word2* , ...>)

必須引数

string
単語に区切る区切り文字を含む文字列を指定します。これは固定文字列、フィールド名、または式で指定できます。

注 *string* が NULL の場合、この関数は NULL 値を返します。*string* が空("")の場合は、値 1 を返します。

delimiter
文字列の単語を区切る文字を指定します。これは固定文字列、フィールド名、または式で指定できます。

word1
最初に見つかった単語を表す文字列を指定します。これはフィールド名で指定できます。

オプション引数

word2, ...
最初の文字列の後に見つかった文字列を順に表す 1 つ以上の文字列を指定します。これらはフィールド名で指定されます。

詳細

PARSE 関数は、*string* で見つかり、区切り文字で区切られた単語を指定されたパラメーターに代入します。戻り値は、見つかった単語の数を示します。

比較

APARSE 関数も同じです。APARSE 関数は、最大単語数をあらかじめに知る必要がないため、より自由度が高いです。文字列の最後の単語を簡単に判別することもできます。

例

```
string = "one:two:three"
delimiter = ":"
nwords = parse(string, delimiter, word1, word2) // outputs 3
// word1 will contain the value "one"
// word2 will contain the value "two"
```

PATTERN 関数

文字列に数字、大文字、小文字が含まれているかを示します。

カテゴリ: 文字列

返されるデータの種類: 文字列

構文

PATTERN(*string*)

必須引数

string
数値、大文字、小文字の評価対象となる文字列を指定します。これは固定文字列、フィールド名、または式で指定できます。

注 string が NULL の場合、この関数は NULL 値を返します。string が空("")の場合は、空の値を返します。

詳細

返された文字列には、各数字の代わりに 9、大文字には"A"、小文字には"a"が含まれています。その他の文字は入れ替えません。

例

```
source_string = "12/b Abc-Str."
result = pattern(source_string) // outputs "99/a Aaa-Aaa."
```

REPLACE 関数

ある文字列が最初に検索された箇所を別の文字列で置き換え、結果の文字列を返します。

カテゴリ: 文字列

返されるデータの種類: 文字列

構文

REPLACE(*source*, *search-string*, *replace-string*<, *count*>)

必須引数

source
検索する文字列を指定します。これは、文字列定数、フィールド名、または式で指定できます。

注 source が NULL の場合、この関数は NULL 値を返します。

search-string
置き換えるテキストを指定します。これは文字列定数、フィールド名、または式で指定できます。

replace-string
search-string を置き換える文字列を指定します。これは文字列定数、フィールド名、または式で指定できます。

オプション引数

count
いくつ置換を行うかを表す整数を指定します。これは、数値定数、フィールド名、または式で指定できます。

注 count が省略されている、またはゼロに設定されている場合は、文字列内のすべての出現が置換されます。

例

```
source_string =
```

```
    "It's a first! This is the first time I came in first place!"
search = "first"
replace = "second"
count = 2
result = replace(source_string, search, replace, count)
// outputs "It's a second! This is the second time I came in first place!"
```

RIGHT 関数

文字列の右端の文字を返します。

カテゴリ: 文字列

返されるデータの種類: 文字列

構文

RIGHT(*source*, *count*)

必須引数

source
検索する文字列を指定します。これは、文字列定数、フィールド名、または式で指定できます。

注 *source* が NULL の場合、この関数は NULL 値を返します。

count
返す文字数を示す整数を指定します。これは、数値定数、フィールド名、または式で指定できます。

注 *count* が 0 以下の場合は、空文字列が返されます。

例

```
source = "abcdefg"
result = right(source, 4) // outputs the string "defg"

source = "abcdefg "
result = right(source, 2) // outputs the string "fg "
```

SORT 関数

文字がアルファベット順に並べ替えらえた文字列を返します。

カテゴリ: 文字列

返されるデータの種類: 文字列

構文

SORT(*source* <, *ascending*, *remove_duplicates*>)

必須引数

source
並べ替える文字列を指定します。これは、文字列定数、フィールド名、または式で指定できます。

注 *source* が NULL の場合、この関数は NULL 値を返します。

オプション引数

ascending
テキストを昇順に並べ替えるかどうかを指定します。これは、ブール定数、フィールド名、または式で指定できます。この値は、TRUE または FALSE のどちらかを評価する必要があります。

TRUE 入力文字列は昇順に並べ替えられます。

FALSE 入力文字列は降順に並べ替えられます。

デフォルト TRUE

remove_duplicates
重複した文字を削除するかどうかを指定します。これは、ブール定数、フィールド名、または式で指定できます。この値は、TRUE または FALSE のどちらかを評価する必要があります。

TRUE 重複した文字は削除されます。

FALSE 重複した文字は削除されません。

デフォルト FALSE

詳細

順番を決める際には、特殊文字は頭文字の大文字より優先され、頭文字の大文字は小文字より優先されます。

例

```
source_string = "A short Sentence."
ascending = true
remove_duplicates = true
result = sort(source_string, ascending, remove_duplicates)
// outputs ".AScehnorst"
```

SORT_WORDS 関数

アルファベット順に並び替えられたワードで構成される文字列を返します。

カテゴリ: 文字列

返されるデータの種類: 文字列

注: ",.!"などの特殊文字は区切り文字として扱われません。

構文

SORT_WORDS(*source*<*ascending*>,<*remove_duplicates*>)

必須引数

source
ソートする文字列を指定します。文字列は文字列定数、フィールド名、式のいずれかで指定できます。

注 source が NULL の場合、この関数は NULL 値を返します。

オプション引数

ascending
入力文字列の単語を昇順に並べ替えるかどうかを指定します。これは、ブール定数、フィールド名、または式で指定できます。この値は、TRUE または FALSE のどちらかを評価する必要があります。

TRUE 入力文字列は昇順に並べ替えられます。

FALSE 入力文字列は降順に並べ替えられます。

デフォルト　TRUE

remove_duplicates
重複した単語を削除するかどうかを指定します。これは、ブール定数、フィールド名、または式で指定できます。この値は、TRUE または FALSE のどちらかを評価する必要があります。

TRUE　重複した単語は削除されます。

FALSE　重複した単語は削除されません。

デフォルト　FALSE

詳細

順番を決める際には、頭文字が大文字の単語は小文字より優先されます。また、特殊文字が連結された単語は別の単語として扱われます。たとえば、first と first!は別の単語です。

例

```
source_string =
    "It's a first! This is the first time I came in first place!"
ascending = true
remove_duplicates = true
result = sort_words(source_string, ascending, remove_duplicates)
// outputs "I It's This a came first first! in is place! the time"
```

TODATE 関数

システムのロケール設定に基づいて、引数を日付値に変換します。

カテゴリ:　文字列

操作:　TODATE( )関数は、お使いの Microsoft Windows 環境でデフォルトの **Short date** 地域設定に依存します。Windows でロケールの設定を変更できます。

返されるデータの種類:　日付

構文

TODATE(*any*)

必須引数

any
日付に変換する値を指定します。

詳細

TODATE( )関数が評価されると、Windows の **Short date** ロケールフィールドの値が調べられ、フォーマットが MM/DD/YYYY と DD/MM/YYYY のどちらであるか判断されます。月日の順番は、**Short date** 値で決定されます。

例

```
// Declare the date variable to contain the date value
date dateval

// Use the TODATE function to populate the date variable
dateval=todate(3750)

// Returns the value:
4/7/10 12:00:00 AM
```

TOINTEGER 関数

引数を整数値に変換します。

カテゴリ:	文字列
返されるデータの種類:	整数

構文

TOINTEGER(*any*)

必須引数

any
整数に変換する値を指定します。

例

例 1

```
if tointeger(counter)<= 5
setoutputslot(0)
else
setoutputslot(1)
```

例 2

```
// Declare an integer variable to contain the integer value
integer intval

// Use the TOINTEGER function to populate the integer variable
intval=tointeger(3750.12345)

// Returns the value:
3750
```

関連項目

タイプを強制する際のルールと特別な配慮については、次を参照してください。 “強制” (*式言語: リファレンスガイド*).

TOREAL 関数

引数を実数値に変換します。

カテゴリ:	文字列
返されるデータの種類:	実数

構文

TOREAL(*any*)

必須引数

any
実数に変換する値を指定します。

例

```
// Declare a real variable to contain the real value
real realval

// Use the TOREAL function to populate the real variable
realval=toreal(3750.12345)

// Returns the value:
3750.12345
```

関連項目

タイプを強制する際のルールと特別な配慮については、次を参照してください。 *"強制" (式言語: リファレンスガイド)*.

TOSTRING 関数

引数を文字列値に変換します。

カテゴリ:	文字列
返されるデータの種類:	文字列

構文

TOSTRING(*any*)

必須引数

any
文字列以外の値を指定して文字列に変換します。

例

```
// Declare a string variable to contain the string
String stringval

// Use the TOINTEGER function to populate the integer variable
stringval=tostring(3750.12345)

// Returns the string
3750.12345
```

関連項目

タイプを強制する際のルールと特別な配慮については、次を参照してください。 “強制” (*式言語: リファレンスガイド*).

TRIM 関数

先頭および末尾の空白を削除します。

カテゴリ: 文字列

返されるデータの種類: 文字列

構文

TRIM(*source*)

必須引数

source
文字列定数、s フィールド名、または式で指定することができます。

注 source が NULL の場合、この関数は NULL 値を返します。source が空の値 ("")の場合、この関数は空の値を返します。

例

```
source = " abcd " // 2 leading and 2 trailing spaces
result = trim(source) // outputs "abcd"
length = len(source) // outputs 8
length = len(result) // outputs 4
```

UPPER 関数

文字列を大文字に変換します。

カテゴリ: 文字列

返されるデータの種類: 文字列

構文

UPPER(*source*)

必須引数

source
文字列定数、フィールド名、式を指定します。

注 *source* が NULL の場合、この関数は NULL 値を返します。

例

```
source = "MÜNCHEN in Germany"
upcase_string = upper(source) // outputs "MÜNCHEN IN GERMANY"
```

USERNAME 関数

SAS Management Studio では、USERNAME 関数は、オペレーティングシステムにログインしているユーザーのユーザー名を返します。ただし、DataFlux Data Management Server では、USERNAME 関数は Data Management Server プロセスのアカウント名を返します。

カテゴリ: 文字列

返されるデータの種類: 文字列

構文

USERNAME ( )

引数なし

この関数は引数を取りません。

例

```
// Declare the string value for the function
string user

// For Data Management Studio, use the USERNAME
// function to get the operating system user name
user = username()
```

```
// For Data Management Server, use the USERNAME
// function to get the account name for the Data
// Management Server process.
user = username()
```

21

関数ウィンドウと通知ウィンドウの機能

Event Stream Processing の関数

ARRAYITEM

参照された配列のインデックス付きの値を返します。

arrayItem($*arrayfield*,*index*)

たとえば、ソースウィンドウの入力スキーマに次のフィールドが含まれているとします。

```
<field name="scores" type="array(dbl)"/>
```

イベントループでは、次のように配列を参照し、配列からインデックス値を取得できます。

```
<function name="score">arrayItem($scores,object_id)</function>
```

ここで、object_id は、機能ウィンドウのスキーマのフィールドです。

CONTQUERYNAME

現在のイベントを含む連続クエリの名前を返します。

contqueryName()

```
contqueryName()=cq_1
contqueryName()=myquery
```

ENGINEMETADATA

引数で指定されたエンジンメタデータ項目の値を返します。

engineMetadata(*argument*)

表 21.1 ENGINEMETADATA の引数

引数	説明
argument	値がエンジンメタデータ項目である文字列を指定します (**version** または **model** など)。

```
engineMetadata('version')=1.2
engineMetadata('model')=prodmodel
```

ESPCONFIGVALUE

引数で指定された構成値を返します。

espConfigValue(*argument*)

表 21.2 ESPCONFIGVALUE の引数

引数	説明
argument	構成値を指定します。構成値の詳細については、“エッジ環境での ESP サーバーの構成” (*SAS Event Stream*

引数	説明
	Processing: Edge 環境での ESP サーバーの使用)を参照してください。

esp-properties.yaml に次の値を指定した場合:

```
modelvalues:
  magicnumber: 11
```

関数は次を返します。

```
espConfigValue('meta.meteringhost')=espsrv01
product(espConfigValue('modelvalues.magicnumber'),10)=110
```

EVENTCOUNTER

関数ウィンドウによって生成された 0 ベースのイベント数を返します。

eventCounter()

```
string($id,'-',eventCounter())=eventid-0
string($id,'-',eventCounter())=eventid-1
string($id,'-',eventCounter())=eventid-2
```

EVENTNUMBER

現在の受信イベントによって生成された 0 ベースのイベント数を返します。数値は、関数が呼び出されるたびにインクリメントされます。

eventNumber()

```
string($id,'-',eventNumber())=eventid-0
string($id,'-',eventNumber())=eventid-1
string($id,'-',eventNumber())=eventid-2
```

EVENTSPROCESSED

出力ウィンドウによって処理された合計イベント数を返します。

eventsProcessed()

```
eventsProcessed()=10000
```

EVENTTIMESTAMP

現在のイベントのタイムスタンプを返します。

eventTimestamp()

```
eventTimestamp()=1506347669000000
```

INPUT

イベントストリーム処理入力ウィンドウの名前を返します。

input()

```
input()=sourceWindow
```

ISFAILOVERSTANDBY

実行中のプロジェクトのセットに、現在ホットフェイルオーバースタンバイモードでサブスクライバーコネクタを実行している 1 つ以上のプロジェクトが含まれている場合、**true** を返します。ホットフェイルオーバースタンバイモードで実行されているサブスクライバーコネクタがない場合は、**false** を返します。

isFailoverStandby()

```
isFailoverStandby()=true
```

ISLASTEVENTINBLOCK

処理中のイベントがイベントブロック内の最後のイベントである場合は true を返します。それ以外の場合は false を返します。

isLastEventInBlock()

```
isLastEventInBlock()=true
```

ISNOTRETENTION

このイベントが保持ポリシーによって生成されない場合は true を返します。このイベントが保持ポリシーによって生成された場合は false を返します。

isNotRetention()

isNotRetention()=true

ISRETENTION

イベントが保持ポリシーによって生成された場合は true を返します。イベントが保持ポリシーによって生成されない場合は false を返します。

isRetention()

isRetention()=true

OPCODE

現在のイベントの演算コードを返します。

opcode()

opcode()=insert
opcode()=upsert
opcode()=delete

OUTPUT

イベントストリーム処理出力ウィンドウの名前を返します。

output()

output()=myFunctionalWindow

PROJECTMETADATA

引数で指定されたプロジェクトメタデータ項目の値を返します。

projectMetadata(*projectname*,*argument*)

