SYBASE®
An SAP® Company

ユーティリティ・ガイド

Sybase Event Stream Processor 5.0

ドキュメント ID： DC01755-01-0500-01
改訂： 2011 年 12 月

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

目次

第 1 章 Event Stream Processor の実行可能プログラムの概要

Sybase® Event Stream Processor には、ご使用の Event Stream Processor プロジェクトのコマンド・ライン制御を提供する実行可能プログラムが用意されています。

Event Stream Processor サーバの実行可能プログラム

Event Stream Processor の実行可能プログラムは、**esp_server**、**esp_cluster_admin**、**esp_monitor**、**esp_playback** の 4 つです。

- **esp_server –** ESP サーバを開始します。
- **esp_cluster_admin –** 対話型モードまたはコマンド・ライン・モードのどちらかで、クラスタ・マネージャと対話して、プロジェクト環境を設定します。
- **esp_monitor –** Event Stream Processor の実行インスタンスからパフォーマンス・データを受け取って表示します。 パフォーマンス・データは標準出力に書き込まれます。
- **esp_playback –** 送信中のデータをプレイバック・ファイルに記録し、取得したデータを Event Stream Processor の実行インスタンスに戻して再生します。

C & C (Command and Control) の実行可能プログラム

C & C (Command and Control) の実行可能プログラムは、**esp_cnc** と **esp_client** の 2 つです。

- **esp_cnc –** 個々の制御コマンドを実行します。このユーティリティはソースに添付されているので、クライアント・アプリケーションの作成例として使用できます。
- **esp_client –** メタデータのクエリ、データのローの挿入、ストリーム・スナップショットの作成、Event Stream Processor サーバの実行インスタンスの制御を実行します。 また、ESP サーバの停止、ストリームとその定義のリストの取得、ゲートウェイ・インタフェースのホストとポートの決定も行います。

パブリッシュとサブスクライブの実行可能プログラム

パブリッシュとサブスクライバの実行可能プログラムは、**esp_iqloader**、**esp_convert**、 **esp_kdbin**、 **esp_kdbout**、**esp_rapexport**、**esp_subscribe**、**esp_upload** の 7 つです。

- **esp_iqloader –** 指定されたストリームからのデータをアーカイブします。
- **esp_convert –** 標準入力から XML レコードまたは区切られたレコードを読み取って、標準出力にバイナリ・フォーマットのレコードを生成します。
- **esp_kdbin –** KDB データベース・テーブルから Event Stream Processor ストリームにデータを読み取ります。
- **esp_kdbout –** Event Stream Processor から KDB データベース・テーブルにストリーミング・データをフィードします。
- **esp_rapexport –** Event Stream Processor から Sybase RAP - The Trading Edition にデータをパブリッシュするアダプタを実行します。
- **esp_subscribe –** ESP サーバの実行インスタンスに接続して、ストリーミング・データをサブスクライブします。 受信レコードは XML (または必要に応じて区切りフォーマット) に変換されて、標準出力に書き込まれます。
- **esp_upload –** 標準入力からのバイナリ・レコードを記録し、ゲートウェイ・インタフェースを介して Event Stream Processor の実行インスタンスにそれらをパブリッシュします。

オーサリングの実行可能プログラム

オーサリングの実行可能プログラムは、**esp_compiler** と **esp_studio** の 2 つです。

- **esp_compiler –** Event Stream Processor のプロジェクト・ファイル (CCL) をランタイム・ファイル (CCX) にコンパイルします。
- **esp_studio –** このシェル・スクリプトは ESP スタジオを起動します。ESP スタジオは、Event Stream Processor プロジェクトの作成と、ESP サーバの起動とモニタに使用できるグラフィカルな環境です。

第 2 章 サーバの実行可能プログラム

コマンド・ライン・ユーティリティを使用して、希望する設定で ESP サーバを起動します。

esp_server

ESP サーバを起動するシェル・スクリプト。

構文

```
esp_server  [options...]
```

オプション

- **-cluster-node** <***file***> – (省略可能) 使用するノード設定ファイルを指定します。クラスタ・ノードの設定については、『管理者ガイド』を参照してください。
- **-cluster-log-properties** <***file***> – (省略可能) 使用するクラスタ・ロギング・プロパティ・ファイルを指定します。詳細については、『管理者ガイド』を参照してください。
- **-h または --help** – (省略可能) 使用可能なオプションのリストを各オプションの簡単な説明とともに画面に出力します。
- **-v または --version** – (省略可能) **esp_server** ユーティリティのバージョンを出力します。

使用法

```
cd $ESP_HOME/cluster/nodes/node1
$ESP_HOME/bin/esp_server --cluster-node node1.xml
```

esp_cluster_admin

対話型モードまたはコマンド・ライン・モードのどちらかで、クラスタ・マネージャとの対話に使用できるコマンド・ライン・ユーティリティ。

esp_cluster_admin ユーティリティでは、プロジェクト環境の設定、機密データの暗号化、キーストアの配備に使用できるいくつかのコマンドがサポートされています。このユーティリティにアクセスするには、認証を設定しないことを選択していない限り、ユーザ名とパスワードを指定します。

ESP サーバにログインすれば、**exit** または **quit** を実行して終了するまで、サーバ側でコマンドを実行し続けることができます。

注意： 対話型モードでは、ログインして、コマンドを実行し続けることができます。ユーティリティはクラスタ・マネージャとのセッションを維持します。コマンド・ライン・モードでは、コマンドの実行が終了するたびにユーティリティからログアウトされます。再度ログインしてから次の操作を実行します。

デフォルトでは、クラスタは 20 秒後に `TIMEOUT` を生成します。20 秒後に `TIMEOUT` オプションが発生しないように、対話型モードまたはコマンド・ライン・モードでタイムアウトを設定できます。SDK には、**START PROJECT** コマンドまたは **STOP PROJECT** コマンドのタイムアウト値 (秒単位) が用意されています。コマンドが `TIMEOUT` を返した場合は、**GET PROJECT** コマンドまたは **GET PROJECTS** コマンドを使用してプロジェクトの状況を取得します。

構文

esp_cluster_admin を呼び出します。

```
$ESP_HOME/bin/esp_cluster_admin <uri|help|helpi> [credentials]
[command] [options]
```

クラスタ・ノードが SSL に対応している場合は、**esp_cluster_admin** を呼び出します。

```
esp_cluster_admin --uri=esps://<host>:<port> [...]
```

RSA 認証でクラスタを管理するには、**esp_cluster_admin** を呼び出します。

```
esp_cluster_admin --uri=esp://<host>:<port> --key-alias=serverkey --
storepass=<storepass> --keystore=keystore.jks
```

Kerberos 認証または LDAP 認証でクラスタを管理するには、**esp_cluster_admin** を呼び出します。

```
esp_cluster_admin --uri=esp://<host>:<port> --user name=<user name>
--password=<password>
```

コマンド

- **get managers** – クラスタ・マネージャをリストします。
- **get controllers** – クラスタ・コントローラをリストします。
- **get workspaces** – 既存のすべてのワークスペースをリストします。
- **get projects** – ワークスペースに依存しない既存のすべてのプロジェクトをリストします。
- **get project <workspace-name>/<project-name>** – 関連付けられたワークスペース内のプロジェクトが表示されます。

  コマンドに続けて、ワークスペース名、スラッシュ、プロジェクト名を入力します。変数エントリにはスペースを使用しないでください。

たとえば、ワークスペース tradespace のプロジェクト sample を表示するには、次のように入力します。

```
get project tradespace/sample
```

- **get streams <workspace-name>/<project-name>** – プロジェクト内のすべてのストリームが表示されます。

  コマンドに続けて、ワークスペース名、スラッシュ、ストリーム名を入力します。変数エントリにはスペースを使用しないでください。

  たとえば、ワークスペース tradespace のストリーム tradedata を表示するには、次のように入力します。

  ```
  get streams tradespace/tradedata
  ```

- **get schema <workspace-name>/<project-name> <stream-name>** – 関連付けられたストリームのスキーマが表示されます。

  コマンドに続けて、ワークスペース名、スラッシュ、スキーマ名を入力します。変数エントリにはスペースを使用しないでください。

  たとえば、tradedata のスキーマ (dataformat) を表示するには、次のように入力します。

  ```
  get schema tradespace/tradedata/dataformat
  ```

- **add workspace <workspace-name>** – 新しいワークスペースを追加します。

  コマンドに続けて、ワークスペース名を入力します。変数エントリにはスペースを使用しないでください。

  たとえば、ワークスペース stockspace を追加するには、次のように入力します。

  ```
  add workspace stockspace
  ```

- **add project <workspace-name>/<project-name> <ccx> [<ccr>]** – ワークスペースにプロジェクトを追加します。

  コマンドに続けて、プロジェクトの追加先のワークスペース名、スラッシュ、プロジェクト名、ファイル名 (拡張子付き) を入力します。変数エントリにはスペースを使用しないでください。

  たとえば、プロジェクト tradeanalysis を同じファイル名でワークスペース tradespace に追加するには、次のように入力します。

  ```
  add project tradespace/tradeanalysis tradeanalysis.ccx
  ```

  ccr ファイルは、ccx ファイルのプロジェクト設定ファイルです。

- **remove workspace <workspace-name>** – ワークスペースを削除します。

  コマンドに続けて、削除するワークスペースの名前を入力します。変数エントリにはスペースを使用しないでください。

たとえば、ワークスペース stockspace を削除するには、次のように入力します。

```
remove workspace stockspace
```

- **remove project <workspace-name>/<project-name> –** プロジェクトを削除します。

  コマンドに続けて、削除するプロジェクトの名前を入力します。変数エントリにはスペースを使用しないでください。

  たとえば、ワークスペース tradespace からプロジェクト tradeanalysis を削除するには、次のように入力します。

  ```
  remove project tradespace/tradeanalysis
  ```

- **start project <workspace-name>/<project-name> [timeout(sec)] [<instance-index>] –** プロジェクトを開始します。

  コマンドに続けて、ワークスペース名、スラッシュ、開始するプロジェクトの名前を入力します。変数エントリにはスペースを使用しないでください。

  たとえば、ワークスペース tradespace でプロジェクト sample を開始するには、次のように入力します。

  ```
  start project tradespace/sample
  ```

- **stop project <workspace-name>/<project-name> [timeout(sec)] [<instance-index>] –** プロジェクトを停止します。