表 21.3 PROJECTMETADATA の引数

引数	説明
projectname	メタデータが返されるプロジェクトの名前を指定します。
argument	値がプロジェクトメタデータアイテムである文字列を指定します。

次の例では、ScoreProj というプロジェクトのデータをスコアリングするために使用される MAS モジュールの ID を返します。

```
projectMetadata('scoreproj','mm_linked_module1')=1.2
```

PROJECTNAME

現在のイベントを含むプロジェクトの名前を返します。

projectName()

```
projectName()=project_1
projectName()=myproject
```

TOLOWERUTF8

引数の値を小文字に変換します。引数はマルチバイトデータ (表現に複数バイトを必要とする文字) に対応できます。

toLowerUtf8(*string*)

表 21.4　TOLOWERUTF8 の引数

引数	説明
string	文字列を指定します。

この関数はマルチバイトデータを受け入れますが、実際にはこの種類のデータは処理しません。次の例では、大文字のシングルバイト文字は期待どおりに変換されますが、マルチバイト文字は変更されません。

```
toLowerUtf8("ZAŻÓŁĆ GĘŚLĄ JAŹŃ")=zaŻÓŁĆ gĘŚlĄ jaŹŃ
```

TOUPPERUTF8

引数の値を大文字に変換します。引数はマルチバイトデータに対応できます。

toUpperUtf8(*string*)

表 21.5　TOUPPERUTF8 の引数

引数	説明
string	文字列を指定します。

この関数はマルチバイト データを受け入れますが、実際にはこの種類のデータは処理しません。次の例では、小文字のシングルバイト文字は期待どおりに変換されますが、マルチバイト文字は変更されません。

```
toUpperUtf8("zażółć gęślą jaźń")=ZAżółć GęśLą JAźń
```

一般的な関数

ABS

指定された引数の絶対浮動小数点値を返します。

abs(*argument*)

表 21.6　ABS の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、**#myXML** と指定します。 ■ 関数。

```
abs(-55)=55
abs(44)=44
```

AND

両方の引数が true の場合は true を返します。それ以外の場合は false を返します。

and(*argument1* , *argument2*)

表 21.7　AND の引数

引数	説明
argument1 , *argument2*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、**#myXML** と指定します。 ■ 関数。

引数が数値を返す場合、関数は数値がゼロ以外の場合は true を返します。それ以外の場合は false を返します。

引数が文字列値を返す場合

- 文字列の値が 'true'の場合、関数は true を返します。
- 文字列の値が 'false'の場合、関数は false を返します。
- それ以外の場合、文字列の長さが> 0 の場合は true を返し、そうでない場合は false を返します。

```
and(gt(3,2),gt(2,5))=0
and(gt(3,2),gt(12,5))=1
and(0,1)=0
and('non empty string',55)=1
and('true',55)=1
and('',4)=0
```

BASE64DECODE

指定された base64 でエンコードされた文字列をデコードします。

base64Decode(*string*)

表 21.8　BASE64DECODE の引数

引数	説明
string	base64 でエンコードされた文字列を指定します。この文字列には長さ制限はありません。

```
base64Decode('dGhpcyBpcyBhIHRlc3Q=')=this is a test
```

BASE64DECODEBINARY

指定された base64 エンコード文字列をデコードし、結果をそのデータのバイナリ表現に設定します。

base64DecodeBinary(*string*)

表 21.9 BASE64DECODEBINARY の引数

引数	説明
string	base64 でエンコードされた文字列を指定します。この文字列には長さ制限はありません。

base64DecodeBinary('dGhpcyBpcyBhIHRlc3Q=')=<binary data>

BASE64ENCODE

指定された文字列をエンコードします。

base64Encode(*string*)

表 21.10 BASE64ENCODE の引数

引数	説明
string	テキスト文字列を指定します。この文字列には長さ制限はありません。

base64Encode('this is a test')=dGhpcyBpcyBhIHRlc3Q=

BASE64ENCODEBINARY

指定されたバイナリデータをエンコードし、エンコードされた文字列を返します。

base64Encode(binary_data)

base64EncodeBinary(<binary data>)=dGhpcyBpcyBhIHRlc3Q=

BETWEEN

最初の引数が 2 番目の引数より大きく、3 番目の引数よりも小さい場合は true を返します。それ以外の場合は false を返します。

表 21.11　BETWEEN の引数

引数	説明
argument1 , *argument2* , *argument3*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、**#myXML** と指定します。 ■ 関数。

```
between(20,17,30)=1
between(20,17,15)=0
```

BOOLEAN

指定された引数が文字列の場合、文字列の長さが 0 より大きい場合は true を返します。指定された引数が数値の場合、値が 0 でない場合は true を返します。指定された引数がブール式である場合、値が true の場合は true を返します。それ以外の場合は false を返します。

boolean(*argument*)

表 21.12　BOOLEAN の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。

引数	説明
	■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、**#myXML** と指定します。 ■ 関数。

注: 特殊な文字列値**'true'**および**'false'**は、文字列長> 0 の範囲外で処理されます。**'true'**の場合、関数は 1 を返します。**'false'**の場合、関数は 0 を返します。

```
boolean('my string')=1
boolean('')=0
boolean(10)=1
boolean(0)=0
boolean(gt(4,7))=0
boolean(gt(7,5))=1
```

CEILING

指定された引数の数値よりも上の整数値を返します。

ceiling(*argument*)

表 21.13　CEILING の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、**#myXML** と指定します。 ■ 関数。

```
ceiling(product(4,4.1))=17
```

COMPARE

最初の引数を 2 番目の引数と比較します。最初の引数が 2 番目の引数よりも小さい場合、-1 を返します。最初の値が 2 番目の値より大きい場合は、1 を返します。最初の値が 2 番目の値と等しい場合は、0 を返します。

compare(*argument1* , *argument2*)

表 21.14　COMPARE の引数

引数	説明
argument1 , *argument2*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、**#myXML** と指定します。 ■ 関数。

第 1 引数の種類は、等価性が文字列比較か数値比較かによって決定されるかどうかを決定します。

```
compare('bears','lions')=-1
compare('lions','bears')=1
compare('bears','bears')=0
compare(10,20)=-1
compare(20,10)=1
compare(10,10)=0
```

CONCAT

指定された引数の文字列値を連結した文字列を返します。

concat(*argument1* , *argument2* ,...<*argumentN* >)

表 21.15　CONCAT の引数

引数	説明
argument1 , ...*argument N*	次のいずれかを指定します。

引数	説明
	■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、**#myXML** と指定します。 ■ 関数。

最低でも 2 つの引数が必要です。

concat('Name: ','Joe',', Age: ',floor(sum(25,10)),'.') = Name: Joe, Age: 35.

CONCATDELIM

指定された区切り文字で区切った、指定値を連結した文字列を返します。

concatDelim(*'delimiter '*,*argument1* , *argument2* , ...<*argumentN* >)

表 21.16 CONCATDELIM の引数

引数	説明
'delimiter '	区切り文字として使用される文字を指定します。
argument1 , ...*argument N*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、**#myXML** と指定します。 ■ 関数。

concatDelim('.','www','sas','com') = www.sas.com

CONTAINS

最初の引数の文字列値に 2 番目の文字列値が含まれている場合は true を返します。それ以外の場合は false を返します。

contains(*argument1* , *argument2*)

表 21.17　CONTAINS の引数

引数	説明
argument1 , *argument2*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、**#myXML** と指定します。 ■ 関数。

```
contains('www.sas.com','sas') = true
contains('www.google.com','sas') = false
```

DECREMENT

指定された引数から 1 を引いた数値を返します。この関数は整数のみをサポートします。

decrement(*argument*)

表 21.18　DECREMENT の引数

引数	説明
argument	整数を指定します。

```
decrement(10)=9
```

DIFF

最初の引数から 2 番目の引数の値を引いた値を返します。

diff(*argument1* , *argument2*)

表 21.19 DIFF の引数

引数	説明
argument1 , *argument2*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、**#myXML** と指定します。 ■ 関数。

```
diff(sum(8,7),22)=-7.0
```

EQUALS

最初の引数が 2 番目の引数と等しい場合は true を返します。それ以外の場合は false を返します。

equals(*argument1* , *argument2*)

表 21.20 EQUALS の引数

引数	説明
argument1 , *argument2*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、**#myXML** と指定します。 ■ 関数。

比較のタイプは、最初の引数の型に依存します。

```
equals('sas.com',string('sas','.com'))=1
equals('sas.com','google.com')=0
equals(10,sum(5,5))=1
```

FALSE

引数のブール値が false の場合は true を返します。それ以外の場合は true を返します。

表 21.21　FALSE の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、**#myXML** と指定します。 ■ 関数。

```
false(0)=1
false(equals(10,10))=0
```

FLOOR

指定された引数の数値よりも小さい整数値を返します。

floor(*argument*)

表 21.22　FLOOR の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、**#myXML** と指定します。 ■ 関数。

floor(product(3.5,7))=24

GT

最初の引数が 2 番目の引数より大きい場合は true を返します。それ以外の場合は false を返します。

gt(*argument* , *argument2*)

表 21.23 GT の引数

引数	説明
argument , *argument2*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

比較のタイプは、最初の引数の型に依存します。

```
gt(sum(10,4),13)=true
gt('internet explorer','internet explorer')=false
gt('internet explorer','netscape')=false
```

GTE

最初の引数が 2 番目の引数以上の場合は true を返します。それ以外の場合は false を返します。

gte(*argument* , *argument2*)

表 21.24 GTE の引数

引数	説明
argument , *argument2*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。

引数	説明
	■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

比較のタイプは、最初の引数の型に依存します。

```
gte(sum(10,4),13)=true
gte('internet explorer','internet explorer')=true
gte('netscape','internet explorer')=true
```

GUID

グローバルに一意な識別子を返します。

guid()

```
guid()=46ca7b9e-b11d-41be-a3eb-be8bc8553aed
guid()=319cd2a6-1b30-4c1b-8ac7-55e9465ea066
```

INCREMENT

最初の引数+ 1 の数値を返します。この関数は整数のみをサポートします。

increment(*argument*)

表 21.25　INCREMENT の引数

引数	説明
argument	整数値または整数値を返す関数を指定します。

```
increment(10)=11
```

IF

最初の引数のブール値が true の場合、2 番目の引数を返します。それ以外の場合は、指定されている場合は 3 番目の引数を返します。

if(*argument1* , *argument2* , <*argument3* >)

表 21.26 IF の引数

引数	説明
argument1	ブール式を指定します。
argument2	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。
argument3	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
if(equals('x','x'),'one','two')=one
if(equals('x','y'),'one','two')=two
if(equals('x','y'),'one')=
```

IFNEXT

ペアの最初の引数を評価します。引数が true と評価されると、この関数はそのペアの 2 番目の引数の値を返します。

ifNext(*argument* , *argument2* , ...<*argumentN* >, <*argumentN+1* >)

表 21.27 IFNEXT の引数

引数	説明
argument	ブール式を指定します。
argument2	次のいずれかを指定します。

引数	説明
	■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
ifNext(gt(20,10),'value 1',lt(20,10),'value 2')=value 1
ifNext(gt(20,100),'value 1',lt(20,100),'value 2')=value 2
```

IN

expr0 が expr1 と一致する場合、expr1 と exprN の間の式が両立的な場合は true を返します。それ以外の場合は false を返します。

in(*expr0* , *expr1* <, ...*exprN*>)

表 21.28　IN の引数

引数	説明
expr0 , *expr1*	評価される式を指定します。式の最小数は 2 です。
exprN	評価される追加の式を指定します。

INDEX

引数 N の値を返します。N は指定されたインデックスの数値です。

index(*index* , *argument0* , ...<*argumentN* >)

表 21.29　INDEX の引数

引数	説明
Index	整数または整数を返す関数を指定します。

引数	説明
argument0 , *argument1* , ...*argument N*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

引数の最小数は 2 です。

```
index(1,'larry','moe','curly')=moe
index(random(0,4),10,20,30,40,50)=40
index(random(0,4),10,20,30,40,50)=20
```

INDEXOF

2 番目の引数の文字列値の最初の引数の文字列値の、0 から始まるインデックスを返します。値が見つからない場合は-1 を返します。

indexOf(*argument1* , *argument2*)

表 21.30　INDEXOF の引数

引数	説明
argument1 , *argument2*	文字列を指定します。

```
indexOf('SAS Event Stream Processing','Stream')=10
indexOf('SAS Event Stream Processing','Google')=-1
```

INTEGER

引数の整数値を返します。

integer(*argument*)

表 21.31　INTEGER の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
integer('88.45')=88
integer(111.23)=111
```

INTERVAL

最初の引数の数値をとり、残りのすべての引数の数値と比較します。最初の引数の数値が後に続く引数の 1 つよりも小さい場合、それに続く引数の値が返されます。

interval(*argument* , *argument2* , *argument3* , ...<*argumentN* >)

表 21.32　INTERVAL の引数

引数	説明
arguments	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
interval(85,60,'F',70,'D',80,'C',90,'B','A')=B
interval(90,60,'F',70,'D',80,'C',90,'B','A')=A
```

ISNULL

引数が設定されていない場合は true を返し、そうでない場合は false を返します。

isNull(*argument*)

表 21.33　ISNULL の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
isNull($unresolved)=1
isNull('my string')=0
```

ISSET

引数が設定されている場合は true を返し、そうでない場合は false を返します。

isSet(*argument*)

表 21.34　ISSET の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。

引数	説明
	■ 関数。

```
isSet($unresolved)=0
isSet('my string')=1
```

JSON

最初の引数で指定された JSON オブジェクトを解析し、2 番目の引数の関数として値を返します。

json(*argument* , *argument2*)

表 21.35　JSON の引数

引数	説明
argument	JSON オブジェクトを指定します。
argument2	評価文字列を指定します。名前の値のペアで、値を返すオブジェクトの名前を指定します。

```
json('{first:"john",last:"smith",hobbies:["running","reading","golf"]}',
        'first')=john
json('{first:"john",last:"smith",hobbies:["running","reading","golf"]}',
    'last')=smith
json('{first:"john",last:"smith",hobbies:["running","reading","golf"]}',
    'hobbies[1]')=reading
json(#myJson,'hobbies[1]')=reading
```

json JSONPath クエリ言語の構文のほとんどを実装します。詳細については、JSONPath Syntax ドキュメントを参照してください。ただし、基になるスクリプト言語の使用は、json 関数でサポートされていません。JSONPath では、かっこ内に配置され、スクリプト言語で記述された式を、明示的な名前またはインデックスの代替として使用できます。

次の例は、JSONPath フィルターを使用して、条件を満たす特定の要素を選択する方法を示しています。この例では、people 配列をフィルタリングして、age > 30 以上のオブジェクトを検索し、name プロパティを返します。