  コマンドに続けて、ワークスペース名、スラッシュ、停止するプロジェクトの名前を入力します。変数エントリにはスペースを使用しないでください。

  たとえば、ワークスペース tradespace でプロジェクト sample を停止するには、次のように入力します。

  ```
  stop project tradespace/sample
  ```

- **stop node <node-name> –** ノード (コントローラかマネージャまたはその両方) を停止します。

  コマンドに続けて、ノード名を入力します。

  例を示します。

  ```
  stop node node1
  ```

- **encrypt <clear-text> –** プレーン・テキスト・データを暗号化します。

  コマンドに続けて、暗号化する機密データを入力します。このコマンドを実行すると、ユーティリティによって暗号化テキストが生成され、このテキストを使用して、関連ファイル内の機密データを置き換えることができます。

  たとえば、パスワード 1234, を暗号化するには、次のように入力します。

  ```
  encrypt 1234
  ```

- **deploykey <new-username> <keystore> <storepass> <key-alias> [<store-type>] –** 新しいユーザ・キーをキーストアに配備して、新しいユーザを追加します。

コマンドに続けて、新しいユーザ名、キーストア・ファイル・パス、ストアパス・キーのエイリアスを入力します。

クラスタ内のすべてのノードは同じキーストア・ファイル・パスを共有します。deploy コマンドの送信先のノードがキーストアを更新してから、その他のノードがそのファイルを再ロードします。deploykey が正常に動作しているかどうかをテストするには、新しいキーを使用して、ただし別のノードから、クラスタにログインします。

たとえば、新しいユーザ名 `jdoe` で、ストアキーのエイリアス `jdoe` を使用して新しいキー `123456` を配備するには、次のように入力します。

```
deploykey jdoe ./mykeystore.jks 123456 jdoe
```

- **reload policy –** 実行中のクラスタに `policy.xml` ファイルを再ロードします。

  既存のポリシー・ファイルを最近更新した場合は、再ロード時に新しいポリシー設定に照らし合わせてクラスタが再検証されます。
- **connect –** プロジェクトをクラスタに接続または再接続します。このコマンドは対話型モードのみです。
- **quit または exit –** 対話型モードからログアウトされます。このユーティリティに再アクセスするには、認証を設定しないことを選択していない限り、ユーザ名とパスワードを指定します。
- **help –** 対話型モードの場合のユーティリティのコマンドと使用方法に関する説明がプレーン・テキストで表示されます。

コマンド・ライン・モードでの esp_cluster_admin

esp_cluster_admin ユーティリティでサポートされるコマンドのコマンド・ライン・オペレーションを実行します。

esp_cluster_admin 呼び出しのインスタンスごとに 1 つのコマンドを実行します。必要な回数だけ **esp_cluster_admin** 呼び出しを繰り返します。

構文

コマンド・ライン・モードの **esp_cluster_admin** コマンドを実行します。

```
$ESP_HOME/bin/esp_cluster_admin --uri=esp://<host>:<port> --
username=<user-name> --password=<password> --<command> <required-
parameters>
```

クラスタ・ノードが SSL に対応している場合は、**esp_cluster_admin** を呼び出します。

```
$ESP_HOME/bin/esp_cluster_admin --uri=esp://<host>:<port> [...]
```

RSA 認証でクラスタを管理するには、**esp_cluster_admin** を呼び出します。

```
esp_cluster_admin --uri=esp://<host>:<port> --key-alias=mytest --
storepass=<password> --keystore=<key_store> --<command> [options]
```

Kerberos 認証または LDAP 認証でクラスタを管理するには、**esp_cluster_admin** を呼び出します。

```
esp_cluster_admin --uri=esp://<host>:<port> --username=<username> --
password=<password>
```

オプション

- **--get_managers –** クラスタ・マネージャをリストします。
- **--get_controllers –** クラスタ・コントローラをリストします。
- **--get_workspaces –** 既存のすべてのワークスペースをリストします。
- **--get_projects –** ワークスペースに依存しない既存のすべてのプロジェクトをリストします。
- **--get_projectdetail –** コマンドに続けて、ワークスペース名、スラッシュ、希望するプロジェクト名を入力します。

  たとえば、ワークスペース `ws1` のプロジェクト `project1` の詳細を表示するには、次のように入力します。

  ```
  esp_cluster_admin --uri=esp://localhost:19011 --username=user --
  password=pass --get_projectdetail --workspace-name=ws1 --project-
  name=project1
  ```
- **--get_streams –** コマンドに続けて、ワークスペース名、希望するプロジェクト名を入力します。

  たとえば、ワークスペース `ws1` のプロジェクト `project1` のストリームを表示するには、次のように入力します。

  ```
  esp_cluster_admin --uri=esp://localhost:19011 --username=user --
  password=pass --get_streams --workspace-name=ws1 --project-
  name=project1
  ```
- **--get_schema –** コマンドに続けて、ワークスペース名、プロジェクト名、ストリーム名を入力します。

  たとえば、ワークスペース `ws1` のプロジェクト `project1` のストリーム `win` のスキーマを表示するには、次のように入力します。

  ```
  esp_cluster_admin --uri=esp://localhost:19011 --username=user --
  password=pass --get_schema --workspace-name=ws1 --project-
  name=project1 --stream-name win
  ```
- **--add_workspace –** コマンドに続けて、ワークスペース名を入力します。

  たとえば、ワークスペース `ws2` を追加するには、次のように入力します。

  ```
  esp_cluster_admin --uri=esp://localhost:19011 --username=user --
  password=pass --add_workspace --workspace-name=ws2
  ```
- **--add_project –** コマンドに続けて、ワークスペース名、プロジェクト名、プロジェクト・ファイル名 (拡張子付き) を入力します。

  たとえば、ワークスペース `ws2` にプロジェクト `project1` を追加するには、次のように入力します。

```
esp_cluster_admin --uri=esp://localhost:19011 --username=user --
password=pass --add_project --workspace-name=ws1 --project-
name=project1 --ccx=project1.ccx
```

- **--remove_project –** コマンドに続けて、ワークスペース名、プロジェクト名を入力します。

  たとえば、ワークスペース `ws1` からプロジェクトを `project1` を削除するには、次のように入力します。

  ```
  esp_cluster_admin --uri=esp://localhost:19011 --username=user --
  password=pass --remove_project --workspace-name=ws1 --project-
  name=project1
  ```

- **--start_project –** コマンドに続けて、ワークスペース名、プロジェクト名を入力します。

  たとえば、ワークスペース `ws1` でプロジェクト `project1` を開始するには、次のように入力します。

  ```
  esp_cluster_admin --uri=esp://localhost:19011 --username=user --
  password=pass --start_project --workspace-name=ws1 --project-
  name=project1
  ```

- **--stop_project –** コマンドに続けて、ワークスペース名、プロジェクト名を入力します。

  たとえば、ワークスペース `ws1` でプロジェクト `project1` を停止するには、次のように入力します。

  ```
  esp_cluster_admin --uri=esp://localhost:19011 --username=user --
  password=pass --stop_project --workspace-name=ws1 --project-
  name=project1
  ```

- **--stop_node –** コマンドに続けて、ノード名を入力します。

  たとえば、ノード `node1` を停止するには、次のように入力します。

  ```
  esp_cluster_admin --uri=esp://localhost:19011 --username=user --
  password=pass --stop_node --node-name=node1
  ```

- **--encrypt_text –** コマンドに続けて、暗号化する機密データを入力します。

  たとえば、テキスト `1234` を暗号化するには、次のように入力します。

  ```
  esp_cluster_admin --uri=esp://localhost:19011 --username=user --
  password=pass --encrypt_text --text=1234
  ```

- **--reload_policy –** `policy.xml` ファイルを再ロードします。
- **--deploy_key –** コマンドに続けて、新しいユーザ名、キーストア・ファイル・パス、ストアパス・キーのエイリアスを入力します。

  たとえば、キーストア `mykeystore.jks` に新しいストアパス・キーのエイリアス `user` を追加するには、次のように入力します。

  ```
  esp_cluster_admin --uri=esp://localhost:19011 --username=user --
  password=pass --deploy_key --new-user=user --
  keystore=mykeystore.jks --storepass=123456 --key-alias=user
  ```

- **--help –** ユーティリティのコマンドと使用方法に関する説明がプレーン・テキストで表示されます。

esp_monitor

Event Stream Processor の実行インスタンスからパフォーマンス・データを読み取り、そのデータを XML フォーマットで標準出力に出力します。

データをモニタできるのは、プロジェクト設定 (CCR) ファイルに時間粒度オプションが設定されている場合のみです。 時間粒度オプションでは、実行中の Event Stream Processor からパフォーマンス・レコード (ストリームあたり 1 つとゲートウェア接続あたり 1 つ) のセットを取得する間隔を秒単位で指定します。 **_ESP_Clients_Monitor** ストリームには接続クライアントに関する基本情報がありますが、パフォーマンス関連のフィールドにデータが挿入されるのはモニタ・オプションを設定した場合のみです。

ストリームごとにこのフォーマットのレコードが生成されます。

```
<_ESP_Streams_Monitor ESP_OPS="i"
    stream="stream1"
 cpu_pct="0.000000"
    trans_per_sec="0.499451"    rows_per_sec="1.098791"
    inc_trans="5" inc_rows="11"
    queue="0"
    store_rows="2"
    last_update="2008-08-26 14:17:14"
    sequence="123"
 posting_to_client="-1"
/>
```

- **ESP_OPS –** レコードの opcode を保持します。
- **stream –** ストリームの名前が入ります。このストリームの統計がレポートされます。
- **cpu_pct –** 前回のレポート間隔での CPU 使用率。
- **trans_per_sec –** この間隔でのトランザクション率。
- **rows_per_sec –** この間隔でのロー受信率。
- **inc_trans –** この間隔でのトランザクション数。
- **inc_rows –** この間隔での新規ロー数。
- **queue –** ストリームのキュー内のレコード数。
- **store_rows –** テーブルのロー数。
- **last_update –** 前回の更新日時。
- **sequence –** 更新のシーケンス番号 (ストリーム名と last_update によって既にユニークな識別が行われているので、これは冗長です)。

- **posting_to_client –** ストリームがその時点でデータのポストを試みたゲートウェイ・クライアントのハンドル。ない場合は -1 です。

ゲートウェイ・クライアントごとにこのフォーマットのレコードが生成されます。

```
<_ESP_Clients_Monitor ESP_OPS="i"
    handle="130"
    ip="127.0.0.1"
    host="localhost"
    port="59645"
    login_time="2011-08-11 06:35:27.647137"
/>
<_ESP_Clients_Monitor ESP_OPS="u"
    handle="129" user_name="user" ip="127.0.0.1" host="localhost"
    port="12345" login_time="2008-08-26 12:05:01" conn_tag="rdr"
    cpu_pct="0.000000" last_update="2008-08-26 14:17:14"
    subscribed="1" sub_trans_per_sec="0.499451"
    sub_rows_per_sec="1.098791" sub_inc_trans="5"
    sub_inc_rows="11" sub_total_trans="502" sub_total_rows="1018"
    sub_dropped_rows="0" sub_accum_size="0"
    sub_queue="0" sub_queue_fill_pct="0.000000" sub_work_queue="0"
    pub_trans_per_sec="0.000000" pub_rows_per_sec="0.000000"
    pub_inc_trans="0" pub_inc_rows="0" pub_total_trans="0"
    pub_total_rows="0" pub_stream_id="-1"
    />
```

- **ESP_OPS –** レコードの opcode。
- **handle –** このゲートウェイ・クライアントのハンドル。
- **user_name –** このクライアントのユーザ名。
- **ip –** このクライアントの接続元のアドレス。
- **host –** このクライアントの接続元のホスト名 (解決可能な場合)。
- **port –** このクライアントの接続元のポート。
- **login_time –** このクライアントがログインした時点のタイムスタンプ。
- **conn_tag –** 接続タグ (存在する場合)。
- **cpu_pct –** このクライアントのゲートウェイ・スレッドによる、前回のレポート間隔での CPU 使用率。
- **last_update –** 前回の更新日時。
- **subscribed –** このクライアントがサブスクライブしている場合は 1、していない場合は 0。
- **sub_trans_per_sec –** この間隔でのサブスクリプション・トランザクション率。エンベロープとサービス・メッセージもトランザクションとしてカウントされます。
- **sub_rows_per_sec –** この間隔でのサブスクリプション・ロー率。
- **sub_inc_trans –** この間隔でのサブスクリプション・トランザクション／エンベロープ／メッセージの数。

- **sub_inc_rows –** この間隔でのサブスクリプション・ロー数。
- **sub_total_trans –** 送信されたサブスクリプション・トランザクション／エンベロープ／メッセージの総数。
- **sub_total_rows –** 送信されたトランザクション・ローの総数。
- **sub_dropped_rows –** クライアントが保持していないために削除されたサブスクリプション・ローの数。
- **sub_accum_size –** パルス・サブスクリプションの場合に、アキュムレータに収集されて次のパルスで送信される現在のローの数。
- **sub_queue –** このクライアントの「適切なキュー」内のレコード数 (バッファされたデータの合計量は **sub_accum_size**、**sub_queue**、**sub_work_queue** で構成されます)。
- **sub_queue_fill_pct –** その制限を基準とする **sub_queue** のサイズの比率 (%)。
- **sub_work_queue –** キューからソケット・バッファに転送されるレコードの数。
- **pub_trans_per_sec –** この間隔でのパブリッシュ・トランザクション率。エンベロープとサービス・メッセージもトランザクションとしてカウントされます。
- **pub_rows_per_sec –** この間隔でのパブリッシュ・ロー率。
- **pub_inc_trans –** この間隔でのパブリッシュ・トランザクション、エンベロープ、またはメッセージの数。
- **pub_inc_rows –** この間隔でのパブリッシュ・ロー数。
- **pub_total_trans –** この間隔でのパブリッシュ・トランザクション、エンベロープ、またはメッセージの総数。
- **pub_total_rows –** 受信したパブリッシュ・ローの総数。
- **pub_stream_id –** パブリッシャが現在のパブリッシュ先のストリームに書き込めない場合は -1、それ以外の場合はそのストリームの数値 ID。

構文

```
esp_monitor -p [<host>:]<port>/workspace-name/project-name -c
user[:password] [OPTION...]
```

必須の引数

- **-p *[<host>:]<port>/workspace-name/project-name* –** (必須) *host:<port>/<workspace name>/ <project name>* のすべての引数で、ESP サーバ (クラスタ・マネージャ) に接続する場合の URI を指定します。たとえば、ESP クラスタ・サーバを起動したときのポートが [19011] で、ホスト名が [localhost] として設定され、デフォルトのワークスペースで [prj1] プロジェクトを実行している場合は、`-p` を次のように指定します。 `-p localhost:19011/default/prj1`

オプション

- **-c *user[:password]*** – (必須) *user* ID と、オプションで *password* を使用して認証します。 *password* が指定されていないと、ユーザにパスワードの入力を求めるプロンプトが表示されます。 Event Stream Processor でこれらのクレデンシャルを使用して認証が正常に行われた場合は、接続が維持されます。それ以外の場合は、Event Stream Processor はすぐに接続を切断します。
- **-k *privateRsaKeyFile*** – (省略可能) パスワード認証の代わりに、RSA プライベート・キー・ファイルのメカニズムを使用して認証を実行します。 **privateRsaKeyFile** には、プライベート RSA キー・ファイルのパス名を指定します。

注意： ESP サーバの起動時に `-k` オプションで RSA キーの保存先のディレクトリを指定したことを確認してください。

例

- **Event Stream Processor のインスタンスのモニタ** – ホスト "myhost.sybase.com" で 31415 の C & C (Command and Control) ポートを使用している Event Stream Processor の実行インスタンスをモニタするには、次のように入力します。

```
esp_monitor -p myhost.sybase.com:31415/workspace-name/project-
name -c user:pass
```

esp_playback

指定された速度でさまざまなソースから Event Stream Processor にデータをロードします。

現在サポートされているフォーマットは、ESP XML、カンマ区切り値、ESP バイナリ、ODBC ソースです。

このツールは、ユーザ指定の速度でデータを再生することもできます。この速度は、ロー数／ミリ秒単位、または入力データの timestamp/datetime カラムの値によって決まる速度のどちらででも指定できます。 timestamp/datetime カラムの方法を使用する場合は、時間スケールの速度を指定できます。これを使用してプレイバック速度を速くしたり遅くしたりできます。

このツールの目的は、Sybase ファイル・ソースでのみ機能する、**esp_upload** と **esp_convert** を置き換えることです。

構文

```
esp_playback -p [<host>:]<port>/workspace-name/project-name -c
user[:password] [OPTION...]
```

オプション

- **-a** – (省略可能) 非同期パブリッシュを使用します。 このモードでは、パブリッシュは ESP サーバからの受信データの受信確認を待ちません。 デフォルトは同期パブリッシュです。
- **-p** ***[<host>:]<port>/workspace-name/project-name*** – (必須) *host:<port>/<workspace name>/ <project name>* のすべての引数で、ESP サーバ (クラスタ・マネージャ) に接続する場合の URI を指定します。たとえば、ESP クラスタ・サーバを起動したときのポートが [19011] で、ホスト名が [localhost] として設定され、デフォルトのワークスペースで [prj1] プロジェクトを実行している場合は、`-p` を次のように指定します。 `-p localhost:19011/default/prj1`
- **-c** ***user[:password]*** – (必須) *user* ID と、オプションで *password* を使用して認証します。 *password* オプションも **-k** オプションも指定されていないと、ユーザにパスワードの入力を求めるプロンプトが表示されます。 Event Stream Processor でこれらのクレデンシャルを使用して認証が正常に行われた場合は、接続が維持されます。それ以外の場合は、Event Stream Processor はすぐに接続を切断します。
- **-h** – (省略可能) 使用可能なオプションのリストを各オプションの簡単な説明とともに画面に出力します。
- **-i** – (省略可能) シャインスルー・フラグをオンにします。 このモードでは、更新時に欠落しているカラムがあると、更新対象のローの以前の値がそれらのカラムに挿入されます。 デフォルトの動作では、欠落しているカラムに NULL が挿入されます。
- **-e** – (省略可能) ESP サーバへの通信を暗号化することを指定します。 これを機能させる場合は、Event Stream Processor を暗号化モードで起動します。
- **-k** ***privateRsaKeyFile*** – (省略可能) パスワード認証の代わりに、RSA プライベート・キー・ファイルのメカニズムを使用して認証を実行します。 **privateRsaKeyFile** には、プライベート RSA キー・ファイルのパス名を指定します。

  注意： ESP サーバの起動時に `-k` オプションで RSA キーの保存先のディレクトリを指定したことを確認してください。
- **-C** ***connStr*** – (依存的に必須) ソースの接続文字列を指定します。接続文字列はソースによって異なります。

  サポートされるソースとその接続文字列のフォーマットは、次のとおりです。

  - ODBC：`{0}odbc:<dsnName>:<sql>|<file>:<fileName>|<query>`
  - ESP XML：`espxml:<inputFile>`
  - ESP DLM：`espdlm:<inputFile>[:<delimiter>]`

- バイナリ：`binary:<inputFile>`
- レコーダ：`recorder:<inputFile>`
- KDB： `kdb:<host>:<port>:<sql>|<file>:<filename>|<query>:[<user>:<password>]`

注意： ESP XML ソースまたは ESP DLM ソースを使用する場合は、このパラメータは (省略可能) です。

- **odbc –** ODBC ソース文字列 ID。このソース・オプションを使用するには、このユーティリティを実行しているマシンに、必要なソースの ODBC 3.0 準拠ドライバをインストールして設定します。
- ***dsnName* –** ODBC データ・ソース名。
- **file|sql –** 次の引数がファイル名か SQL クエリかを指定します。 どちらかの値を指定する必要があります。
- ***fileName*|*query* –** *fileName* は、SQL 文があるファイルの名前です。 query は、データを取得するために実行する必要がある SQL クエリです。 どちらかの値を指定する必要があります。

```
espxml:inputFile
```

それぞれの意味は、次のとおりです。

- **espxml –** ESP XML ファイル・ソースの識別子です。
- ***inputFile* –** ESP DLM フォーマットのデータがあるファイルのフル・パスと名前です。

```
espdlm:inputFile[:delimiter]
```

それぞれの意味は、次のとおりです。

- **espdlm –** ESP DLM ファイル・ソースの識別子です。
- ***inputFile* –** ESP DLM フォーマットのデータがあるファイルのフル・パスと名前です。
- ***delimiter* –** このオプション・パラメータは 1 文字のフィールド・デリミタです。デフォルトはカンマです。

```
espxml:inputFile
```