```
json('{
  "people": [
    {"name": "John", "age": 30},
    {"name": "Jane", "age": 25},
    {"name": "Doe", "age": 40}
  ]
}', 'people[?(@.age > 30)].name') = Doe
```

次の例は、ネストされたプロパティにアクセスし、スライシング(インデックス)を使用する方法を示しています。people[0].name は、people 配列の最初の人の name を取

得します。data.numbers[1:3]は、配列のスライス、具体的には 2 番目と 3 番目の要素を取得します。

```
json('{
  "people": [
    {"name": "John", "age": 30},
    {"name": "Jane", "age": 25},
    {"name": "Doe", "age": 40}
  ]
}', 'people[0].name') = John

json('{
  "data": {
    "numbers": [10, 20, 30, 40, 50]
  }
}', 'data.numbers[1:3]') = [20,30]
```

次の例では、フィルターとネストされたアクセスを組み合わせています。この例では、価格が price < 600 のオブジェクトを products 配列からフィルタリングし、その name を取得します。

```
json('{
  "products": [
    {"id": 101, "name": "Laptop", "price": 800},
    {"id": 102, "name": "Tablet", "price": 200},
    {"id": 103, "name": "Phone", "price": 500}
  ]
}', 'products[?(@.price < 600)].name') = ["Tablet","Phone"]
```

LASTINDEXOF

最初の文字列値の 2 番目の引数の文字列値の最後のインデックスを返します。値が見つからない場合は-1 を返します。

lastIndexOf(*argument* , *argument2*)

表 21.36　LASTINDEXOF の引数

引数	説明
argument , *argument2*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

lastIndexOf('http://www.sas.com/products/webanalytics','/')=27

LISTITEM

この関数には 2 つの用途があります。1) 指定された区切り文字を使用して最初の引数を解析し、指定されたインデックスに値を返します。2) 指定された関数コンテキスト内の既存のリストを参照し、その値を指定されたインデックスに返します。

listItem(*argument* , *delimiter* , *index*)

listItem(*reference* , *index*)

表 21.37 LISTITEM の引数

引数	説明
argument	区切り文字列を指定します。
delimiter	区切り文字として使用される文字を指定します。
index	インデックスとして使用される数値を指定します。
reference	関数コンテキストへの参照を指定します。

listItem('one,two,three,four',',',2)=three
listItem(#myList,0)=one

LISTSIZE

この関数には 2 つの用途があります。1) 指定された区切り文字を使用して最初の引数を解析し、そのサイズを返します。2) 指定された関数コンテキスト内の既存のリストを参照し、そのサイズを返します。

listSize(*argument* , *delimiter*)

listSize(*reference*)

表 21.38 LISTSIZE の引数

引数	説明
argument	区切り文字列を指定します。
delimiter	区切り文字として使用される文字を指定します。
reference	関数コンテキストへの参照を指定します。

```
listSize('one,two,three,four',',')=4
listSize(#myList)=4
```

LONG

引数の long 値を返します。

long(*argument*)

表 21.39　LONG の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
long('88.45')=88
long(111.23)=111
```

LT

最初の引数が 2 番目の引数より小さい場合は true を返します。それ以外の場合は false を返します。

lt(*argument* , *argument2*)

表 21.40　LT の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。

引数	説明
	■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。
argument2	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
lt(sum(10,4),13)=false
lt('internet explorer','internet explorer')=true
lt('internet explorer','netscape')=true
```

LTE

最初の引数が 2 番目の引数以下の場合は true を返します。それ以外の場合は false を返します。

lte(*argument* , *argument2*)

表 21.41　LTE の引数

引数	説明
argument , *argument2*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
lte(sum(10,3),13)=true
lt('internet explorer','internet explorer')=true
lt('internet explorer','netscape')=true
```

MAPVALUE

この関数には 2 つの用途があります。1) 指定された外部区切り文字と指定された内部区切り文字を使用して、最初の引数から名前と値のペアを解析し、指定された名前の値を抽出します。2) 参照された関数コンテキストの既存の値マップを参照し、その名前の値を抽出します。

mapValues(*argument* , *outerdelimiter* , *innerdelimiter* , *delimiter* , *name*)

mapValues(*#reference* , *name*)

表 21.42 MAPVALUE の引数

引数	説明
argument	名前と値のペアの区切り文字列を指定します。
outerdelimiter , *innerdelimiter delimiter*	区切り文字として使用される文字を指定します。
name	*argument* で指定された名前と値のペアの名前を指定します。
#reference	関数コンテキストへの参照を指定します。

```
mapValue('first:John;last:Doe;occupation:plumber',';',':','occupation')=plumber
mapValue(#myMap,'occupation')=plumber
```

MAPVALUES

この関数には 2 つの用途があります。1) 指定された外部区切り文字と指定された内部区切り文字を使用して、最初の引数から名前と値のペアを解析し、指定された名前ごとに値を抽出します。2) 参照された関数コンテキストの既存の値マップを参照し、指定された名前ごとに値を抽出します。

mapValues(*argument* , *outerdelimiter* , *innerdelimiter* , *delimiter* , *name1* ,... <*nameN* >)

mapValues(*#reference* , *name1* , ...<*nameN* >)

表 21.43 MAPVALUES の引数

引数	説明
argument	名前と値のペアの区切り文字列を指定します。

引数	説明
outerdelimiter , *innerdelimiter delimiter*	区切り文字として使用される文字を指定します。
name1 , ...*nameN*	*argument* で指定された名前と値のペアの名前を指定します。
#reference	関数コンテキストへの参照を指定します。

mapValues('first=John,last=Doe',',','=',':','first','last')=John:Doe
mapValues(#myMap,'first','last')=John:Doe

MAX

指定されたすべての引数の中で最大の数値を返します。

max(*argument* , *argument2* , ...<*argumentN* >)

表 21.44　MAX の引数

引数	説明
argument , *argument2* , ...*argument N*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

引数の最小数は 1 です。

max(33,44.2,sum(1,3,2,12),-33.21)=44.2

MEAN

指定されたすべての引数の平均値を返します。

mean(*argument* , *argument2* , ...<*argumentN* >)

表 21.45 MEAN の引数

引数	説明
argument , *argument2* , ...*argument N*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

mean(33,44.2,sum(1,3,2,12),-33.21)=15.4975

MIN

指定されたすべての引数の最小の数値を返します。

min(*argument* , *argument2* , ...<*argumentN* >)

表 21.46 MIN の引数

引数	説明
argument , *argument2* , ... *argumentN*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

引数の最小数は 1 です。

min(33,44.2,sum(1,3,2,12),-33.21)=-33.21

MOD

最初の引数の残りを秒で割ったものを返します。

mod(*argument* , *argument2*)

表 21.47　MOD の引数

引数	説明
argument , *argument2*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

mod(10,3)=1.0

NEG

指定された引数の負の数値を返します。

neg(*argument*)

表 21.48　NEG の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

neg(55)=-55

NORMALIZESPACE

指定された引数の余分な空白を単一のスペースで置き換えることによって作成される文字列を返します。

normalizeSpace(*argument*)

表 21.49 NORMALIZESPACE の引数

引数	説明
argument	文字列を指定します。

normalizeSpace('Sentence with many spaces')=Sentence with many spaces

NEQUALS

最初の引数が 2 番目の引数と等しくない場合は true を返します。それ以外の場合は false を返します。

nequals(*argument* , *argument2*)

表 21.50 NEQUALS の引数

引数	説明
argument , *argument2*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

nequals('sas.com',string('sas','.com'))=0
nequals('sas.com','google.com')=1
nequals(10,sum(5,5))=0

NEWLINE

指定された改行文字の数を返します。引数なしで、単一の新しい行を返します。

newline(<*number*>)

表 21.51　NEWLINE の引数

引数	説明
number	*number* の改行文字を返します。

NOT

引数のブール値が false の場合は true を返します。それ以外の場合は false を返します。

not(*argument*)

表 21.52　NOT の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
not(0)=1
not(equals(10,10))=0
```

NUMBER

引数の数値を返します。

number(*argument*)

表 21.53 NUMBER の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
number('88.45')=88.45
number(111.23)=111.23
number(gt(2,1))=1
```

OR

指定された引数のいずれかが true の場合は true を返します。それ以外の場合は false を返します。

or(*argument* , *argument2* , ...<*argumentN* >)

表 21.54 OR の引数

引数	説明
argument , *argument2* , ...*argument N*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

引数の最小数は 1 です。

```
or(equals('a','b'),nequals('a','b'))=1
or(equals('a','b'),nequals('a','a'))=0
```

OUTSTR

引数に文字列または文字列グループのいずれかが含まれている場合は、関連する値を返します。一致するものがなければ、指定されたデフォルト値を返します。

outstr(*argument* , *string1* , *value_associated_with_string1* , ...<*stringN* >, <*value_associated_with_stringN* >, *default*)

表 21.55　OUTSTR の引数

引数	説明
argument	文字列を指定します。
string1 ...stringN	文字列または文字列のグループを指定します。
value_associated_with_string1 ...value_associated_with_stringN	文字列に関連付けられた値、または文字列のグループに関連付けられた値を指定します。
default	文字列または文字列のグループを指定します。

```
outstr('government spending','govern','Government','Other')=Government
outstr('spending',('govern','spend'),
     'Government or Spending','Other')=Government or Spending
outstr('bob',('govern','spend'),'Government or Spending',
    ('john','jack','bob'),'Names','Other')=Names
outstr('stream processing',('govern','spend'),'Government or Spending',
     ('john','jack','bob'),'Names','Other')=Other
```

PRECISION

最初の引数の小数点の precision を 2 番目の引数に設定します。

precision(*argument* , *argument2*)

表 21.56　PRECISION の引数

引数	説明
argument , *argument2*	数値を指定します。

```
precision(123.44567,2)=123.45
```

PRODUCT

指定された引数の積を返します。

product(*argument* , *argument2* ...<*argumentN* >)

表 21.57　PRODUCT の引数

引数	説明
argument , *argument2* , ...*argument N*	数値または数値を返す関数を指定します。

```
product(3,sum(2,4),2)=36
```

QUOTIENT

指定された引数の商を返します。

quotient(*argument* , *argument2* ...<*argumentN* >)

表 21.58　QUOTIENT の引数

引数	説明
argument , *argument2* , ...*argument N*	数値または数値を返す関数を指定します。

```
quotient(3,sum(2,4),2)=0.25
```

RANDOM

最初の引数と 2 番目の引数の間の乱数を返します。

random(*argument* , *argument2*)

表 21.59　RANDOM の引数

引数	説明
argument , *argument2*	数値を指定します。

```
random(100,1000)=741
random(100,1000)=356
random(100,1000)=452
precision(random(0,.5),2)=0.17
```

RGX

指定された正規表現を指定された文字列で実行し、結果を返します。グループが指定されている場合、結果は指定された数値正規表現グループの内容になります。

rgx(*regular_expression* , *string* ...<*group* >)

表 21.60　RGX の引数

引数	説明
regular_expression	関数コンテキストの正規表現または正規表現への参照を指定します。
string	文字列を指定します。
group	正規表現への数値参照を指定します。

```
rgx('.*/view/([0-9]*)/([0-9]*)',
        'http://cistore-dev.unx.sas.com/products/view/23/4',
        1)=23
rgx(#myExpr,
   'http://cistore-dev.unx.sas.com/products/view/23/4'
   ,2)=4
```

RGXINDEX

与えられた文字列に対して、指定された正規表現を実行します。一致するものが見つかると、その一致のインデックスを返します。一致するものが見つからない場合は-1 を返します。

rgxIndex(*regular_expression* , *string* ...<*stringN* >)

表 21.61 RGXINDEX の引数

引数	説明
regular_expression	関数コンテキストの正規表現または正規表現への参照を指定します。
string ...stringN	文字列を指定します。

```
rgxIndex('developer','larry - manager','moe - tester','curly - developer')=2
```

RGXLASTTOKEN

最初の引数の正規表現を 2 番目の正規表現内の区切り文字として使用して、その式で区切られたすべての文字列を検索します。

rgxLastToken(*regular_expression1* , *regular_expression2 ...<index_value* >)

表 21.62 RGXLASTTOKEN の引数

引数	説明
regular_expression1	関数コンテキストの正規表現または正規表現への参照を指定します。
regular_expression2	正規表現を指定します。
index_value	正規表現の最後のトークンから数えられるインデックス値(デフォルトは 0)を指定します。この値が 0 より大きく、正規表現のトークンの数以下の場合、その値のトークンが返されます。それ以外の場合は、ヌルが戻されます。

```
rgxLastToken('/','data/opt/sas/dataflux')=dataflux
rgxLastToken('/','data/opt/sas/dataflux',2)=opt
rgxLastToken('\.','www.sas.com')=com
```

RGXMATCH

最初の引数の正規表現を 2 番目の引数と比較し、一致するものがあるかどうかを示すブール値を返します。

rgxMatch(*regular_expression1* , *string ...<group* >)

表 21.63　RGXMATCH の引数

引数	説明
regular_expression1	関数コンテキストの正規表現または正規表現への参照を指定します。
string	文字列を指定します。
group	正規表現への数値参照を指定します。指定すると、結果は指定された数値の正規表現グループの内容になります。

```
rgxMatch('(google|yahoo|bing)','http://www.google.com')=1
rgxMatch('(google|yahoo|bing)','http://www.sas.com')=0
```

RGXREPLACE

最初の引数の正規表現を 2 番目の引数の文字列と比較して解析し、最初の一致を 3 番目の引数の文字列に置き換えます。

rgxReplace(*regular_expression1* , *string1* , *string2*)

表 21.64　RGXREPLACE の引数

引数	説明
regular_expression1	関数コンテキストの正規表現または正規表現への参照を指定します。
string	文字列を指定します。
string2	文字列を指定します。

```
rgxReplace('(google|yahoo|bing)','http://www.google.com','sas')=http://www.sas.com
```

RGXREPLACEALL

最初の引数の正規表現を 2 番目の引数の文字列と比較して解析し、一致するものを 3 番目の引数の文字列に置き換えます。

rgxReplaceAll(*regular_expression1* , *string1* , *string2*)

表 21.65 RGXREPLACEALL の引数

引数	説明
regular_expression1	関数コンテキストの正規表現または正規表現への参照を指定します。
string	文字列を指定します。
string2	文字列を指定します。

```
rgxReplaceAll('(google|yahoo|bing)','http://www.google.com/google/products','sas')
    =http://www.sas.com/sas/products
```

RGXTOKEN

第 1 引数を第 2 引数内の区切り文字として使用し、区切り文字で区切られたすべての文字列を検索します。

rgxToken(*delimiter* , *string* , <*index* >)

表 21.66 RGXTOKEN の引数

引数	説明
delimiter	関数コンテキストの正規表現または正規表現への参照を指定します。
string	文字列を指定します。
index	文字列の解析時にインデックスとして機能する数値を指定します。インデックスが文字列内のトークンの数以下の場合、インデックス値のトークンが返されます。それ以外の場合は、ヌルが戻されます。デフォルト値は 0 です。

```
rgxToken('/','data/opt/sas/dataflux')=data
rgxToken('/','data/opt/sas/dataflux',2)=sas
rgxToken('\.','www.sas.com')=www
```

RGXV

最初の引数の正規表現を 2 番目の引数の文字列と比較して解析し、指定された区切り文字で区切られたすべての一致を返します。

rgxV(*regular_expression* , *string* , *delimiter* <*group* >)

表 21.67　RGXV の引数

引数	説明
regular_expression	関数コンテキストの正規表現または正規表現への参照を指定します。
string	文字列を指定します。
delimiter	*string* を解析する際の区切り文字となる文字値を指定します。
group	正規表現への数値参照を指定します。指定すると、結果は指定された数値の正規表現グループの内容になります。

```
rgxV('(jerry|scott|vince)',
        'The ESP product has jerry, scott, and vince working on it',
        ' : ')=jerry : scott : vince
```

ROUND

引数の丸められた数値を返します。

round(*argument*)

表 21.68　ROUND の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
round(34.56)=35
round(34.46)=34
```

SETCONTAINS

この関数には 2 つの用途があります。1) 指定された区切り文字を含む指定されたトークンセットを解析し、指定された文字列がセット内に現れるかどうかをチェックします。2) 指定された文字列がセットに含まれているかどうかを確認するために、関数コンテキストの既存のトークンセットを参照します。

setContains(*set_of_tokens* , *delimiter* , *string*)

setContains(*#reference string*)

表 21.69　SETCONTAINS の引数

引数	説明
set_of_tokens	トークンの文字列を指定します。
delimiter	区切り文字として使用される文字値を指定します。
string	文字列を指定します。
reference	関数コンテキストへの参照を指定します。

```
setContains('one,two,three,four',',','two')=1
setContains(#mySet,'five')=0
```

STARTSWITH

最初の引数が 2 番目の引数で開始する場合は true を返します。それ以外の場合は false を返します。

startsWith(*argument1* , *argument2*)

表 21.70　STARTSWITH の引数

引数	説明
argument1 , *argument2*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。

引数	説明
	■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
startsWith('www.sas.com','www.')=1
startsWith('www.sas.com','sww.')=0
```

STRING

1 つ以上の引数から連結された単一文字列値を返します。

string(*argument* <, *argument2* ,...*argumentN* >)

表 21.71　STRING の引数

引数	説明
argument(s)	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
string(33.9)=33.9
string($id,'-',eventNumber())=eventid-0
```

STRINGLENGTH

引数の文字列値の長さを返します。

stringLength(*argument*)

表 21.72 STRINGLENGTH の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
stringLength('SAS ESP XML')=11
```

STRIP

引数から先頭または末尾の空白を削除した後に作成される文字列を返します。

strip(*argument*)

表 21.73 STRIP の引数

引数	説明
argument	文字列を指定します。

```
strip('      SAS ESP XML    ')=SAS ESP XML
```

SUBSTRING

指定されたインデックスにある最初の引数の値の部分文字列をとって作成された文字列を返します。

substring(*argument* , *index* , <*length* >)

表 21.74 SUBSTRING の引数

引数	説明
argument	文字列を指定します。

引数	説明
index	*argument* を解析するためのインデックスを定義する数値を指定します。
length	数値を指定します。指定すると、部分文字列は *length* のサイズになります。そうでない場合は、文字列の最後までのすべての文字が含まれます。

```
substring('www.sas.com',4,3)=sas
substring('www.sas.com',4)=sas.com
```

SUBSTRINGAFTER

二番目の引数の値の出現後に、最初の引数の値をとった結果の文字列を返します。

substringAfter(*argument1* , *argument2* , <*index* >)

表 21.75 SUBSTRINGAFTER の引数

引数	説明
argument1 , *argument2*	文字列を指定します。
index	*argument1* を解析するためのインデックスを定義する数値を指定します。指定すると、その発生以降の内容が返されます。

```
substringAfter('www.sas.com','.')=sas.com
substringAfter('www.sas.com','.',2)=com
```

SUBSTRINGBEFORE

二番目の引数の値が出現する前に、最初の引数の値をとった結果の文字列を返します。

substringBefore(*argument1* , *argument2* , <*index* >)

表 21.76 SUBSTRINGBEFORE の引数

引数	説明
argument1 , *argument2*	文字列を指定します。

引数	説明
index	*argument1* を解析するためのインデックスを定義する数値を指定します。指定すると、その発生以降の内容が返されます。

```
substringBefore('www.sas.com','.')=www
substringBefore('www.sas.com','.',2)=www.sas
```

SUM

すべての引数の数値の合計を返します。

sum(*argument* , *argument2* ...<*argumentN* >)

表 21.77　SUM の引数

引数	説明
argument , *argument2* , ...*argumentN*	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
sum(33,22,55.4,34,min(0,4,-9))=135.4
```

SWITCH

二番目の引数で始まる引数を解析します。引数が最初のものと一致すると、次の引数を返します。一致するものが見つからなければ、ヌルを返します。

switch(*argument* , *argument2* , ...<*argumentN* >, <*argumentN+1* >)

表 21.78　SWITCH の引数

引数	説明
argument , argument2 ...argumentN +1	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
switch('bob','jerry','manager','bob','developer')=developer
switch('jerry','jerry','manager','bob','developer')=manager
switch('moe','jerry','manager','bob','developer')=
```

SYSTEMMICRO

1970 年 1 月 1 日から現在時刻までのマイクロ秒数を返します。

systemMicro()

```
systemMicro()=1420557039483912
```

SYSTEMMILLI

1970 年 1 月 1 日から現在時刻までのミリ秒数を返します。

systemMilli()

```
systemMilli()=1420557039483
```

TIMECURRENT

現在の時刻を返します。

timeCurrent()

```
timeCurrent()=1421157236
timeString(timeCurrent())=Tue Jan 13 08:53:56 2015
```

TIMEDAYOFMONTH

現在または指定された時刻の日付けを返します。

timeDayOfMonth(<*argument* >)

表 21.79　TIMEDAYOFMONTH の引数

引数	説明
argument	特定の時間を定義する式、または特定の時間を返す関数を指定します。

```
timeDayOfMonth()=13
timeDayOfMonth(timeParse('06/21/2015 00:00:00','%m/%d/%Y %H:%M:%S'))=21
```

TIMEDAYOFWEEK

現在または指定された時刻の曜日を返します。

timeDayOfWeek(<*argument* >)

表 21.80　TIMEDAYOFWEEK の引数

引数	説明
argument	特定の時間を定義する式、または特定の時間を返す関数を指定します。0 - 6(日曜日から土曜日)の値を返します。

```
timeDayOfWeek()=2
timeDayOfWeek(timeParse('06/21/2015 00:00:00','%m/%d/%Y %H:%M:%S'))=0
```

TIMEDAYOFYEAR

現在または指定された時刻の通日(DOY)を返します。

timeDayOfYear(<*argument* >)

表 21.81　TIMEDAYOFYEAR の引数

引数	説明
argument	特定の時間を定義する式、または特定の時間を返す関数を指定します。

```
timeDayOfYear()=12
timeDayOfYear(timeParse('06/21/2015 00:00:00','%m/%d/%Y %H:%M:%S'))=171
```

TIMEGMTTOLOCAL

引数に指定されている GMT を現地時間に変換します。

表 21.82　TIMEGMTTOLOCAL の引数

引数	説明
argument	時間値または時間値を返す関数を指定します。

```
timeString(timeGmtToLocal(timeCurrent()))=Tue Jan 13 02:10:08 2015
```

TIMEGMTSTRING

最初の引数で表される GMT 時間を出力します。

timeGmtString(*argument* , <*argument2* >)

表 21.83　TIMEGMTSTRING の引数

引数	説明
argument	時間値または時間値を返す関数を指定します。
argument2	時間形式を指定します。

```
timeString(timeCurrent(),'%Y-%m-%d %H:%M:%S %Z')=2015-02-20 07:57:18 EST
timeGmtString(timeCurrent(),'%Y-%m-%d %H:%M:%S %Z')=2015-02-20 12:57:18 GMT
```

TIMEHOUR

現在または指定された時刻の時刻を返します。

timeHour(<*argument* >)

表 21.84　TIMEHOUR の引数

引数	説明
argument	特定の時間を定義する式、または特定の時間を返す関数を指定します。

```
timeHour()=9
timeHour(timeParse('06/21/2015 13:45:15','%m/%d/%Y %H:%M:%S'))=14
```

TIMEMICRO

指定された引数のマイクロ秒数、または引数が指定されていない場合は 1970 年 1 月 1 日からのマイクロ秒数を返します。

timeMicro(*argument*)

表 21.85　TIMEMICRO の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
timeMicro()=1553283757000000
timeMicro(timeParse('06/21/2015 00:00:00','%m/%d/%Y %H:%M:%S'))=1434859200000000
```

TIMEMILLI

指定された引数のミリ秒数、または引数が指定されていない場合は 1970 年 1 月 1 日からのマイクロ秒数を返します。

timeMilli(*argument*)

表 21.86　TIMEMILLI の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
timeMilli()=1553283876000
timeMilli(timeParse('06/21/2015 00:00:00','%m/%d/%Y %H:%M:%S'))=1434859200000
```

TIMEMINUTE

現在または指定された時刻の分を返します。

timeMinute(<*argument* >)

表 21.87　TIMEMINUTE の引数

引数	説明
argument	特定の時間を定義する式、または特定の時間を返す関数を指定します。

```
timeMinute()=22
timeMinute(timeParse('06/21/2015 13:45:15','%m/%d/%Y %H:%M:%S'))=45
```

TIMEMINUTEOFDAY

現在または指定された時刻の、その日に入ってからの分数を返します。

timeMinuteOfDay(<*argument* >)

表 21.88 TIMEMINUTEOFDAY の引数

引数	説明
argument	特定の時間を定義する式、または特定の時間を返す関数を指定します。

```
timeMinuteOfDay()=563
timeMinuteOfDay(timeParse('06/21/2015 13:45:15','%m/%d/%Y %H:%M:%S'))=885
```

TIMEPARSE

最初の引数で指定された時刻を表す文字列を返します。

timeParse(*time* ,<*format* >)

表 21.89 TIMEPARSE の引数

引数	説明
time	時間指定または時間指定を返す関数を指定します。
format	時間形式を指定します。時間形式を指定しない場合は、デフォルトのシステム時間形式が使用されます。 時間形式は、UNIX の strptime()関数でサポートされているものに準拠している必要があります。

```
timeParse(timeString())=1421159135
timeParse('01/01/2015 00:00:00','%m/%d/%Y %H:%M:%S')=1420088400
```

TIMESECOND

現在または指定された時刻の秒を返します。

timeSecond(<*argument* >)

表 21.90　TIMESECOND の引数

引数	説明
argument	特定の時間を定義する式、または特定の時間を返す関数を指定します。

```
timeSecond()=37
timeSecond(timeParse('06/21/2015 13:45:15','%m/%d/%Y %H:%M:%S'))=15
```

TIMESTAMP

現在の時刻を文字列で返します。

timeStamp(<*format* >)

表 21.91　TIMESTAMP の引数

引数	説明
format	UNIX の strftime 関数でサポートされている時間形式を指定します。

```
timeStamp()=Thu Feb 19 15:09:35 2015
timeStamp('%m-%d-%Y')=02-19-2015
```

TIMESTRING

最初の引数で表される時刻を返します。

timeString(*time* ,<*format* >)

表 21.92　TIMESTRING の引数

引数	説明
time	時間指定または時間指定を返す関数を指定します。
format	時間形式を指定します。時間形式を指定しない場合は、デフォルトのシステム時間形式が使用されます。 時間形式は、UNIX の strftime()関数でサポートされているものに準拠している必要があります。

```
timeString(timeCurrent())=Thu Feb 19 15:09:35 2015
```

```
timeString(timeCurrent(),'%m-%d-%Y')=02-19-2015
```

TIMESECONDOFDAY

現在または指定された時刻の、その日に入ってからの秒数を返します。

timeSecondofDay(<*argument* >)

表 21.93 TIMESECONDOFDAY の引数

引数	説明
argument	特定の時刻を定義する式、または特定の時刻を返す関数を指定します。

```
timeSecondOfDay()=34035
timeSecondOfDay(timeParse('06/21/2015 13:45:15','%m/%d/%Y %H:%M:%S'))=53115
```

TIMETODAY

現地時間を基準にして、現在の日の最初の秒を表す値を返します。

timeToday()

```
timeToday()=1421125200
```

TIMEYEAR

1900 年から現在または指定された時刻までの年数を返します。

timeYear(<*argument* >)

表 21.94 TIMEYEAR の引数

引数	説明
argument	特定の時間を定義する式、または特定の時間を返す関数を指定します。

```
timeYear()=115
timeYear(timeParse('06/21/2013 13:45:15','%m/%d/%Y %H:%M:%S'))=113
```

TOLOWER

引数の値を小文字に変換します。

toLower(*string*)

表 21.95　TOLOWER の引数

引数	説明
string	文字列を指定します。

toLower('Http://Www.Sas.Com/Products/Esp')=http://www.sas.com/products/esp

TOUPPER

引数の値を大文字に変換します。

toUpper(*string*)

表 21.96　TOUPPER の引数

引数	説明
string	文字列を指定します。

toUpper('Http://Www.Sas.Com/Products/Esp')=HTTP://WWW.SAS.COM/PRODUCTS/ESP

TRANSLATE

2 番目の引数の各文字について、最初の引数に対応する文字を見つけ、3 番目の引数に対応する文字に置き換えます。

translate(*argument1* , *argument1* , *argument3*)

表 21.97　TRANSLATE の引数

引数	説明
argument1、*argument2*、*argument3*	文字列を指定します。*argument2* と *argument3* の長さは同じでなければなりません

```
translate('replace all vowels with its capital equivalent','aeiou','AEIOU')
      =rEplAcE All vOwEls wIth Its cApItAl EqUIvAlEnt
```

TRUE

引数のブール値が true の場合、true を返します。それ以外の場合は false を返します。

true(*argument*)

表 21.98　TRUE の引数

引数	説明
argument	次のいずれかを指定します。 ■ リテラル値(文字列または数値)。文字列値は、一重引用符または二重引用符で囲みます。 ■ イベントフィールドから解決される値。フィールド名の前に$文字を付けます。 ■ リソースを参照する値。たとえば、xpath 関数を使用して myXML という名前の XML オブジェクトを参照する場合は、次のように指定します。**#myXML**。 ■ 関数。

```
true('testing')=1
true('')=0
true(gt(10,5))=1
```