それぞれの意味は、次のとおりです。

- **espxml –** ESP XML ファイル・ソースの識別子です。

- ***inputFile*** **–** ESP DLM フォーマットのデータがあるファイルのフル・パスと名前です。

```
binary:inputFile
```

それぞれの意味は、次のとおりです。

- **binary –** ESP バイナリ・ファイル・ソースの識別子です。このフォーマットの長所はロードが高速であることです。データは既に Event Stream Processor が吸収できるフォーマットのため、変換は必要ありません。 このフォーマットの短所はマシンのアーキテクチャに固有であることです。

  esp_convert を使用して、XML または DML のどちらからでもバイナリ・フォーマットにデータを変換できます。
- ***inputFile*** **–** バイナリ・フォーマットのデータがあるファイルのフル・パスと名前です。

```
recorder:inputFile
```

それぞれの意味は、次のとおりです。

- **recorder –** レコーダで生成されるファイルの識別子です。レコーダを開始するには、ESP スタジオを使用するか、`$ESP_HOME/client/pubsub/` フォルダにあるレコーダのサンプルを使用します。
- ***inputFile*** **–** レコーダで生成されたデータがあるファイルのフル・パスと名前です。
- **-R** ***playRate*** **–** データをプレイバックする速度を指定します。 このパラメータが指定されていないと、レコードは可能な限り高速で再生されます。 次の例は、プレイバック速度を指定できる 2 つの方法を示します。

```
          records:milliseconds
```

それぞれの意味は、次のとおりです。

- **records –** 所定のミリ秒間にパブリッシュするレコード数です。 この値に 0 を指定できるのは、*millisecond* コンポーネントも 0 の場合のみです。 値 0 は、できる限り高速で再生することを示します。
- **milliseconds –** 所定の数のレコードをプレイバックする時間 (ミリ秒数) です。

  このプロパティは、レコーダ・タイプのソースではサポートされません。

```
          columnName
```

それぞれの意味は、次のとおりです。

- **columnName –** レコードをプレイバックする速度を制御するターゲット・ストリーム内のカラム名です。 カラム名は大文字と小文字が区別されます。指定されたカラム名がターゲット・ストリームにない場合はエラーが報告されます。

  columnName プロパティは、レコーダ・タイプのソースでは無視されます。また、現時点では、バイナリ・ファイルのソースではサポートされません。

- ***timeScaleRate* –** 2 つの連続するレコードの時間の差分に対する増倍率を指定します。 プレイバック速度をソースのカラムで制御するときに、プレイバック速度と一緒に使用されます。 -N と +M の間の `integer` 値を取ります。 +1 より大きい正の値では、速度が速くなります。-1 より小さい負の値では、速度が遅くなります。 +1 または -1 では、カラムで指定された速度でプレイバックが実行されます。値 0 はカラムを無視することを指定します。 デフォルト値は 1 です。
- **-r *interval* –** パブリッシュされた統計のレポートから次のレポートまでの最小待機時間 (秒数) を指定します。 デフォルトは 5 秒です。 値 0 を指定すると、レポートは実行されません。 パブリッシュ対象のレコードがあると、時間間隔チェックがトリガされます。 つまり、レポート間隔を過ぎても、チェックをトリガするレコードがなければ、統計はレポートされません。
- **-B *bufferSize* –** 内部の読み取りと書き込みの各バッファ・サイズを指定します。デフォルトのバッファ・サイズは 32K です。これは最大許容値でもあります。バッファ・サイズが小さいほど、使用するメモリは減りますが、実行速度が遅くなる可能性があります。
- **-t *size* –** トランザクション・ブロックを使用して ESP サーバにデータをパブリッシュすることを指定します。各ブロックのサイズは *size* で指定します。 トランザクション・ブロックでデータを送信すると、ESP サーバでのデータの処理が高速になります。 パフォーマンスの向上は、次の 2 つの方法で実現されます。まず、大量のレコードが 1 つのネットワーク・パケットにまとめられるので、ネットワークの使用効率が向上します。また、ESP サーバは複数のレコードを 1 つのブロックとして扱うので、失敗か成功かはブロック全体で決まります。このため、処理のオーバヘッドが削減されます。 アプリケーションの性質に応じて、このオプションは適さない場合もあります。
- **-w *size* –** エンベロープを使用して Event Stream Processor にデータをパブリッシュすることを指定します。エンベロープのサイズは *size* で指定します。 *-t* オプションも *-w* オプションも指定されていない場合、デフォルト値は -w64 です。*size* の値は 1 から 1024 の間で指定してください。 このオプションを使用した場合は、レコードを個々のレコードとして扱うように ESP サーバを変更して、ESP サーバにデータを送信するときのネットワークの効率を確保します。

- **-H [*hostname* :] *port* –** *hostname* と *port* を指定するか、ESP サーバのホット・スペア・インスタンスの C & C (Command and Control) インタフェースのポートを単に指定します。 デフォルトの hostname は localhost です。
- **-s *streamName* –** (必須) このユーティリティのターゲット・ストリーム名を指定します。 ODBC ソースには、このオプションは必須です。 その他のすべてのソースでは、ターゲット・ストリーム名／ストリーム ID はソースに埋め込まれています。

  streamName 変数は大文字と小文字が区別されます。
- **-S *maxStringSize* –** 処理できる最大文字列サイズを指定します。 このオプションは ODBC ソースで使用されます。 デフォルト値は 1024 です。

  この値はグローバルであり、ソース内のすべての文字列に適用されます。 このオプションに大きい値を指定すると、レコード内の文字列の数と、`-B` オプションで指定した値に応じて、メモリの使用量が増加する可能性があります。
- **-m *dateMask* –** XML ファイルと区切りファイルに使用する日付マスク。 デフォルトは "%Y-%m-%dT%H:%M:%S" です。 日付マスクは、すべての日付カラムに共通です。 このオプションは、ESP XML ファイル・ソースと区切りファイル・ソースの場合にのみ有効です。

例

- **データの読み取りとプレイバック –** たとえば、コマンド・ラインの SQL 文を使用して、DSN 名が foo の ODBC ソースからストリーム foobar にデータを読み取るには、次のように入力します。

```
esp_playback -c user:pass -p localhost:19022/w1/p1 -C
'odbc:foo:query:select * from foobar_db' -s foobar
```

  たとえば、ESP XML ソース `foo.xml` からデータを読み取るには、10,000 ロー／秒の速度でデータを再生して、15 秒ごとに進捗状況をレポートします。

```
esp_playback -c user:pass -p localhost:19022/w1/p1 -C 'espxml:/
tmp/foo.xml' -s bar -R 10000:1000 -r 15
```

第 3 章 C & C (Command and Control) の実行可能プログラム

コマンド・ライン・ユーティリティを使用して、Event Stream Processor に接続してその情報を取得します。

esp_cnc

C & C (Command and Control) インタフェースとゲートウェイ・インタフェースを介して Event Stream Processor に接続し、単純な C & C コマンドをサーバに発行します。結果を標準出力に出力します。

構文

```
esp_cnc -C command -p [host:]port/workspace-name/project-name
[OPTION]
```

必須の引数

- **-C *command*** – (必須) 次のリテラルのいずれかを指定できます。 **getGateway**, **getBaseStreams**, **getDerivedStreams**, **getStreamDefinition**, **getAddressSize**, **getDateSize**, **sendStreamsExit**, **augmentSubscriber**, **removeSubscriber**, **isBigEndian**, **isQuiesced**, **isQuiescedNow**, **returnWhenQuiesced**, **setParam**, **getVersion**, **getStreamHandle**
- **-p *[<host>:]<port>/workspace-name/project-name*** – (必須) *host:<port>/<workspace name>/ <project name>* のすべての引数で、ESP サーバ (クラスタ・マネージャ) に接続する場合の URI を指定します。たとえば、ESP クラスタ・サーバを起動したときのポートが [19011] で、ホスト名が [localhost] として設定され、デフォルトのワークスペースで [prj1] プロジェクトを実行している場合は、`-p` を次のように指定します。 `-p localhost:19011/default/prj1`

オプション

- **-c *user[:password]*** – (省略可能) *user* ID と、オプションで *password* を使用して認証します。 *password* が指定されていないと、ユーザにパスワードの入力を求めるプロンプトが表示されます。 Event Stream Processor でこれらのクレデンシャルを使用して認証が正常に行われた場合は、接続が維持されます。それ以外の場合は、Event Stream Processor はすぐに接続を切断します。

- **-e** – (依存的に必須) openSSL を介するトラフィックを暗号化します。 このオプションがないと、暗号化は行われません。
- **-h** – (省略可能) 使用可能なオプションのリストを各オプションの簡単な説明とともに画面に出力します。
- **-H *handle*** – (依存的に必須) **augmentSubscriber** コマンドまたは **removeSubscriber** コマンドを発行するためのクライアント・ハンドルを指定します。 これらのコマンドには、このオプションは必須です。
- **-k *privateRsaKeyFile*** – (省略可能) パスワード認証の代わりに、RSA プライベート・キー・ファイルのメカニズムを使用して認証を実行します。 *privateRsaKeyFile* には、プライベート RSA キー・ファイルのパス名を指定します。 `-V rsa` オプションを指定して ESP サーバを起動した場合は、このオプションは必須です。 このオプションが有効な場合、ユーザ名は `-c` オプションで指定する必要がありますが、パスワードは必要ありません。

  注意： ESP サーバの起動時に `-k` オプションで RSA キーの保存先のディレクトリを指定したことを確認してください。
- **-P *name:value*** – (依存的に必須) 変数に関連付ける新しい値を指定します。 **setParam** オプションに必須です。実行中のプロジェクトではパラメータの値を設定できません。`-P` オプションを使用するのは、変数の新しい値を設定する場合のみです。
- **-s *stream*** – (依存的に必須) 1 つのストリームを指定します。 コマンド **getStreamDefinition**、**augmentSubscriber**、**getStreamHandle**、**removeSubscriber** に必須です。

esp_client

Event Stream Processor の実行インスタンスからの情報を制御して取得します。

構文

```
esp_client -p [<host>:]<port>/workspace-name/project-name
[OPTION...] [COMMAND...]
```