URLDECODE

引数で表される URL をデコードします。

urlDecode(*argument*)

表 21.99　URLDECODE の引数

引数	説明
argument	文字列を指定します。

URL のエンコードおよびデコードの詳細については、このリファレンスを参照してください。

```
urlDecode('http%3A%2F%2Fwww%2Esas%2Ecom%2Fproducts%2Fevent%20stream
%20processing')
=http://www.sas.com/products/event stream processing
```

URLENCODE

引数で表される URL をエンコードします。

urlEncode(*argument*)

表 21.100　URLENCODE の引数

引数	説明
argument	文字列を指定します。

URL のエンコードおよびデコードの詳細については、このリファレンスを参照してください。

```
urlEncode('http://www.sas.com/products/event stream processing')
=http%3a%2f%2fwww%2esas%2ecom%2fproducts%2fevent%20stream%20processing
```

XPATH

最初の引数に XML を解析し、XML コンテキストの 2 番目の引数を評価します。

xpath(*argument1* , *argument2* , <*argument3* >)

表 21.101　XPATH の引数

引数	説明
argument1	XML のインスタンスを指定します。これは、有効な XML テキストコンテキストまたは他の場所の XML への参照によって表されます。
argument2	評価文字列を指定します。
argument3	関数が複数の結果を返すときに使用するセパレーターを指定します。

```
xpath('<info><name>john smith</name><hobby>running</hobby>
          <hobby>reading</hobby><hobby>golf</hobby></info>',
         './/name/text()')=john smith
xpath(#myXml,'.//hobby/text()',',')=running,reading,golf
```

3 部

ウィンドウと対話するその他の方法

22

スコアリング API の使用

スコアリング API の利点

スコアリング API を使用すると、入力イベントをスコアリングし、オンデマンドでスコアリングされたイベントを返すことができます。

スコアリング API は、要求が実行中の ESP プロジェクトに送信され、応答が返される要求応答通信パターンを使用します。コミュニケーションパターンは同期的です。システムは応答を待ってから続行します。これは、データが継続的かつ非同期的に流れるストリーミングやパブリッシュ/サブスクライブの通信パターンとは対照的です。

スコアリング API を使用して、ESP プロジェクトの設計とデバッグを支援できます。コネクタとアダプターの構成は、プロジェクトを実行する環境によって異なります。SAS Event Stream Processing Studio を使用して、ライブデータが利用できない環境でプロジェクトを設計およびテストする場合があります。スコアリング API を使用すると、ライブデータソースを必要とせずに、またはライブデータソースに加えてデータ入力をシミュレートできます。SAS Event Stream Manager を使用して、プロジェクトのプロジェクトコンテナーを作成し、さまざまな環境に配置します。詳細については、*"Working with Project Containers" (SAS Event Stream Processing: Using SAS Event Stream Manager)*を参照してください。

スコアリング API の他のユースケースを次に示します。

- リアルタイム不正チェック。金融機関がトランザクションを処理する前に不正がないか確認したいとします。スコアリング API を使用して、トランザクション

金額、場所、ユーザー ID などのフィールドを含む要求イベントを送信できます。SAS Event Stream Processing は、学習済み不正検出モデルを参照するスコアウィンドウにイベントをルーティングします。応答イベントには不正リスクスコアが含まれる場合があります。

- 即座のパーソナライズ。小売 Web サイトで、ユーザーのビヘイビアーに基づいてパーソナライズされた商品のレコメンデーションが提供される必要があるとします。ユーザーがサイトにアクセスして商品を表示すると、ユーザーのプロファイルとアクティビティデータとともにスコアリング要求を送信できます。SAS Event Stream Processing は、パーソナライズモデルを適用して、カスタマイズされたレコメンデーションを生成できます。応答イベントには、レコメンデーションされた商品を含めることができます。
- オンデマンドスコアリング。カスタマーサポートエージェントが顧客と電話をしていて、その顧客が製品またはサービスの使用を停止するリスクを評価したいとします。顧客データを伴うスコアリング要求を送信し、SAS Event Stream Processing は予測モデルに基づいてリスクスコアを返すことができます。エージェントはスコアを使用して、維持戦略をガイドできます。
- イベント駆動型のオートメーション。スマートファクトリでは、センサーデータが機械の破損を示したときにメンテナンスアクションをトリガーしたいとします。センサーデータは、スコアリング API を使用して SAS Event Stream Processing に送信できます。SAS Event Stream Processing は、予測メンテナンスモデルを使用してデータを評価することができます。スコアがしきい値を超えると、さらにアクションがトリガーされます。

スコアリング API の機能の概要

スコアリング要求は、複数の入力イベントと複数の出力イベントをサポートします。要求の JSON コードは、すべての入力と出力を配列で囲みます。

次に要求の例を示します。

```
curl -X POST http://myserver:1234/eventStreamProcessing/v2/score --data-binary
@input.json
```

スコアリング API に関連する情報は、project 要素の次の属性を使用して、プロジェクトの XML コードに追加されます。

- score-input-window
- score-input-field
- score-output-window
- score-output-field

これらの属性の詳細については、“project” (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

SAS Event Stream Processing Studio を使用して、プロジェクトレベルの属性を設定できます。詳細については、“Update Project Properties” (*SAS Event Stream Processing: Using SAS Event Stream Processing Studio*)を参照してください。

SAS Event Stream Processing Studio のテストモードを使用して、要求を送信したり応答を受信できます。詳細については"Use the Scoring API When Testing a Model" (*SAS Event Stream Processing: Using SAS Event Stream Processing Studio*)を参照してください。

スコア要求は、1 つ以上の出力イベントを生成する 1 つ以上の入力イベントに依存しています。出力イベントが一定時間内に ESP サーバーに表示されない場合、要求はタイムアウトになり、500 応答コード(サーバーエラー)が送り返されます。次に例を示します:

```
$ curl -w "response code is %{http_code}\n" "http://myserver:2222/
eventStreamProcessing/v2/score" --data-binary @claim_2.xml
Score request timed out after 20 seconds.
response code is 500
```

タイムアウトのデフォルトは 20 秒ですが、 **esp-properties.yml** ファイルで設定できます。次に例を示します:

```
server:
   score_timeout: 30  # Timeout, in seconds, for Scoring API requests
```

要求

要求は、要求本文が次の 2 つの形式で可能な POST 要求です。JSON 配列または生データ。

最初の形式は JSON 配列で、そのオブジェクトはイベント値を表す name および value の子の配列です。

```
[
  {
    "inputs":
    [
      {"name":"patient","value":"Patient 1"},
      {"name":"statistic","value":"pulse"},
      {"name":"value","value":122},
      {"name":"_opcode_","value":"upsert"}
    ]
  }, {
    "inputs":
    [
      {"name":"patient","value":"Patient 1"},
      {"name":"statistic","value":"systolic_bp"},
      {"name":"value","value":110},
      {"name":"_opcode_","value":"upsert"}
    ]
  }
]
```

または、JSON オブジェクトには、イベントを表す名前と値を含めることができます。

```
[
```

```
  {
    "data": {
      "patient":"Patient 1",
      "statistic":"pulse",
      "value":33,
      "_opcode_":"upsert"
    }
  },{
    "data": {
      "patient":"Patient 1",
      "statistic":"systolic_bp",
      "value":110,
      "_opcode_":"upsert"
    }
  }
]
```

inputs または data オブジェクトの値は、スコアリングのためにイベントを受信したソースウィンドウにパブリッシュされる SAS Event Stream Processing イベントフィールドの名前と値です。

イベントの演算コードは_opcode_値で指定できます。デフォルトの演算コードは Insert です。

要求本文の 2 番目の形式は生データです。この手法は、project 要素の score-input-field 属性を使用して、イベントデータ内の単一のフィールド値を設定し、それをモデルに公開します。これは、大きくて不明確なデータ(画像など)や JSON や XML コードの大きなチャンクを扱う場合に役立ちます。このアプローチでは、Base64 エンコーディングなどの問題の可能性がある JSON オブジェクトをフォーマットする代わりに、要求で生データを送信できます。

この方法は、autogen-key 属性が設定されているソースウィンドウでのみ使用できます。次の例では、画像処理モデルには、識別子と画像を含むエントリポイントがあります:

```
<window-source index="pi_EMPTY" insert-only="true" autogen-key="true" name="w_data">
 <schema>
  <fields>
   <field key="true" name="id" type="int64"/>
   <field name="image" type="blob"/>
  </fields>
 </schema>
</window-source>
```

project 要素は次のようになります:

```
<project name="annotate" threads="8" score-input-window="w_data" score-input-
field="image" score-output-window="w_annotate">
```

次のように、画像をモデルに直接送信できます:

```
curl -v -X POST "http://myserver:2222/eventStreamProcessing/v2/score" --data-binary
@cars.jpg -H "content-type: application/octet-stream"
```

応答

要求本文と同様に、応答本文も 0 個以上のイベントを含む JSON 配列、または生データの 2 つの形式をとることができます。次の条件が満たされている場合、応答に生データを含めることができます。

- project 要素の score-output-field 属性は、フィールドを指定するために使用されました。
- score-output-window 属性が単一のイベントを収集しました。

これらの条件が満たされない場合、応答には、要求の説明でリストされている 2 つの形式のいずれかの要素を含む JSON 配列が含まれます。name および value オブジェクトの配列を使用した応答の例です:

```
[
  {
    "outputs":[
      { "name":"patient", "value":"Patient 1" },
      { "name":"statistic", "value":"pulse" },
      { "name":"value", "value":122.000000 },
      { "name":"lower", "value":50.000000 },
      { "name":"system", "value":"cardiovascular" },
      { "name":"upper", "value":100.000000 }
    ]
  }, {
    "outputs":[
      { "name":"patient", "value":"Patient 1" },
      { "name":"statistic", "value":"systolic_bp" },
      { "name":"value", "value":110.000000 },
      { "name":"lower", "value":60.000000 },
      { "name":"system", "value":"cardiovascular" },
      { "name":"upper", "value":80.000000 }
    ]
  }
]
```

単一の JSON オブジェクトを使用した応答の例を次に示します。

```
[
  {
    "data":{
      "patient":"Patient 1",
      "statistic":"pulse",
      "value":33.000000,
      "lower":50.000000,
      "system":"cardiovascular",
      "upper":100.000000
    }
  }, {
    "data":{
      "patient":"Patient 1",
```

```
            "statistic":"systolic_bp",
            "value":110.000000,
            "lower":60.000000,
            "system":"cardiovascular",
            "upper":80.000000
         }
      }
   ]
```


応答の JSON 形式は、要求本文の形式と一致します。要求本文が name-value 配列形式の場合、応答もその形式です。要求本文がオブジェクト形式(または生形式)の場合、応答は JSON オブジェクト形式です。

入力によってイベントが生成されなかった場合でも、要求は常に JSON 配列を返します。この場合、空の JSON 配列が返されます:

```
[]
```

例

クレーム処理の例

この例では、自動車事故のクレームが XML コードとして表されています。

```
<Claim>
  <ClaimID>214961</ClaimID>
  <Policy_No>12345</Policy_No>
  <paid_assessors>660.00</paid_assessors>
  <paid_extras>8822.80</paid_extras>
  <paid_investigators>0.00</paid_investigators>
  <claim_value>117777.00</claim_value>
  <Parties>
     <Party>
        <gender>male</gender>
        <NationalID>000001</NationalID>
        <role>PH driver</role>
        <vehicle>2000 Kia Spectra</vehicle>
        <vehicleReg>REG-1</vehicleReg>
        <ClaimID>214961</ClaimID>
        <injuries>
           <injury>CUTS</injury>
           <injury>BRUISES</injury>
        </injuries>
     </Party>
     <Party>
        <NationalID>000002</NationalID>
        <role>TP driver</role>
        <vehicle>1999 Peugeot 504</vehicle>
        <vehicleReg>REG-2</vehicleReg>
        <ClaimID>214961</ClaimID>
```

```
        <injuries>
          <injury>BROKEN LEG</injury>
        </injuries>
      </Party>
      <Party>
        <gender>male</gender>
        <NationalID>000003</NationalID>
        <role>TP passenger</role>
        <vehicle>1999 Peugeot 504</vehicle>
        <vehicleReg>REG-3</vehicleReg>
        <ClaimID>214961</ClaimID>
        <injuries>
          <injury>WHIPLASH</injury>
          <injury>SOFT TISSUE</injury>
        </injuries>
      </Party>
    </Parties>
    <Vehicles>
      <Vehicle>
        <vehicle_make_model>2000 Kia Spectra</vehicle_make_model>
        <vehiclereg>ABC0798</vehiclereg>
        <vehicle_value>88356.05</vehicle_value>
        <vehicle_damage>72467.15</vehicle_damage>
      </Vehicle>
      <Vehicle>
        <vehicle_make_model>1999 Peugeot 504</vehicle_make_model>
        <vehiclereg>XYZ5340</vehiclereg>
        <vehicle_value>43682.25</vehicle_value>
        <vehicle_damage>35826.95</vehicle_damage>
      </Vehicle>
    </Vehicles>
  </Claim>
```

スコア入力ウィンドウには、識別子のフィールドと、生のクレームデータを保持する文字列フィールドが含まれています。

```
<window-source name="claimdata" insert-only="true" autogen-key="true">
  <schema>
    <fields>
      <field name="id" type="int64" key="true"/>
      <field name="claimData" type="string"/>
    </fields>
  </schema>
</window-source>
```

スコア出力ウィンドウには、事故により発生した障害が表示されます。

```
<window-lua name="injuries" events="create">
  <schema copy="parties">
    <fields>
      <field name="injury" type="string"/>
    </fields>
  </schema>
  <code>...</code>
</window-lua>
```

ソース ウィンドウの claimData フィールドは生の XML コードを受信するため、project 要素は次のようになります:

```
<project name="claims" threads="4" score-input-window="claimdata"
    score-input-field="claimData" score-output-window="injuries">
```

この構成では、次の要求によって、クレーム XML コードがモデルにパブリッシュされます。XML コードは、 **claim.xml** というファイルに格納されていると想定されます。

```
$ curl -X POST "http://myserver:2222/eventStreamProcessing/v2/score" --data-binary
@claim.xml
```

応答には、このクレームによって引き起こされたすべての事故が含まれています。

```
[
  {
    "data":{
      "id":"27",
      "claim_id":"214961",
      "vehicle":"2000 Kia Spectra",
      "registration":"REG-1",
      "national_id":"000001",
      "injury":"CUTS"
    }
  },
  {
    "data":{
      "id":"28",
      "claim_id":"214961",
      "vehicle":"2000 Kia Spectra",
      "registration":"REG-1",
      "national_id":"000001",
      "injury":"BRUISES"
    }
  },
  {
    "data":{
      "id":"30",
      "claim_id":"214961",
      "vehicle":"1999 Peugeot 504",
      "registration":"REG-2",
      "national_id":"000002",
      "injury":"BROKEN LEG"
    }
  },
  {
    "data":{
      "id":"32",
      "claim_id":"214961",
      "vehicle":"1999 Peugeot 504",
      "registration":"REG-3",
      "national_id":"000003",
      "injury":"WHIPLASH"
    }
  },
  {
    "data":{
      "id":"33",
      "claim_id":"214961",
      "vehicle":"1999 Peugeot 504",
```

```
        "registration":"REG-3",
        "national_id":"000003",
        "injury":"SOFT TISSUE"
      }
    }
]
```

画像注釈の例

この例では、画像処理の効率を向上させるために生の入力と出力が使用されます。入力ソースウィンドウに、ID とイメージがあります。

```
<window-source name="w_data" index="pi_EMPTY" insert-only="true" autogen-key="true">
 <schema>
  <fields>
   <field key="true" name="id" type="int64"/>
   <field name="image" type="blob"/>
  </fields>
 </schema>
</window-source
```

出力スコアウィンドウは、注釈付き入力画像のコピーを含む annotated_image フィールドを持つ Python ウィンドウです。

```
<window-python name="w_annotate" events="annotate_event">
 <schema>
  <fields>
   <field key="true" name="id" type="int64"/>
   <field name="annotated_image" type="blob"/>
  </fields>
 </schema>
 <code>...</code>
</window-python>
```

プロジェクト要素は、次のウィンドウとフィールドを指します。

```
<project index="pi_EMPTY" name="computer_vision_with_onnx" threads="8"
   score-input-window="w_data" score-input-field="image"
   score-output-window="w_annotate" score-output-field="annotated_image">
```

たとえば、空を飛ぶペリカンの写真をアップロードすることができます。

次のコマンドを使用して、要求本文にバイナリデータが含まれ、クライアントがバイナリデータの受信を要求していることを ESP サーバーに指定します。

```
curl -X POST "http://myserver:2222/eventStreamProcessing/v2/score"
  --data-binary @pelicans.jpg -H "content-type: application/octet-stream" 1
  -H "accept: application/octet-stream" -o annotated.jpg 2
```

1 content-type ヘッダーは、バイナリコンテンツがクライアントから送信されていることをサーバーに示します。

2 accept ヘッダーは、出力画像をバイナリデータとして返すようにサーバーに指示します。

curl コマンドは、注釈が適用された画像を含む annotated.jpg という名前のローカルファイルにバイナリ応答を書き込みます。

23

モデルコンテキストプロトコルの使用

使用目的と使用制限

使用目的

SAS Event Stream Processing モデルコンテキストプロトコル(MCP)統合は、カスタマーが ESP プロジェクト機能を MCP サーバーとしてパブリッシュし、大規模言語モデル(LLM)が実行中の ESP プロジェクトと相互作用できるようにすることを目的としています。この統合により、LLM 駆動型アプリケーションが ESP プロジェクトロジックをクエリまたは実行するためのインターフェイスが提供されます。顧客は、独自の LLM の選択と構成、および関連するライセンス、品質管理、セキュリティ、およびコンテンツの保護策を管理する責任があります。顧客は、MCP 統合の使

用が適用される法律、データプライバシー要件、およびカスタマーの内部ガバナンスポリシーに準拠していることを確認する責任を持ちます。

MCP 機能は、イベントストリーム処理の概念と MCP 統合の利点とリスクを理解する担当者を使用することを想定しています。MCP は、SAS Event Stream Processing ですでに利用可能な機能を超える新しい機能を導入しません。アクセスパターンを従来の API から MCP ベースのインタラクションに変更します。MCP を介してこれらの機能にアクセスしても、ESP プロジェクト自体が提供する内容を超えて、追加の保護レイル、入力検証、またはセキュリティコントロールが自動的に追加されることはありません。たとえば、ESP プロジェクトが作成、読み取り、更新、削除(CRUD)などの操作をサポートしている場合、これらの操作も MCP を介してアクセスできます。特にセキュリティに関する設定とベストプラクティスに関する詳細なガイダンスについては、公式の MCP ドキュメントを参照する必要があります。

使用制限

LLM を含むジェネレーター AI モデルは、統計的予測を使用し、誤ったまたは予期しない出力や問題のある出力を生成することができます。カスタマーが MCP と選択した LLM を統合する場合、これらの LLM は不完全または不正確な応答を生成する可能性があります。データ整合性、データプライバシー、セキュリティなど、MCP 統合の利点とリスクを最初に理解せずに、MCP 統合を使用するようにしてください。顧客は、すべての出力を検証する前に検証し、特定のユースケースに適切なレビュープロセスを実装する必要があります。

モデルコンテキストプロトコルの概要

モデルコンテキストプロトコル(MCP)は、大規模言語モデル(LLM)と外部リソース(API やデータベースなど)の間のインターフェイスを提供するプロトコルです。これは、アプリケーションが言語モデルとコンテキスト情報を共有し、ツールを人工知能(AI)システムにパブリッシュし、適応可能なワークフローを構築するための標準化された方法を提供します。

モデルコンテキストプロトコルは、ローカルまたはリモートトランスポートメカニズムを使用して実装できます。SAS Event Stream Processing サーバーは、ストリーミング可能な HTTP トランスポートメカニズムを使用して、MCP リソースへのリモートアクセスを可能にします。/eventStreamProcessing/v2/mcp のようなエンドポイントは、MCP クライアントが ESP サーバーとの HTTP 接続を確立できるようにします。接続が確立されると、接続を介して JSON メッセージが送受信されます。永続的なサーバー送信イベント(SSE)接続も確立されます。SSE 接続は、サーバーがデータを送信できるサーバーからクライアントへの一方向の接続です。詳細については、ストリーミング可能な HTTP を参照してください。

MCP 構成を含む SAS Event Stream Processing プロジェクトで、プロジェクトを実行すると MCP サーバーがパブリッシュされます。これで、プロジェクトは LLM が連絡できるリソースになりました。

モデルコンテキストプロトコルには、プロンプト、リソース、ツールの 3 つの主要な要素があります。プロンプトは開始点として機能し、意図とインタラクションを定義します。リソースはコンテキストまたはデータを提供します。モデルがタスクを認識すると、ツールを使用してアクションが実行されます。詳細については、MCP ドキュメントを参照してください。SAS Event Stream Processing では、MCP ツールを使用して、プロジェクトと LLM の相互作用を構成します。MCP リソースはこのリリースではサポートされていません。

SAS Event Stream Processing モデルの mcp 要素の構成は、project 要素の下にある単一の mcp の内で指定されます。構成された mcp 要素の例を示します。

```
<project name="myproject" index="pi_EMPTY" threads="4">1
 <mcp>
  <prompts>
     ...
  </prompts>

  <tools>
     ...
  </tools>
  <code snippet="..."><![CDATA[2
     Lua or Python Code
  ]]></code>
 </mcp>
 ...
</project>
```

1 myproject というプロジェクトを作成します。

2 code エレメントはオプションです。使用される場合、MCP ツールをサポートするために使用される Lua または Python コードが含まれます。詳細については、*"code" (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*を参照してください。

SAS Event Stream Processing Studio を使用し、MCP に関連する XML 要素を構成できます。詳細については、プロジェクトを MCP サーバーとして公開するを参照してください。

プロンプトの使用

*プロンプト*により、ESP サーバーは言語モデルとの対話方法に関するメッセージまたは指示を提供します。これらは ESP サーバーからクライアントに送信されて、ユーザーに表示されます。プロンプトはユーザーによって 制御され、ESP サーバーの機能を呼び出すためのクライアントへのガイドとして機能します。

構成されたプロンプトの例を次に示します。

```
<prompt name="...">
  <title>...</title>
  <description>...</description>
  <text>...</text>
  <fields>
```

```
      <field name="...">
        <description>...</description>
      </field>
    </fields>
  </prompt>
```

prompt 要素には、名前属性、タイトル、説明、テキスト、およびフィールドの各要素が含まれます。属性と要素の説明は次のとおりです。

- Name: この属性はプロンプトの名前であり、モデルに固有である必要があります。
- Title: この要素は、プロンプトのタイトルを指定します。
- Description: この要素は、プロンプトの動作の説明を指定します。
- Text: この要素には、ダイアログウィンドウを通じて外部クライアントによって表示されるテキストが含まれています。先頭に$文字が付いたテキストは変数として扱われ、ユーザーに変数の値を入力するよう促します。
- Fields: この要素には、text 要素で作成された変数を参照するフィールド要素が含まれています。フィールド要素には、名前属性と、その中にネストされた description 要素があります。text 要素に変数が作成されていない場合は、フィールドを定義する必要はありません。

詳細については、“prompts” (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

ツールの使用

ツールは、入力を受け取り、サーバー上でタスクを実行し、結果をクライアントに送り返す要素です。プロンプトとは異なり、ツールはモデルによって制御されます。これは、言語モデルがそのコンテキスト理解とユーザーのプロンプトに基づいてツールを自動的に検出して呼び出すことができることを意味します。各ツールには、種類、一意の名前、タイトル、および説明が必要です。

ESP サーバーは、MCP クライアントにサービスを提供できる次のツールを提供します。

- スコアツール: このツールを使用すると、クライアントはイベントデータをソースウィンドウに渡し、モデルを通過するデータから結果を取得できます。結果は指定された出力ウィンドウからイベントデータの形式で返されます。スコアツールは、input、output、および functions 要素で構成されています。input 要素には、入力データに関する情報と、output 要素には、スコアツールが生成するデータに関する情報が含まれます。functions 要素には、スコアツールの実行中に実行されるオプションの関数が含まれます。構成されたスコアツールの例については、“スコアツールの例”を参照してください。
- クエリツール: このツールを使用すると、クライアントはウィンドウのコンテンツから特定の情報を要求できます。指定された出力ウィンドウのコンテンツを LLM に直接返します。クエリツールは fields 要素と filter 要素を含む output 要素を使用して構成されます。fields 要素内の field 要素には出力フィールド名がリストされ、filter 要素は、出力イベントに適用できるオプションのフィルターです。

構成されたクエリツールの例については、“クエリツールの例”を参照してください。

- サブスクライブツール: このツールを使用すると、クライアントはモデル内のウィンドウをサブスクライブし、イベントデータの形式で通知を受信できます。クエリツールとサブスクライブツールは似ており、同じ方法で構成されます。ただし、重要な違いは、サブスクライブツールがイベントを要求元の LLM に直接返さないことです。構成されたクエリツールの例については、“サブスクライブツールの例”を参照してください。
- ツールのパブリッシュ: このツールを使用すると、クライアントはデータを Event Stream Processing モデルにパブリッシュできます。パブリッシュツールは、field、data、url、および blocksize 要素を含む input 使用して構成されます。input 要素には受信データに関する情報が含まれ、そのウィンドウ属性はイベントをパブリッシュするソースウィンドウを指定します。構成されたクパブリッシュツールの例については、“パブリッシュツールの例”を参照してください。
- カスタムツール: このツールを使用すると、モデルのデザインは、入力を受け入れてクライアントに返される結果を生成するカスタム Lua コードまたは Python コードを記述できます。カスタムツールは、output および functions 要素を使用して構成されます。output 要素には、カスタムツールが生成するデータに関する情報が含まれ、functions 要素は呼び出す関数を指定します。ESP サーバーは関数を呼び出し、適切な応答を作成してクライアントに返します。構成されたカスタムツールの例については、“カスタムツールの例”を参照してください。

さまざまなツールの種類の要素と属性の詳細については、“モデルコンテキストプロトコルに関連する XML 言語要素” (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

MCP ツールの例

スコアツールの例

構成されたスコアリッシュツールの例を次に示します。

```
<tool type="score" name="..."> 1
  <title>...</title>
  <description>...</description>
  <config>
   <input window="..." array="..."> 2
    <fields>
     <field name="..." mime-type="...">
      <description>...</description>
     </field>
     ...
    </fields>
   </input>
   <output window="..." array="..."> 3
    <fields>
     <field name="..." mime-type="...">
      <description>...</description>
```

```
      </field>
      ...
     </fields>
    </output>
    <functions> 4
     <function name="pre" code="..." />
     <function name="post" code="..." />
     <function name="events" code="..." />
      <use>...</use>
     </function>
    </functions>
   </config>
  </tool>
```

1 ツールタイプは score に設定され、ツールには一意の名前が付けられています。

2 入力データに関する情報が指定されています。これには、データを受け取るソースウィンドウの名前、およびツールがオブジェクトの配列を受け取るかどうかが含まれます。

3 出力データに関する情報が指定されます。これには、イベントをモニターする下位ウィンドウの名前と、ツールがオブジェクトの配列を返す場合の出力配列の名前が含まれます。

4 スコアツールの実行中に実行される関数はすべてここで呼び出されます。name の値によって、関数がいつ実行されるか決定します。

スコアツールの要素と属性の詳細については、*"tools" (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*を参照してください。

クエリツールの例

構成されたクエリツールの例を次に示します。

```
<tool type="query" name=""> 1
  <title>...</title>
  <description>...</description>
  <config>
   <output window="..."> 2
    <fields>
     <field name="..." type="...">
      <description>...</description>
     </field>
     ...
    </fields>
    <filter function="..."> 3
     <fields>
      <field name="..." type="..." optional="..." array="...">
       <description>...</description>
      </field>
      ...
      </field>
     </fields>
    </filter>
   </output>
  </config>
```

```
</tool>
```

1 ツールタイプは query に設定され、ツールには一意の名前が付けられています。

2 クエリツールが生成するデータに関する情報が指定されます。これには、ツールがイベントを読み込むウィンドウが含まれます。

3 出力イベントをフィルタリングするために、オプションのフィルターが定義されます。イベントをフィルタリングする Lua または Python 関数を指定する必要があります。

クエリツールの要素と属性の詳細については、*"tools" (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*を参照してください。

サブスクライブツールの例

構成されたサブスクライブツールの例を次に示します。これは、type 属性を除いて、構成されたクエリツールと同じです。

```
<tool type="query" name=""> 1
  <title>...</title>
  <description>...</description>
  <config>
   <output window="..."> 2
    <fields>
     <field name="..." type="...">
      <description>...</description>
     </field>
     ...
    </fields>
    <filter function="..."> 3
     <fields>
      <field name="..." type="..." optional="..." array="...">
       <description>...</description>
      </field>
      ...
      </field>
     </fields>
    </filter>
   </output>
  </config>
 </tool>
```

1 ツールタイプは subscribe に設定され、ツールには一意の名前が付けられています。

2 サブスクライブツールが生成するデータに関する情報が指定されます。これには、ツールがイベントを読み込むウィンドウが含まれます。

3 出力イベントをフィルタリングするために、オプションのフィルターが定義されます。イベントをフィルタリングする Lua または Python 関数を指定する必要があります。