オプション

- **-c *user[:password]*** – (省略可能) *user* ID と *password* を使用して認証を実行します。 *password* が指定されていないと、パスワードの入力を求めるプロンプトが表示されます。 Event Stream Processor でこれらのクレデンシャルを使用して認証が正常に行われた場合は、接続が維持されます。それ以外の場合は、Event Stream Processor はすぐに接続を切断します。

- **-h** – (省略可能) 使用可能なオプションのリストを各オプションの簡単な説明とともに画面に出力します。
- **-i *file*** – (省略可能) 指定されたファイル内のコマンドを実行します。
- **-k *privateRsaKeyFile*** – (省略可能) パスワード認証の代わりに、RSA プライベート・キー・ファイルのメカニズムを使用して認証を実行します。 RSA プライベート・キー・ファイルの *privateRsaKeyFile* パス名が指定されていることを確認してください。 `-V` RSA オプションを指定して Event Stream Processor を起動した場合、このオプションは必須です。 このオプションが有効な場合、ユーザ名は `-c` オプションで指定する必要がありますが、パスワードは必要ありません。

  注意： ESP サーバの起動時に `-k` オプションで RSA キーの保存先のディレクトリを指定したことを確認してください。
- **-p *[<host>:]<port>/workspace-name/project-name*** – (必須) *host:<port>/<workspace name>/ <project name>* のすべての引数で、ESP サーバ (クラスタ・マネージャ) に接続する場合の URI を指定します。たとえば、ESP クラスタ・サーバを起動したときのポートが [19011] で、ホスト名が [localhost] として設定され、デフォルトのワークスペースで [prj1] プロジェクトを実行している場合は、`-p` を次のように指定します。 `-p localhost:19011/default/prj1`
- **-q** – (省略可能) プロンプトを無効にします。Windows 環境で標準入力にパイプで接続されているファイルからコマンドを読み取る場合にこのオプションは役立ちます。
- **-x** – (省略可能) 実行前のコマンドのエコーを有効にします。これは、コマンド **echo on|off** で後で変更することもできます。
- **-v** – (省略可能) **esp_client** ユーティリティのバージョンを出力します。

コマンド
コマンドがパラメータとして取る任意の値を、Windows の場合は一重逆引用符 (`) で囲み、Linux または Solaris の場合は中カッコ `{}` で囲みます。たとえば、`` `parameter` `` または `{parameter}` と入力します。 コマンドはセミコロンで区切ります。

注意： シェルや SPLASH の式の構文で使用される通常の一重引用符 (') と二重引用符 (") との混同を避けるために、逆引用符 (`) を使用します。

逆引用符で囲まれた文字列では、文字列自体の中で逆引用符を使用できません。コマンド・ラインでは、**esp_client** コマンドに二重引用符を使用することはできません。UNIX シェルが解釈しようとするためです。 これを防ぐため、逆引用符を中カッコ ({}) で保護します。例を示します。

```
esp_client -p localhost:19022/w1/p1
"help;addrsize;streams;endian;datesize;clock;idx {allTypes1};stream
```

```
{allTypes2}"
 esp_client -p 10.44.147.231:22555 "history {175}"
```

文字列を中カッコ ({}) で囲む場合、その文字列の内側にある左右の中カッコの数が同じであることが必要です。例を示します。

```
{ currow.value = '{a}' }
```

複数行の値には逆チックと中カッコを使用できません。設定ファイルなど、大きな複数行パラメータ向けの別の引用スタイルがあります。これらのパラメータは <<! で始まり、その後にパラメータのインライン・テキストが続き、最後は ！のみの行になります (この前後に空白文字は入りません)。 この構文は、シェルの "<<" 構文と似ていますが、終了ワード "!" は変更できません。"<<!" の後の改行と最後の "!" の前の改行はパラメータの一部ではないので、この構文を使用して単一行の値を指定することもできます。 複数のインライン・パラメータを指定するには、!<<! の行を使用してパラメータを区切ります。 例を示します。

```
load_config_inline_conv {nobackup,nocompat} <<!
... text of the model ...
!<<!
... text of the conversion model ...
!
```

注意： この構文はどのコマンドでも使用できます。

多くのコマンドでは、次のように入力して、出力をファイルにリダイレクトできます。

```
command > `outfile.dat`
command >> `outfile.dat`
command | `filter-program`
```

">" 演算子を指定すると、既存のファイルが上書きされます。">>" を指定すると、既存のファイルに追加されます。演算子 "|" はパイプを使用して出力を UNIX コマンド・パイプラインに接続します。 ファイル名またはフィルタ・プログラムは、逆引用符または中カッコのどちらかを使用して囲みます。

一般的なコマンドは、次のとおりです。

- **addrsize –** 接続された Event Stream Processor インスタンスのアドレスまたはポインタ・サイズ (バイト単位) を出力します (32 ビット・アドレス指定の場合は 4 バイト、64 ビット・アドレス指定の場合は 8 バイトになります)。この値は、接続されたプラットフォーム・インスタンスをコンパイルする方法 (32 ビットまたは 64 ビット) を反映しています。 たとえば、32 ビットのプラットフォームは 64 ビットのホストで実行でき、その場合、**addrsize** コマンドは値 8 ではなく 4 を返します。
- **backup –** すべてのログ・ストアのバックアップを作成します。バックアップ・ファイルはサフィックス ".bak" を付けて作成されます。たとえば、

`dynamic.log` はファイル `dynamic.bak` にバックアップされます。 バックアップ・ファイルは、内容が圧縮されて作成されます。

- **clear base stream `*str*` –** 指定された基準ストリームの内容を削除します。
- **clock –** Event Stream Processor 論理クロックの現在の状態を出力します。 例を示します。

```
current time: 1071014401.018 2003-12-10 00:00:01.018
rate: 6.000 real: 0 stop depth: 0 max sleep: 100
```

time は、UNIX エポックからの秒数とユーザ判読可能な値の両方で出力されます。 rate は、実時間を基準とするクロック・レートです。 10 は「10 倍速い」、0.1 は「10 倍遅い」という意味です。 real フラグは、Event Stream Processor が実行されているマシンのシステム時間とクロックが一致するか (1)、クロックは人為的に変更されているか (0) を示します。 stop depth は、クロックが再帰的に停止された回数 (時間の流れを実際に再開するためにクロックの開始を呼び出す必要がある回数) を示します。 クロックが実行されている場合、stop depth は 0 です。max sleep はすべてのスリーパがクロック・レートまたはクロック時間の変化を検出することを保証する時間 (実際のミリ秒単位) です。 クロック・レートを変更する呼び出しは自動的にスリープして (論理クロックが停止して)、その影響が確実に反映されるようにします。

- **clock [rate `*number*`] –** プラットフォームのクロック・レートを変更します。rate は浮動小数点数型として指定します。最小レートは 0.001 です。
- **clock [time `*number*`] –** プラットフォームの現在のクロック時間を変更します。time は、UNIX エポックからの秒数を浮動小数点型として指定するか、year-month-day Thour:min:sec のフォーマットで指定することができます。同じ指定でミリ秒を付け加える場合は、year-month-dayThour:min:sec.NNN になります。文字 "T" は、デフォルトの Event Stream Processor 時間フォーマットの場合と同様に、リテラルです。 time 値にプレフィクスとして **add** を付けた場合は、現在の時間に加える変更が指定されます。 この場合は、秒数を浮動小数点数型で指定します。 コマンドを実行する前のクロックの以前の状態が出力されます。 **clock** コマンドの説明を参照してください。
- **clock [rate `*number*`] [time [add] `*number*`] –** 論理クロックの現在の時間かレートまたはその両方を変更します。rate は浮動小数点数型として指定します。最小レートは 0.001 です。time は、UNIX エポックからの秒数を浮動小数点数型で指定するか、year-month-day Thour:min:sec のフォーマットで指定することができます。同じ指定でミリ秒を付け加える場合は、year-month-dayThour:min:sec.NNN になります。文字 "T" は、デフォルトの Event Stream Processor 時間フォーマットの場合と同様に、リテラルです。 time 値にプレフィクスとして **add** を付けた場合は、現在の時間に加える変更 (浮動小数点数型の

秒数) が指定されます。 コマンドを実行する前のクロックの状態が出力されます。 **clock** コマンドの説明を参照してください。

- **clock real –** Event Stream Processor の論理クロックを実時間に戻します (Event Stream Processor が実行されているマシンのシステム・クロックを使用します)。 停止している間は、クロックを実時間に設定できません。 コマンドを実行する前のクロックの以前の状態が出力されます。**clock** コマンドの説明を参照してください。
- **clock stop on pause *[`0|1`]* –** Event Stream Processor がトレース・モードで一時停止したときに Event Stream Processor の論理クロックを停止するかどうかを制御するフラグを表示または変更します。コマンドを実行する前のクロックの状態が出力されます。**clock** コマンドの説明を参照してください。
- **clear base stream *`stream`* –** 指定された基準ストリームの内容を削除します。
- **datesize –** 接続された Event Stream Processor インスタンスの日付フィールドのサイズ (バイト単位) を出力します (Win32 filetime を使用しているインスタンスの場合は 8 バイト、32 ビット整数の time_t filetime を使用しているインスタンスの場合は 4 バイトになります)。
- **echo *`string`* –** 文字列を標準出力に出力します。
- **echo –** 実行前の標準出力へのコマンドの出力を有効または無効にします。
- **endian –** 接続された Event Stream Processor インスタンスが実行されているマシンのエンディアン値 ("big" または "little") を出力します。
- **fd –** フィールド・デリミタ値を表示します。
- **fd *delimiter* –** 新しいフィールド・デリミタ値を設定します。デリミタを引用符で囲まないでください。スペース以外のすべての文字が新しいデリミタとして取得されます。
- **gateway –** ゲートウェイ・インタフェースのホストとポート番号を出力します。
- **get_config –** 現在実行されている XML 設定を取得して、標準出力に出力するか、コマンドの後に *>file* を指定してローカル・ファイルに出力します。
- **help –** 一般的なヘルプ・メッセージを出力します。
- **help flags –** 一連の出力制御フラグのヘルプを出力します。
- **history *`number`* *[`stream`]* –** ストリームの最大履歴サイズを変更します。トレース・モードがオンの場合にのみ履歴が収集されます。 トレース・モードがオフになるたびに、履歴は廃棄されます。 履歴が収集される場合は、入力トランザクションと出力トランザクションのペアの最後の番号のみが保持され、古い番号は廃棄されます。 Event Stream Processor でのデフォルトの上限は 100 です。
- **history ex *`stream`* –** ストリームの現在の最大履歴サイズを表示します。
- **idx *`streamName`* –** 指定されたストリームのインデックスを出力します。

- **immediate stop –** ストリームをシャットダウンせずに、Event Stream Processor をすぐに停止します。
- **immediate pause –** Event Stream Processor を強制的に一時停止状態にします。 これは、Event Stream Processor が内部でデッドロックまたはフリーズしている場合に検査するための最後の手段です。 このコマンドは Event Stream Processor の状態に影響するため、プロセッサのスタックが解消されない場合にのみ使用してください。
- **kill `*handle*` –** 指定されたハンドルでクライアント接続を強制終了します。メタデータ ストリーム _ESP_Clients に、開いているすべての接続のリストがあります。
- **kill every `*name*` –** 指定されたタグ名でクライアント接続を強制終了します。メタデータ ストリーム _ESP_Clients に、開いているすべての接続とそのタグ名のリストがあります。 タグ名は、**esp_subscribe**、**esp_upload** などのコマンドのオプション `-m` を使用して指定できます。
- **lock timeout *[`seconds`]* –** Event Stream Processor の排他ロック・タイムアウトの値を表示または変更します。大半のコマンド (特定のイベントを待つコマンドを除く) はロックを使用して直列化され、コマンドは 1 つずつ実行されます。タイムアウトを使用することで、エラーが発生した場合に後続のコマンドがこのロックを待って永久にハングすることを防ぎます。 デフォルトのタイムアウトは 60 秒です。
- **loglevel `*level*` –** Event Stream Processor のロギング・レベルを設定します。
- **moneyprecision –** "money" データ型の精度を (小数点以下の桁数で) 出力します。
- **putd `*delimited record*` –** 1 つの区切られたレコード (逆引用符または中カッコで囲まれたもの) をゲートウェイ I/O プロセスに入れます。 「put コマンドに関する注意事項」を参照してください。
- **putx `*XML record*` –** 1 つの XML フォーマットのレコード (逆引用符または中カッコで囲まれたもの) をゲートウェイ・インタフェースに入れます。 「put コマンドに関する注意事項」を参照してください。
- **quiesced –** クワイス状態 (true の場合は 1、false の場合は 0) を出力します。状態が 1 であるのは、パブリッシャ接続がなく、すべての入力データがモデルを介して完全に伝達された場合です。
- **quit – esp_client** ユーティリティを終了します。
- **refresh_calendars –** ファイルからカレンダ・データをリフレッシュします。Event Stream Processor がそのカレンダ機能を介してカレンダをロードしていなければ、このコマンドは何も実行しません。
- **save_config `*remoteFile*` –** 現在実行中の XML 設定を ESP サーバ上のこのファイルに保存します。ファイルはまだ存在しない可能性があるので、引用符でファイル名を囲みます。

- **setparam `*variable*` `*value*` –** 指定された変数を指定された値に設定します。実行中のプロジェクトではパラメータの値を設定できません。
- **settings –** フィールド・セパレータ、フラグ値などを出力します。
- **snapshot *[`streamName`]* –** 出力制御フラグの設定を使用して、ストリームの現在の内容を表形式で出力します。「put コマンドに関する注意事項」を参照してください。
- **start adapters initial –** Event Stream Processor の **ADAPTER START** 文で指定されたすべてのアダプタを起動します。既に実行しているアダプタは実行を継続します。現在実行していないアダプタは再起動されます。このコマンドは、起動されたすべてのアダプタが初期ロードを完了するまで待ちます。
- **start clock –** Event Stream Processor の論理クロックを再開します。コマンドを実行する前のクロックの状態を出力します。 **clock** コマンドの説明を参照してください。
- **start adapter `*adapter-or-group*` – ADAPTER START** 文で名前が指定されたアダプタまたは名前が指定されたグループのすべてのアダプタを起動します。 既に実行しているアダプタには、このコマンドは影響しません。 このコマンドは、アダプタが完全に起動するまで待たず、すぐに戻ります。初期ロードの完了まで待つには **wait connector initial** を使用します。
- **stop – exit streams** コマンドを Event Stream Processor の C & C インタフェースに発行して、Event Stream Processor エンジンを終了させます。
- **stop clock –** Event Stream Processor の論理クロックを停止します。レコードの処理は続行されますが、タイムスタンプは変更されず、タイマ・イベントは発生しません。 クロックが停止している間に、時間とレートを変更することはできますが、クロックを実時間に変更することはできません。 **stop clock** は複数回呼び出すことがありますが、フローを再開するために **start clock** も同じ回数だけ呼び出します。Event Stream Processor は内部でクロックの停止と再開を実行することがありますが、再開は、停止が開始された場合にのみ実行してください。コマンドを実行する前のクロックの状態を出力します。 **clock** コマンドの説明を参照してください。
- **stop adapter `*adapter-or-group*` – ADAPTER START** 文で名前が指定されたアダプタまたは名前が指定されたグループのすべてのアダプタを停止します。 実行していないアダプタには、このコマンドは影響しません。出力アダプタはその出力キューの処理を完了してから停止します (ただし、新しいコマンドはキューに追加されません)。 このコマンドは、アダプタが停止するまで待たず、すぐに戻ります。アダプタの完了まで待つには、**wait connector** を使用します。
- **stop adapter immediate `*adapter-or-group*` – ADAPTER START** 文で名前が指定されたアダプタまたは名前が指定されたグループのすべてのアダプタを停止します。 **stop adapter** と似ていますが、出力アダプタからの出力キューは破棄され、アダプタはすぐに停止することが要求されます。 入力アダプタの場合は、出力キューがないため、**stop adapter** と **stopadapter immediate** は同じです。 この

コマンドは、アダプタが停止するまで待ちません。 アダプタの完了まで待つには、**wait adapter** を使用します。

- **stream `*streamName*` – hdr** と **sphdr** の出力制御フラグを使用して、指定されたストリームの定義を出力します。 「出力制御フラグ」を参照してください。
- **streams –** 基準ストリームと派生ストリームのリストを出力します。
- **throttle `*number*` [`*stream*`] –** 1 つまたはすべてのストリームの入力キュー・スロットル値を変更します。ストリームの入力キューのサイズがスロットル値の 2 倍に達すると、そのキューへの書き込みはブロックされます。 スロットル値をデフォルト値より大きくすることはできません (小さくすることしかできません)。 この値を小さくすると、デバッグ時のレコードのトレースに役立つことがあります。
- **throttle ex `*stream*` –** ストリームの現在のスロットル値を表示します。
- **trace_mode *[on|off]* –** トレース・モードの現在の状態を変更または取得します。引数がない場合は、現在の状態が出力されます。*on* を指定した場合はトレース・モードが有効になり、*off* を指定した場合はトレース・モードが無効になります。 トレース・モードは、シングルステップ、ブレークポイント、デバッグ情報の検査に関連するコマンドを実行する場合の前提条件です。
- **wait adapter `*adapter-or-group*` – ADAPTER START** 文で名前が指定されたアダプタまたは名前が指定されたグループのすべてのアダプタが終了するまで待ちます。 アダプタは自然に終了する場合 (データ・ソース内にデータがなくなった場合) と **stop adapter** の結果として終了する場合があります。
- **wait adapter initial `*adapter-or-group*` – ADAPTER START** 文で名前が指定されたアダプタまたは名前が指定されたグループのすべてのアダプタが初期ロードを完了するまで待ちます。 このコマンドは、アダプタ状態が "initial" 以外の状態に変更されるまで待ちます。
- **wait quiesced –** すべての入力がモデルを介して完全に伝達されるまで待ちます。このコマンドの後で受信された入力は、以前のデータの伝達が完了するまでバッファされます。 その後、Event Stream Processor は通常の操作を再開します。
- **wait quiesced gateway –** すべてのパブリッシュ・クライアントが切断され、すべての入力がモデルを介して完全に伝達されるまで待ちます。このコマンドは、コマンド **quiesced** が 1 を返す場合の条件を待ちます。このコマンドがデータの伝達を待っている間に新しいクライアントが接続すると、これらのクライアントからのデータは、待機が完了するまでバッファされます。

トレース・モードが必要なコマンド

- **pause –** Event Stream Processor の実行を一時停止させます。一時停止が始まると戻ります。 すべてのデータ検査コマンドとシングルステップ・コマンドでは、このコマンドを明示的に指定して、またはブレークポイントまたは不正なデータでの例外で、最初に Event Stream Processor を一時停止しておく必要がありま

す。 Event Stream Processor が既に一時停止していると、このコマンドはすぐに成功を返します。

- **check_pause –** Event Stream Processor が現在一時停止しているかどうかを表示します。
- **wait_pause –** Event Stream Processor がブレークポイントによって、または **esp_client** の別のインスタンスから一時停止されるまで待ちます。 待機は (**esp_client** を終了すること以外では) 中断できません。
- **run –** 通常のプラットフォーム実行を続行します。
- **step *[`stream`]* –** Event Stream Processor が一時停止している場合にシングル・ステップを実行します。ストリーム名が引数として指定されている場合、シングル・ステップはこのストリームで実行されます。 それ以外の場合、ステップ・スルー対象のストリームは、データを処理する準備が整っているストリーム間でランダムに選択されます。 処理する準備が整っているストリームがない場合 (すべてのストリームが入力または出力を待っている場合)、このコマンドは実行されず、すぐに成功を返します。
- **step timeout *[`number`]* –** 自動ステップのタイムアウトをミリ秒単位で設定します。デフォルトのタイムアウトは 0.3 秒です。負の値または 0 の値を使用すると、タイムアウトはデフォルトにリセットされます。
- **step trans *`stream` [`limit`]* –** ストリームを 1 回以上自動的にステップしてから、トランザクションの終了直前 (ストリーム状態図の "PUT" 位置) に移動します。 2 つ目の引数を指定して、実行するステップの数を制限し、大規模なトランザクションの実行時間を抑えます。 デフォルトの制限は 10000 です。ストリームに保留中の入力がない場合や、ストリームが出力でタイムアウトよりもブロックされた場合は、ステップも停止し、エラーが返されます。
- **step quiesce stream *`stream` [`limit`]* –** すべての入力キューが空になるまで、ストリームとそのすべての下位ストリームを自動的にステップします。このコマンドの最初のストリームはリテラルで、2 番目のストリームはパラメータ (ストリーム名) を表します。 2 つ目の引数を指定して、実行するステップの数を制限し、キューに収集された大量のデータの実行時間を抑えます。デフォルトの制限は 100000 です。ストリームが基準ストリームであり、ストリームへの入力が高速であると、呼び出しが戻るのはステップの制限数に達した場合のみになります。 派生ストリームの場合は、これは問題ではありません。Event Stream Processor はステップの実行時に一時停止されるので、このストリームへの入力も一時停止されます。 派生ストリームの入力キューが満杯になり、入力待ちが続いて既に処理済みのデータがそのキューに堆積すると、入力キューが処理される場合に待機中の入力によってトランザクションが追加されます。どのストリームにも保留中の入力がない場合や、どのストリームも出力でタイムアウトより長くブロックされた場合は、ステップも停止し、エラーが返されます。
- **step quiesce downstream *`stream` [`limit`]* – step quiesce stream** と似ていますが、ストリーム自体はステップ・スルーされず、下位ストリームのみがステップ・

スルーされます。 このコマンドは、下位ストリームの入力キューを空にする場合に役立ちます。引数のストリームが出力を生成する場合に、下位ストリーム内のデータの通過を簡単にトレースできます。

- **step quiesce from base** ***[`limit`]*** – 入力キューが空になるまで、すべての派生 (非基準) ストリームを自動的にステップします。 この引数を指定して、実行するステップの数を制限し、キュー内の大量のデータの実行時間を抑えます。 デフォルトの制限は 100000 です。どのストリームにも保留中の入力がない場合や、どのストリームも出力でタイムアウト値より長くブロックされた場合は、ステップも停止し、エラーが返されます。 このコマンドは、基準ストリームからの整合性のないレコードを処理する前に派生ストリームのキューを空にしておく場合に役立つ場合があります。 派生ストリーム内のデータの通過を簡単に監視できます。
- **dump** ***`filePrefix` [`stream`]*** – 各ストリームまたは 1 つの特定のストリームの内容をファイルにダンプします。各ファイルには名前 `filePrefixdump_streamName.xml` が付けられます。
- **bp add** ***`stream` `inputStream` [`condition`]*** – ストリームにブレークポイントを追加してから、別のストリーム (*inputStream*) からの入力レコードの処理を開始します。 必要に応じて、SPLASH 式を使用して、ブレークポイントをトリガする条件を指定します。 ブレークポイントは、入力レコードで評価される条件が true の場合にのみトリガされます。 この式は、次の事前定義変数のどちらか一方または両方を参照することがあります。
  - **currow** – 現在の入力レコード。
  - **oldrow** – 更新対象または削除対象の、このキーを持つレコードの以前の値。

  条件は、レコードのフィールド *"row.field"* を参照することがあります。また、ストリームのローカル変数とグローバル変数も使用されることがあります。 このコマンドは、新たに作成されたブレークポイントの ID を出力します。
- **bp add** ***`stream` any*** – ストリームにブレークポイントを追加してから、任意のストリームからの入力レコードの処理を開始します。 条件は指定できません。

  このコマンドは、新たに作成されたブレークポイントの ID を出力します。
- **bp add** ***`stream` out [`condition`]*** – ストリームが入力レコードを処理して、何らかの (おそらく空の) 出力を生成してから、そのストリームにブレークポイントを追加します。 必要に応じて、SPLASH 式を使用して、ブレークポイントをトリガする条件を指定します。 ブレークポイントは、入力レコードで評価される条件が true の場合にのみトリガされます。 式は、1 つの事前定義変数 *currow* を参照することがあります。この変数は現在の出力レコードです。1 つの入力レコードから複数の出力レコードが生成される場合があるので、条件は出力レコードごとに順に評価されます。 出力が生成されなかった場合も、*currow* は NULL に設定されて、条件は 1 回評価されます。条件はレコードのフィールド *"row.field"* を参照することがあります。

このコマンドは、新たに作成されたブレークポイントの ID を出力します。

- **bp del `*id*`** – 指定された ID (次のコマンドで返される値：**bp add** または **bp list)** を持つブレークポイントを削除します。
- **bp del all** – すべてのブレークポイントを削除します。
- **bp on|off `*id*`** – 指定された ID を持つブレークポイントを有効または無効にします。
- **bp on|off all** – すべてのブレークポイントを有効または無効にします。
- **bp every `*count*` `*id*`** – 指定された ID を持つブレークポイントを n 回目ごとにトリガします。たとえば、ID 8 を持つブレークポイントを 100 番目のレコードごとにトリガするには、"**bp every** `100` `8`" を使用します。 カウントを 1 に設定すると、ブレークポイントは 1 つのレコードごとにトリガされます。
- **bp every `*count*` *all*** – すべてのブレークポイントを N 回目ごとにトリガします。
- **bp list** – ブレークポイントをリストします。**"ex `breakpoints`"** の代わりに使用できます。
- **ex `*kind*` [`*stream*` [`*object*`]]** – Event Stream Processor 内のデータを検査します。**ex** は、データの種類の名前、データが属するストリームの名前、特定のオブジェクトの名前を取ります。 データの種類によっては、ストリームとオブジェクトの引数が適用されないことがあります。 データは XML フォーマットで出力され、ほとんどのデータの種類の要素名は "row" に設定されます。 データがトランザクションを表す場合は、*<trans>* 要素で囲みます。データが更新ペアを表す場合は、*<pair>* 要素で囲みます。 正確なフィールドは検査対象のデータによって異なります。

  入力データの種類 (入力キュー、現在の入力トランザクションとロー、入力履歴) を検査すると、データはさまざまなストリームで生成された異なる種類のローの混合であることがあります。 XML 要素の名前は、それを生成したストリームの名前に設定されます (基準ストリームでは、これは基準ストリーム自体の名前になります)。

  現在サポートされているデータの種類は、次のとおりです。

  - **`pause`** – 一時停止されている場合のユーザ・ストリームの状態。フィールドは次のとおりです。
    - **name** – ストリームの名前。
    - **loc** – ストリームが一時停止されている場所。
    - **onbp** – トリガされたブレークポイント上の場合は、そのブレークポイントの ID。それ以外の場合、**onbp** は 0 に設定されます。複数のブレークポイントが同時にトリガされた場合、**onbp** には、いずれかのブレークポイントの ID が入ります。
    - **throttle** – 入力キューのスロットル値。

    - **history** – 保持される履歴の最大サイズ。
    - **postSeq** – 入力キューにポストされたトランザクションの数。
    - **inSeq** – 入力キューから読み取られたトランザクションの数。
    - **outSeq** – 出力に処理されたトランザクションの数 (破棄される空のトランザクションと有効期限切れのトランザクションを含む)。
    - **stepSeq** – トレース・モードで実行された (**step** コマンドで定義された) コマンドの数。 これには、シングルステップと実行の両方が含まれます。この数の変化を使用して、状態が変化したストリームを判断します。
- **`pauseall`** – **pause** と同じですが、メタデータ・ストリームも含まれます。
- **`breakpoints`** – 現在登録されているすべてのブレークポイントに関する情報。 フィールドは次のとおりです。

    - **id** – ブレークポイントの ID。 ブレークポイントの有効期間の間は変更しないでください。
    - **stream** – ブレークポイントを定義するストリームの名前。
    - **origin** – 特定の入力ストリーム上のブレークポイントに対する入力ストリームの名前。 任意のストリームからの入力上のブレークポイントの場合は "*" を使用し、出力上のブレークポイントの場合は "" (空) を使用します。
    - **expr** – 条件式。
    - **enabledEvery** – n 番目の一致レコードごとにブレークポイントをトリガするための n。「**bp every**」を参照してください。
    - **leftToTrigger** – ブレークポイントがトリガされるまで現在あといくつの一致が残っているかを示す数。
    - **onit** – ブレークポイントが現在トリガされている場合は 1、それ以外の場合は 0。
- **`var`** **`*var-name*`** – グローバル変数 (グローバルの **DECLARE** ブロックで定義されている変数) の内容。 フィールドは変数のタイプによって異なります。配列内のインデックスは、**ESP_Index** として表示されます。 辞書内のキーは、 **ESP_Key_<field-name>** として表示されます。 レコード定義の場合のように、レコードの値はフィールドと一緒に表示されます。単純な変数は **ESP_Value** フィールドを使用して表されます。 構造化された値では、このコマンドは複数のローを返すことがあります。 変数が NULL の場合は、何も返されません。配列では、NULL 以外の値を持つ要素のみが表示されます。
- **`listVar`** – すべてのグローバル変数名のリスト。

    - **name** – 変数の名前。
    - **type** – 変数のタイプ。

- **`store`** **`*stream*`** – ストリームのストアの内容。 フィールドは、ストリームのロー定義の場合と同じです。
- **`outTrans`** **`*stream*`** – 作成されている、現在の出力トランザクション。 フィールドは、ストリームのロー定義の場合と同じです。
- **`outRow`** **`*stream*`** – 以前の入力ローの処理から生成された出力。複数のローがある場合もあれば、ローがない場合もあります。 フィールドは、ストリームのロー定義の場合と同じです。
- **`badRows`** **`*stream*`** – Event Stream Processor が不正なローの例外で一時停止された場合は、これらの不正なローが入ります。 フィールドは、データを生成したストリーム (または、基準ストリームの場合は、現在のストリーム) のロー定義の場合と同じです。
- **`badRowsReason`** **`*stream*`** – **badRows** で報告される不正なローごとに、このデータには不正である理由を説明するエラー・メッセージが入ります。 メッセージは reason フィールドに入ります。
- **`outHist`** **`*stream*`** – ストリームの履歴からの出力トランザクション。 レコードのすべてのフィールドに NULL が設定されている場合は、空のトランザクションが返されます。 **ex `outHist`** から返されたトランザクションと **ex `inHist`** から返されたトランザクションとの 1 対 1 の一致があります。 フィールドは、ストリームのロー定義の場合と同じです。
- **`lastOutTrans`** **`*stream*`** – ストリームの履歴での最新の出力トランザクション。 **" ex `outHistLatest` `stream` `0`"** と似ていますが、履歴が空の場合はローなしで成功が返されるのに対し、**outHistLatest** の場合はエラーが返されます。 フィールドは、ストリームのロー定義の場合と同じです。
- **`outHistEarliest`** **`*stream*`** **`*index*`** – ストリームの履歴から個々の出力トランザクションを選択します。インデックスは番号で、0 を指定すると、履歴に保存されている最も古いトランザクションが選択されます。インデックスの番号が大きくなるほど、新しいトランザクションであることを示します。 このようなインデックスを持つトランザクションがない場合は、"No such object" エラーが返されます。 フィールドは、ストリームのロー定義の場合と同じです。
- **`outHistLatest`** **`*stream*`** **`*index*`** – ストリームの履歴から個々の出力トランザクションを選択します。インデックスは番号で、0 を指定すると、履歴に保存されている最新のトランザクションが選択されます。インデックスの番号が大きくなるほど、古いトランザクションであることを示します。 このようなインデックスを持つトランザクションがない場合は、"No such object" エラーが返されます。 フィールドは、ストリームのロー定義の場合と同じです。
- **`var`** **`*stream*`** **`*var-name*`** – ストリームのローカル変数の内容。 ローカル変数のみ (ストリームのローカル **DECLARE** ブロックで定義されている変数) のみを検査できます。ローカル変数には、配列、辞書、eventCache の変数が含

まれます。 SPLASH ブロックの内側で定義されている変数は、該当するメソッドが実行される場合にのみ存在するので、検査できません。 フィールドは変数のタイプによって異なります。配列と eventCache 内のインデックスは **ESP_Index** として表示されます。辞書と eventCache 内のキーは **ESP_Key_<field-name>** として表示されます。 レコードの値は、レコード定義の場合のようにフィールドと一緒に表示されます。単純な変数は、**ESP_Value** フィールドを使用して表されます。 構造化された値では、 **var** は複数のローを返すことがあります。変数が NULL の場合は、何も返されません。 配列の場合は、NULL 以外の値を持つ要素のみが表示されます。 この方法では、グローバル変数にアクセスできません。グローバル変数にアクセスするには、代わりに空のストリーム名を使用します。

- **`listVar`** **`*stream*`** – このストリームで定義されている使用可能なすべての変数名のリスト。ローカルの **DECLARE** ブロックを持つことができるストリームにのみ適用されます。 グローバル変数は含まれません。
  - **name** – 変数の名前。
  - **type** – 変数のタイプ。
- **`queue`** **`*stream*`** – 入力データの種類。ストリームの入力キューの内容。フィールドは、データを生成したストリーム (または、基準ストリームの場合は、現在のストリーム) のロー定義の場合と同じです。
- **`inTrans`** **`*stream*`** – 入力データの種類。 処理中の現在の入力トランザクション。 フィールドは、データを生成したストリーム (または、基準ストリームの場合は、現在のストリーム) のロー定義の場合と同じです。
- **`inRow`** **`*stream*`** – 入力データの種類。 処理中の現在の入力ロー。フィールドは、データを生成したストリーム (または、基準ストリームの場合は、現在のストリーム) のロー定義の場合と同じです。
- **`queueHead`** **`*stream*`** **`*index*`** – 入力データの種類。 ストリームの入力キューから個々のトランザクションを選択します。 インデックスは番号で、0 を指定するとキューの先頭にあるトランザクションが選択されます。インデックスの番号が大きいほど、後続のトランザクションであることを示します。 このようなインデックスを持つトランザクションがない場合は、"No such object" エラーが返されます。 フィールドは、データを生成したストリーム (または、基準ストリームの場合は、現在のストリーム) のロー定義の場合と同じです。
- **`queueTail`** **`*stream*`** **`*index*`** – 入力データの種類。 ストリームの入力キューから個々のトランザクションを選択します。 インデックスは番号で、0 を指定するとキューの末尾にある最後のトランザクションが選択されます。インデックスの番号が大きいほど、前のトランザクションであることを示します。 このようなインデックスを持つトランザクションがない場合は、"No such object" エラーが返されます。 フィールドは、データを生成し

たストリーム (または、基準ストリームの場合は、現在のストリーム) のロー定義の場合と同じです。

- **`inHist` `*stream*`** – 入力データの種類。 ストリームの履歴からの入力トランザクション。**ex `outHist`** から返されたトランザクションと **ex `inHist`** から返されたトランザクションとの 1 対 1 の一致があります。 フィールドは、データを生成したストリーム (または、基準ストリームの場合は、現在のストリーム) のロー定義の場合と同じです。
- **`lastInTrans` `*stream*`** – 入力データの種類。 ストリームの履歴内の最新の入力トランザクション。 **"`inHistLatest` `stream` `0`"** と似ていますが、履歴が空の場合はローなしで成功が返されるのに対し、**`inHistLatest`** の場合はエラーが返されます。 フィールドは、データを生成したストリーム (または、基準ストリームの場合は、現在のストリーム) のロー定義の場合と同じです。
- **`inHistEarliest` `*stream*` `*index*`** – 入力データの種類。 ストリームの履歴から個々の入力トランザクションを選択します。 インデックスは番号で、0 を指定すると履歴に保存されている最も古いトランザクションが選択されます。インデックスの番号が大きいほど、新しいトランザクションであることを示します。 このようなインデックスを持つトランザクションがない場合は、"No such object" エラーが返されます。 フィールドは、データを生成したストリーム (または、基準ストリームの場合は、現在のストリーム) のロー定義の場合と同じです。
- **`inHistLatest` `*stream*` `*index*`** – 入力データの種類。 ストリームの履歴から個々の入力トランザクションを選択します。 インデックスは番号で、0 を指定すると履歴に保存されている最新のトランザクションが選択されます。インデックスの番号が大きいほど、古いトランザクションであることを示します。 このようなインデックスを持つトランザクションがない場合は、"No such object" エラーが返されます。 フィールドは、データを生成したストリーム (または、基準ストリームの場合は、現在のストリーム) のロー定義の場合と同じです。
- **`hist` `*stream*`** – 入力データと出力データの両方を含む、履歴の混合表現。 各入力トランザクションの後に、それと一致する出力トランザクションが続きます。 入力トランザクションのローには、元のストリーム名の XML タグが付き、出力トランザクションのローには、XML タグ "row" がつきます。
- **`lastTrans` `*stream*`** – 入出力混合のデータ種類。 **「`hist`」**を参照してください。 ストリームの履歴内の最新の入力トランザクションと出力トランザクション。
- **`histEarliest` `*stream*` `*index*`** – 入出力混合のデータ種類。 **「`hist`」**を参照してください。 ストリームの履歴から個々のトランザクション・ペアを選択します。 インデックスは番号で、0 を指定すると履歴に保存されている最も古いトランザクションが選択されます。インデックスの番号が大きいほど、新しいトランザクションであることを示します。 このようなインデッ

クスを持つトランザクションがない場合は、"No such object" エラーが返されます。

- **`histLatest`** ***`stream`*** ***`index`*** – 入出力混合のデータ種類。 「**`hist`**」を参照してください。ストリームの履歴から個々のトランザクション・ペアを選択します。 インデックスは番号で、0 を指定すると履歴に保存されている最新のトランザクションが選択されます。インデックスの番号が大きいほど、古いトランザクションであることを示します。 このようなインデックスを持つトランザクションがない場合は、"No such object" エラーが返されます。
- **`aggrGroup`** ***`aggregationStream`*** – グループ・インデックスからの、集約ストリームの内部状態。加法的集約を使用するように最適化されていない集約ストリームでのみ動作します。 値フィールドには、入力ストリームのロー定義からの名前が割り当てられます。 キー・フィールドには、このストリームのロー定義と同じ名前が割り当てられますが、**ESP_Key_** プレフィクスは付きません。 集約バケット内のレコードのインデックスは **ESP_Index** フィールドに入ります。
- **`states`** ***`patternStream`*** ***[`patternNum`]*** – パターン・ストリームでのオートマトンの状態。 初期設定では、パターン・ストリームには定義されたパターンごとに 1 つのオートマトンがあります。データが受信され、パターンを基準とする一致が行われると、パターンと一致する可能性があるイベントのシーケンスごとに、新しいオートマトンのクローンが作成されます。 完全なパターンが検出されるか、イベントのシーケンスがパターンと一致しないことが検出されると、オートマトンは破棄されます。

  オプション・パラメータ *patternNum* がある場合、**states** ではそのパターンのオートマトンのみが表示されます。 フィールドは次のとおりです。

  - **pnum** – このオートマトンによって解析されるパターンの番号 (0 から始まります)。
  - **instance** – オートマトンのインスタンス番号。新しいオートマトンのクローンが作成されると、各オートマトンはユニークなインスタンス番号を受け取ります。 インスタンス番号は (Event Stream Processor が再起動されない限り) 繰り返されません。 オートマトンの実行中は、ペア (pnum, instance) でオートマトンを識別します。
  - **state** – オートマトンの現在の状態を識別する数値。 オートマトンは線形です。その論理でループは使用されず、1 つの状態にアクセスできるのは 1 回のみです。 最後の検査から状態が変化していない場合は、オートマトンが新しいデータと一致しなかったことを示します。 状態に使用される数値は順次ではありません。
  - **timed** – 1 に設定すると、有効期限タイマがこのオートマトンにアタッチします。 タイマの有効期限が切れると、パターン一致は失敗したと見なされ、オートマトンは破棄されます。 0 に設定すると、オートマトンの有効期限は設定されません。 無期限のオートマトンは、パターンの非常

に初期のオートマトンで、その他のすべてのオートマトンのクローン作成に使用されます。

- **time_left –** 期限付きオートマトンの有効期限までの時間 (秒単位)。無期限のオートマトンでは、これは常に 0 です。デフォルトでは、Event Stream Processor が一時停止すると、プラットフォームの論理クロックが停止します。 ただし、一時停止時に停止しないようにクロックが設定されていれば、タイマは停止しません。 0 または負の値は、Event Stream Processor の実行が再開するとオートマトンの有効期限が切れることを意味します。

- **`bindings` `*patternStream*` `[*patternNum*]` –** *patternStream* における各オートマトンの、これまでに解析されたデータによって生成されたパターン変数バインド。 オプション・パラメータ *patternNum* がある場合は、そのパターンのデータのみが表示されます。 フィールドは次のとおりです。

  - **pnum –** このオートマトンによって解析されるパターンの番号 (0 から始まります)。
  - **instance –** オートマトンのインスタンス番号。 新しいオートマトンのクローンが