サブスクライブツールの要素と属性の詳細については、*"tools" (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*を参照してください。

パブリッシュツールの例

構成されたパブリッシュツールの例を次に示します。

```
<tool type="publish" name="..."> 1
  <title>...</title>
  <description>...</description>
  <config>
   <input window="..."> 2
    <field>...</field>
    <data>...</data> 3
    <url>...</url> 4
    <blocksize>...</blocksize> 5
   </input>
  </config>
 </tool>
```

1 ツールタイプは publish に設定され、ツールには一意の名前が付けられています。

2 パブリッシュツールが生成するデータに関する情報が指定されます。これには、入力データを受け取るソースウィンドウの名前が含まれます。

3 data 要素はオプションで、パブリッシュするデータのコンテンツが含まれます。data 要素を定義する場合、url 要素をを定義する必要はありません。

4 url 要素はオプションで、パブリッシュするデータのコンテンツの URL を含みます。url 要素を定義する場合、data 要素を定義する必要はありません。

5 blocksize 要素はオプションで、データのパブリッシュ時に各イベントブロックに含めるイベントの数を指定します。

パブリッシュツールの要素と属性の詳細については、*“tools” (SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス)*を参照してください。

カスタムツールの例

構成されたカスタムツールの例を次に示します。

```
<tool type="custom" name="getProcInfo"> 1
    <title>Get Process Information</title>
    <description>Get process information including CPU and memory usage information.</
description>
    <config>
     <output>
      <fields>
       <field name="path" type="string" /> 2
       <field name="cmdline" type="string" />
       <field name="created" type="string" />
       <field name="user" type="string" />
       <field name="cwd" type="string" />
       <field name="status" type="string" />
       <field name="cpu" type="number" />
```

```
        <field name="num_threads" type="number" />
        <field name="memory" type="object"> 3
         <fields>
          <field name="resident" type="number"/>
          <field name="virtual" type="number"/>
          <field name="shared" type="number"/>
          <field name="text" type="number"/>
          <field name="data" type="number"/>
         </fields>
        </field>
       </fields>
      </output>
      <functions>
       <function name="call" code="getProcInfo"/> 4
      </functions>
     </config>
    </tool>
```

1 ツールタイプは custom に設定され、ツールには一意の名前が付けられています。

2 定義された各 field 要素は、出力 JSON ファイル内のフィールドに関連付けられます。フィールドには一意の名前が必要です。

3 フィールドの type 属性が object に設定されている場合、field 要素内にフィールドをネストできます。

4 カスタムツールの実行中に呼び出されると、getProcInfo という関数が実行されます。

getProcInfo 関数は別のコード要素で定義されます。

```
<code><![CDATA[

import datetime
import psutil 1

def getProcInfo(parms):
   data = {}
   p = psutil.Process()
   data["path"] = p.exe()
   data["cmdline"] = p.cmdline()
   data["created"] = datetime.datetime.fromtimestamp(p.create_time()).strftime("%Y-%m-
%d %H:%M:%S")
   data["user"] = p.username()
   data["cwd"] = p.cwd()
   data["status"] = p.status()
   data["cpu"] = p.cpu_percent(interval=.5)
   data["num_threads"] = p.num_threads()
   tmp = p.memory_info()
   data["memory"] =
{"resident":tmp[0],"virtual":tmp[1],"shared":tmp[2],"text":tmp[3],"data":tmp[5]}
   print(str(data))
   return(data)

 ]]></code>
```

1 カスタムツールは Pythonpsutil パッケージを使用し、情報を取得してクライアントに返します。

カスタムツールの要素と属性の詳細については、“tools” (*SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*)を参照してください。

追加の例

この例は、整数の配列を取得し、それらを加算するスコアツールを使用した完全なプロジェクトを示しています。結果はクライアントに送り返されます。

```
<project name="p" pubsub="auto" threads="4">
 <mcp>
  <prompts>
   <prompt name="add_3_numbers"> 1
    <title>Add 3 Numbers</title>
    <description>Use SAS Event Stream Processing to add an arbitrary number of integers
and return the result.</description>
    <text>Can you add these numbers $num1, $num2, and $num3 and return the sum?</
text> 2
    <fields> 3
     <field name="num1">
      <description>The first number</description>
     </field>
     <field name="num2">
      <description>The second number</description>
     </field>
     <field name="num3">
      <description>The third number</description>
     </field>
    </fields>
   </prompt>
  </prompts>
  <tools>
   <tool type="score" name="add_numbers"> 4
    <title>Add Numbers</title>
    <description>Use SAS Event Stream Processing to add an arbitrary number of integers
and return the result.</description>
    <config>
     <score-functions>
      <events function="handleEvent" />
     </score-functions>
     <input window="s"> 5
      <fields>
       <field name="numbers" type="number" array="true">
        <description>The numbers to add</description>
       </field>
      </fields>
     </input>
     <output window="lua"> 6
      <fields>
       <field name="sum">
        <description>the sum</description>
```

```
          </field>
         </fields>
        </output>
       </config>
      </tool>
     </tools>
     <code><![CDATA[ 7
def handleEvent(data):
 response = "The sum of "
 i = 0
 for value in data["numbers"]:
  if i == len(data["numbers"]) - 1:
   response += " and "
  elif i > 0:
   response += ", "
  response += str(value)
  i += 1
 response += " is " + str(data["sum"])
 return response
  ]]></code>
 </mcp>
 <contqueries>
  <contquery name="cq"> 8
   <windows>
    <window-source name="s" autogen-key="true" insert-only="true"> 9
     <schema>
      <fields>
       <field name="id" type="int64" key="true"/>
       <field name="numbers" type="array(i32)"/>
      </fields>
     </schema>
    </window-source>
    <window-lua name="lua" events="create"> 10
     <schema>
      <fields>
       <field name="id" type="int64" key="true"/>
       <field name="numbers" type="array(i32)"/>
       <field name="sum" type="int32"/>
      </fields>
     </schema>
     <code><![CDATA[
      local   id = 1

      function create(data)
         local event = {}
         local sum = 0

         for _,value in ipairs(data.numbers)
         do
           sum = sum + value
         end

         event.id = id
         event.numbers = data.numbers
         event.sum = sum
```

```
            id = id + 1

            return event
          end
        ]]></code>
      </window-lua>
    </windows>
    <edges>
      <edge source="s" target="lua"/> 11
    </edges>
  </contquery>
 </contqueries>
</project>
```

1 add_3_numbers というプロンプトを作成します。

2 ユーザーに表示するテキストを指定し、変数 num1、num2、および num3 を作成します。

3 ユーザーに変数の値を入力するように求めるフィールドを作成します。

4 add_numbers というスコアツールを作成します。

5 入力データの構成を指定します。

6 出力データの構成を指定します。

7 handleEvent 関数を定義するコードが格納されています。

8 cq という連続クエリを作成します。

9 スコア入力ソースウィンドウを定義します。

10 出力ウィンドウを定義します。

11 先頭に s、および末尾に lua を持つエッジを作成します。

Using Apps

Overview

Apps enable MCP servers to provide interactive user interfaces to MCP clients such as GitHub Copilot and Claude Desktop. For more information about the specifications for MCP apps, see apps.mdx.

These apps enable users to view and interact with data that is processed within an ESP model. By connecting data sources to visual representations, MCP apps allow real-time information to be presented in a dynamic way.

An app definition consists of a name, description, and a configuration that defines the event sources (ESP windows) from which to receive data. The definition also includes visual components to display the data. Every app has a

configuration element that defines its characteristics. The content of the element can be embedded in the XML file, or it can exist in an external URL. If the configuration is stored in an external URL, you can modify the app configuration and run the updated app without restarting the ESP server. For more information about the element, see "モデルコンテキストプロトコルに関連する XML 言語要素" in *SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*.

Event Sources

Event sources define where an app retrieves its data within an ESP model. They are configured using elements, where each event source specifies a name and the ESP window from which to obtain data. Each event source is either a collection or a stream. A *collection* represents the full set of events currently in the window, and a *stream* delivers events as they are created.

Event sources include an interval that controls how frequently data is delivered, and they can include a filter to limit the events returned based on specified criteria. Here is an example of an app that uses four different event sources:

```
<event-sources>
   <event-source name="shipdata" type="collection" window="Boat_Current"
interval="250"/>
   <event-source name="speeding" type="stream" window="Last_Speeding_Event"
interval="250"/>
   <event-source name="exclusion" type="stream" window="Last_Exclusion_Violation"
interval="250"/>
   <event-source name="sourcewin" type="stream" window="Boat" interval="250"/>
</event-sources>
```

Visual Components

Visual components enable you to create visual representations of event data within an app. They are used to display data from defined sources in formats such as charts, tables, or maps.

Each visual component is associated with a data source and uses configuration settings to control how the data is displayed and how the visualization behaves. Apps can include one or more visual components, allowing data to be presented across multiple views within the same app. Visual components support interactive features:

- "Table"
- "Bar Chart"
- "Line Chart"
- "Time Series Chart"
- "Compass"

- “Gauge”
- “Pie Chart”
- “Image Viewer”
- “Geo Map”
- “Model Viewer”
- “Log Viewer”
- “Metrics Viewer”
- “Custom Component”

Examples of Visual Components

Table

The table component enables you to display data in a tabular form. It supports the following property values:

- title: The title of the component.
- values: The values to be displayed in the table. The default behavior is to display all values.
- image_width: The width (in pixels or a percentage) of any images in the table.
- image_height: The height (in pixels or a percentage) of any images in the table.

Here is an example of a configured table:

```
<component source="shipdata" type="table">1
  <properties>
    <property name="title">Current Ship Data</property>2
    <property name="values">boat_id,heading,speed,latitude,longitude</property>3
  </properties>
</component>
```

1 Creates a table component that uses data from shipdata.

2 Defines the title of the component as Current Ship Data.

3 Defines the values to be displayed in the table. The specified values appear as the column headings.

Bar Chart

The bar chart component enables you to display data as a vertical or horizontal bar chart. It supports the following property values:

- title: The title of the component.

- y: Numeric event values to plot on the chart. The values determine the length of the bars.
- orientation: The orientation of the chart. Valid values are **horizontal** and **vertical**.
- calculate_range: A Boolean that indicates whether the chart automatically sets the axis minimum and maximum values using the data values. The default value is **false**.
- range: The range of possible values written as a comma-separated list of two values.

Here is an example of a configured bar chart:

```
<component source="shipdata" type="bar"> 1
  <properties>
    <property name="title">Current Ship Speed</property>
    <property name="y">speed</property> 2
    <property name="orientation">horizontal</property>
    <property name="range">0,8</property> 3
  </properties>
</component>
```

1 Creates a bar chart component that uses data from shipdata.

2 Determines the bar chart to display each ship's speed.

3 Defines the range of values from zero to eight.

Line Chart

The line chart component enables you to display one or more numeric values as points connected by lines to show variations in data. It supports the following property values:

- title: The title of the component.
- y: Numeric event values to plot on the chart. The values are plotted as points connected by lines.
- orientation: The orientation of the chart. Valid values are **horizontal** and **vertical**.
- markersize: The size of the markers in the chart. The default value is **0**.
- line_width: The width of the line that connects the points. The default value is **4**.
- curved: A Boolean that indicates whether to draw curved lines. The default value is **false**.
- calculate_range: A Boolean that indicates whether the chart automatically sets the axis minimum and maximum values using the data values. The default value is **false**.
- range: The range of possible values written as a comma-separated list of two values.

Here is an example of a configured line chart:

```
<component source="shipdata" type="line"> 1
  <properties>
   <property name="title">Current Ship Speed</property>
   <property name="y">speed</property>
   <property name="markersize">20</property> 2
   <property name="line_width">10</property> 3
   <property name="range">0,8</property>
  </properties>
</component>
```

1 Creates a line chart component that uses data from shipdata.

2 Sets the marker size to 20.

3 Sets the line width to 10.

Time Series Chart

The time series chart component enables you to display data over time. You must have a date or time field in the data to configure this component. It supports the following property values:

- title: The title of the component.
- time: The date or time field to use for the X axis of the chart.
- (Optional) series: A field used to group the data into separate lines. If not specified, all data points are displayed as a single line.
- y: The name of the field that contains the value to plot.
- markersize: The size of the markers in the chart. The default value is **0**.
- line_width: The width of the line that connects the points. The default value is **4**.
- curved: A Boolean that indicates whether to draw curved lines. The default value is **false**.
- calculate_range: A Boolean that indicates whether the chart automatically sets the axis minimum and maximum values using the data values. The default value is **false**.
- range: The range of possible values written as a comma-separated list of two values.
- max: The maximum number of data points to be displayed for each series. The default value is **50**.

Here is an example of a configured time series chart:

```
<component source="shipdatastream" type="timeseries"> 1
  <properties>
   <property name="title">Ship Speed Over Time</property>
   <property name="series">boat_id</property> 2
   <property name="y">speed</property>
   <property name="time">@timestamp</property> 3
   <property name="range">0,8</property>
  </properties>
</component>
```

1 Creates a time series chart component that uses data from shipdatastream.

2 Specifies to group the data by boat_id and to create separate lines for each ID.

3 Specifies that the timestamp field is the X axis of the chart.

Compass

The compass component enables you to display data with directional information. The compass displays each event by key and uses a specified value field to show direction. It supports the following property values:

- title: The title of the component.
- value: The field that contains the compass heading values and the direction of the needle.
- (Optional) size : The value that specifies the diameter of each compass. The default value is **250**.

Here is an example of a configured compass:

```
<component source="shipdata" type="compass"> 1
  <properties>
   <property name="title">Current Ship Heading</property>
   <property name="value">heading</property> 2
  </properties>
</component>
```

1 Creates a compass component using data from shipdata.

2 Specifies heading as the compass heading values.

Gauge

The gauge component enables you to display a single numeric event field. Create multiple gauges to compare numeric events fields side by side. The component supports the following property values:

- title: The title of the component.
- value: The name of the field that contains the value to be displayed in the gauge.
- range: The range of the gauge written as a comma-separated list of two values. The default range is **0,10**.
- (Optional) size: The size of each gauge. The default value is **250**.

Here is an example of a configured gauge:

```
<component source="shipdata" type="gauge"> 1
  <properties>
   <property name="title">Current Ship Speed</property>
   <property name="value">speed</property> 2
  </properties>
</component>
```

1 Creates a gauge component that uses data from shipdata.

2 Specifies the speed field as the value to be displayed.

Pie Chart

The pie chart component enables you to display numeric values as slices that represent portions of a whole. It supports the following property values:

- title: The title of the component.
- value: The value to be displayed in the pie chart.

Here is an example of a configured pie chart:

```
<component source="violations" type="pie"> 1
  <properties>
    <property name="title">Speeding Violations</property>
    <property name="value">count</property> 2
  </properties>
</component>
```

1 Creates a pie chart component that uses data from violations.

2 Specifies count as the value to be displayed in the pie chart.

Image Viewer

The image viewer component enables you to display event data that contains images. It supports the following property values:

- title: The title of the component.
- image: The name of the image field to be displayed.
- scale: A floating-point value that is used to scale the image.

Here is an example of a configured image viewer:

```
<component type="imageviewer" source="annotate"> 1
  <properties>
    <property name="title">Images</property>
    <property name="image">image</property> 2
    <property name="scale">.35</property> 3
  </properties>
</component>
```

1 Creates an image viewer component that uses data from imageviewer.

2 Specifies image as the field to be displayed.

3 Specifies to scale the image to 35% of its original size.

Geo Map

The geo map component enables you to display event data that contains geographic information, and it uses LeafletJS to display the information. It supports the following property values:

- title: The title of the component.

- lat: The name of the event field that is used for the latitude value.
- lon: The name of the event field that is used for the longitude value.
- (Optional) center: The geographic center of the map in the format lat:<latitude>,lon:<longitude>.
- (Optional) zoom: The zoom level of the map. A low zoom level shows entire continents, and a high zoom level shows the details of a city.
- (Optional) clicktips: A Boolean value that indicates whether to use persistent tooltips. Tooltips appear when a data item is selected.
- (Optional) delegate: The JavaScript object that contains functions used to control the geo map display. The delegate object contains the init, properties, html, and tooltip functions.
  - init(component): Initializes the geo map.
  - properties(map,item): Sets the display properties.
  - html(map,item): Returns HTML to display as the map marker.
  - tooltip(map,item): Returns the tooltip of an item.

Here is an example of a configured geo map:

```
<component source="shipdata" type="map"> 1
  <properties>
   <property name="title">Ship Map</property>
   <property name="delegate">delegate</property> 2
   <property name="lat">latitude</property>
   <property name="lon">longitude</property>
   <property name="clicktips">true</property> 3
   <property name="center">lat:56.000241,lon:-3.424250</property>
  </properties>
  <css><![CDATA[
  ...
  ]]></css>
  <javascript><![CDATA[

  var delegate = {
  ...
  }
  ]]></javascript>
</component>
```

1 Creates a geo map component that uses data from shipdata.

2 Declares the delegate Javascript object.

3 Specifies that the geo map uses persistent tooltips.

Model Viewer

The model viewer component enables you to view the windows and edges that make up the model in the server. Use the toolbar to zoom in, zoom out, or reset the model to its original perspective. Hovering over a window enables you to see information about it (for example, type, index, and class). When you select a window, you can view the current event contents, event stream, or XML

definition of the window. The component supports the following property value:

- title: The title of the component.

Here is an example of a configured model viewer:

```
<component type="modelviewer">
 <properties>
  <property name="title">Ships Model Viewer</property>
 </properties>
</component>
```

Log Viewer

The log viewer component enables you to display server log information and provides controls for managing logging contexts and clearing the log. Use the toolbar to show server logging or to clear the log. It supports the following property value:

- title: The title of the component.

Here is an example of a configured log viewer:

```
<component type="logviewer">
 <properties>
  <property name="title">Ships Log Viewer</property>
 </properties>
</component>
```

Metrics Viewer

The metrics viewer component enables you to view the ESP metrics that exist in a server. Each metric is displayed as a graphic. Use the toolbar to view the metric window that contains the type, name, and current state of all of the metrics that exist in the server. It supports the following property value:

- title: The title of the component.

Here is an example of a configured metrics viewer:

```
<config>
      <component type="metricviewer">
       <properties>
        <property name="title">Metric Viewer</property>
       </properties>
      </component>
    </config>
```

Custom Component

The custom component enables you to create custom MCP components. These components are written using ESP utilities or other technologies (for example, HTML or JavaScript). It supports the following property values:

- html: Contains the HTML to be included with the component.
- html-url: Contains the external HTML to be included with the component.

- delegate: The JavaScript object that contains functions used to control the custom component. The delegate object contains the init function that is called when the component is initialized. This function does all of the required setup for the component.

Note: If you require external resources for the custom component, you must add the <resource-domains> and <connect-domains> elements. For more information, see "リソースドメイン" in *SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*.

Here is an example of a configured custom component:

```
<component type="custom"> 1
    <properties>
     <property name="title">My Custom Component</property>
     <property name="html"> <![CDATA[ 2
     <div style='width:100%;height:100%;display:flex;justify-content:center;align-
items:center;font-size:2rem'>
      This is my simple custom component.
     </div>
     ]]></property>
    </properties>
  </component>
```

1 Creates a custom component.

2 Specifies the HTML code to be included in the component.

App Types and Testing

Visual App

A Visual app displays a single visual component and is configured with any event sources that you want to include. Here is an example of a configured Visual app:

```
<app name="Ships_Heading" type="visual"> 1
 <title>Compasses Showing Ship Heading</title>
 <description>Compasses Showing Ship Heading</description>
 <config>
  <event-sources>
   <event-source name="shipdata" type="collection" window="Boat_Current"
interval="250"/> 2
  </event-sources>
  <component source="shipdata" type="compass"> 3
   <properties>
    <property name="title">Current Heading</property>
    <property name="value">heading</property>
   </properties>
  </component>
```

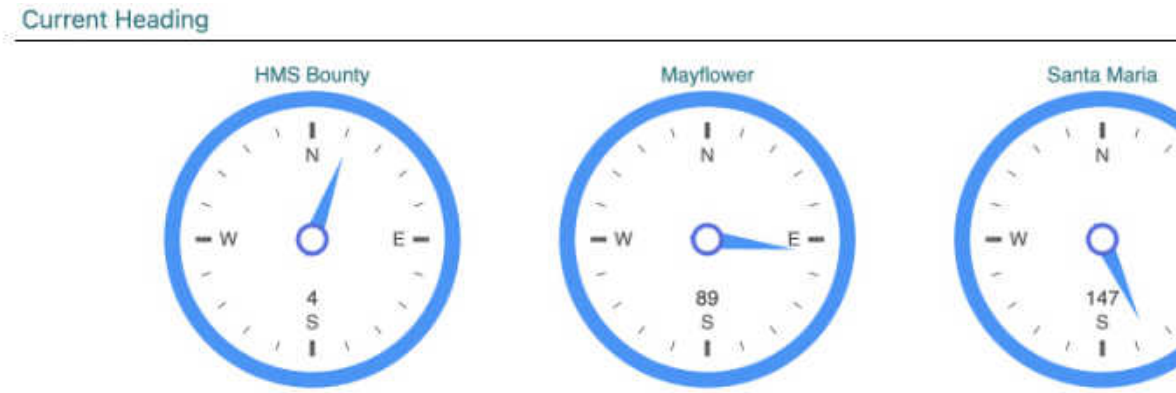
```
 </config>
</app>
```

1 Creates a Visual app called Ships_Heading.

2 Specifies the event-source element called shipdata, which delivers data every 250 milliseconds.

3 Creates a compass component based on the data from shipdata.

The code renders the following Visual app:



Dashboard App

The Dashboard app displays multiple components and is configured with any event sources that you want to include. Dashboard apps are useful when you want to display related components side by side. You can specify the dimensions for each component to determine the layout of the page. Here is an example of a configured Dashboard app with three tables:

```
<app name="Ships_DataDashboard" type="dashboard">1
 <title>Dashboard App for Ship Data.</title>
 <description>Dashboard App for Ship Data.</description>
 <config>
  <event-sources>2
   <event-source name="shipdata" type="collection" window="Boat_Current"
interval="250"/>
   <event-source name="speeding" type="stream" window="Last_Speeding_Event"
interval="250"/>
   <event-source name="exclusion" type="stream" window="Last_Exclusion_Violation"
interval="250"/>
  </event-sources>
  <components>3
   <component name="boat_table" type="table" source="shipdata" width="40%"
height="40%">
    <properties>
     <property name="values">boat_id,heading,speed,latitude,longitude</property>
     <property name="title">Ship Info</property>
    </properties>
   </component>
```

```
    <component name="speeding" type="table" source="speeding" width="40%"
height="50%">
      <properties>
       <property name="values">dateTime,boat_id,Max_speed,Location_Name</property>
       <property name="title">Last Speeding Event</property>
      </properties>
    </component>
    <component name="exclusion" type="table" source="exclusion" width="40%"
height="250px">
      <properties>
       <property name="values">boat_id,Location_Name,Minimum_Distance</property>
       <property name="title">Last Exclusion Violation</property>
      </properties>
    </component>
  </components>
 </config>
</app>
```

1 Creates a Dashboard app called Ships_DataDashboard.

2 Specifies three event-source elements called shipdata, speeding, and exclusion. All of the event sources deliver data every 250 milliseconds.

3 Creates three table components called boat_table, speeding, and exclusion.

The code renders the following Dashboard app:

Ship Info

boat_id	heading	speed	latitude	longitude
HMS Bounty	270.81	3.908	56.006	-3.45
Mayflower	85.95	3.438	56.015	-3.355
Santa Maria	270.29	4.489	55.999	-3.396

Last Speeding Event

dateTime	boat_id	Max_speed
2017-09-02T19:15:06.000Z	HMS Bounty	3.889
2017-09-02T19:15:08.000Z	HMS Bounty	3.889

Last Exclusion Violation

boat_id	Location_Name	Minimum_Distance
HMS Beagle	Rosyth Dockyard exclusion zone	657
HMS Beagle	Rosyth Dockyard exclusion zone	657

Navigation App

The Navigation app displays multiple components and is configured with any event sources that you want to include. Navigation apps are useful when you want to display multiple components in groupings or tabs. Tabs are displayed at the bottom of the app with a name and optional icon. For more information, see *"タブ" in SAS Event Stream Processing: Event Stream Processing モデルの XML 言語リファレンス*. Here is an example of a configured Navigation app with three tabs:

```
<app name="Ships_NavApp" type="navigation"> 1
 <title>Navigation App for Ship Data.</title>
 <description>Navigation App for Ship Data.</description>
```

```
 <config>
  <event-sources> 2
   <event-source name="shipdata" type="collection" window="Boat_Current"
interval="250"/>
   <event-source name="sourcewin" type="stream" window="Boat" interval="250"/>
  </event-sources>
  <tabs> 3
   <tab name="data" text="Ship Data" icon="Boat"/>
   <tab name="server" text="Server" icon="Server"/>
   <tab name="source" text="Source Data" icon="Boat"/>
  </tabs>
  <components> 4
   <component name="info" text="Ship Info" type="table" source="shipdata" width="40%"
height="250px" icon="GridView" tab="data">
    <properties>
     <property name="values">boat_id,heading,speed,latitude,longitude</property>
     <property name="title">Ship Info</property>
    </properties>
   </component>
   <component name="headings" text="Heading" source="shipdata" type="compass"
icon="ArrowSmallUp" tab="data">
    <properties>
     <property name="title">Headings</property>
     <property name="value">heading</property>
    </properties>
   </component>
   <component source="shipdata" text="Speed" type="gauge" icon="GrooveDashboard"
tab="data">
    <properties>
     <property name="title">Current Speed</property>
     <property name="value">speed</property>
    </properties>
   </component>
   <component name="speed" text="Speed Over Time" type="timeseries"
source="shipdata" icon="TimeSeriesChart" tab="data">
    <properties>
     <property name="title">Ship Speed</property>
     <property name="series">boat_id</property>
     <property name="y">speed</property>
     <property name="time">@timestamp</property>
     <property name="range">0,10</property>
     <property name="max">25</property>
     <property name="markersize">2</property>
    </properties>
   </component>
   <component name="model" type="modelviewer" text="Model Viewer" icon="OrgChart"
tab="server">
    <properties>
     <property name="zoom">true</property>
    </properties>
   </component>
   <component name="log" type="logviewer" text="Log Viewer" icon="Log" tab="server">
    <properties>
     <property name="title">Log Viewer</property>
    </properties>
   </component>
```

```
    <component name="src" type="table" source="sourcewin" tab="source">
      <properties>
        <property name="values">boat_id,heading,speed</property>
        <property name="title">Boat Stream Data</property>
      </properties>
    </component>
  </components>
 </config>
</app>
```

1 Creates a Navigation app called Ships_NavApp.

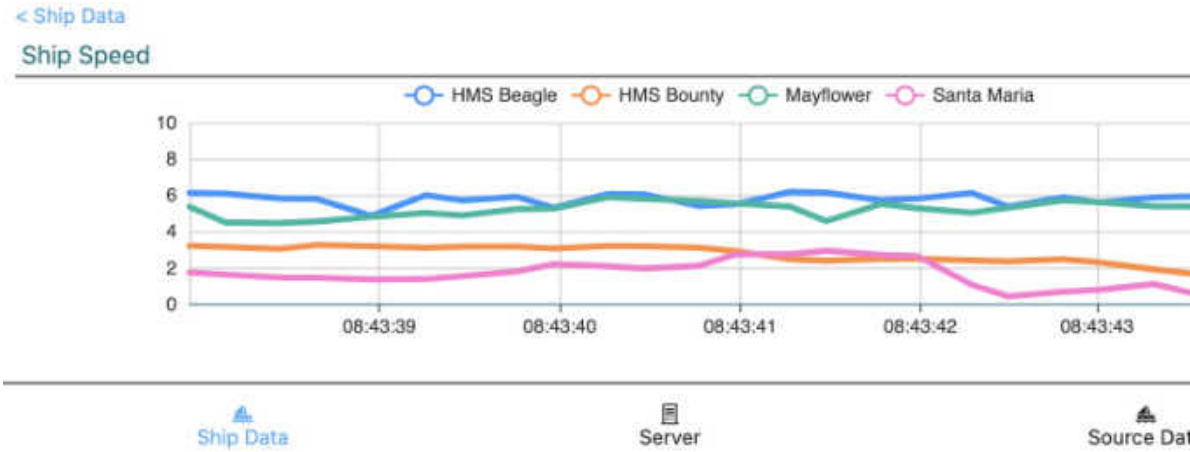
2 Specifies two event-source elements called shipdata and sourcewin. Both of the event sources deliver data every 250 milliseconds.

3 Specifies three tabs called data, server, and source to group the components.

4 Creates components that are displayed in the Navigation app. Each component must have an associated tab. Use the tab attribute to set the value to the name of one of the defined tabs.

The code renders the following Navigation app:

Because there are four components associated with the **Ship Data** tab, they are displayed as a list. You can select each component to expand it. Use the navigation link to return to the list of components:



If you navigate to a tab that has a singular component associated with it, the component is displayed immediately: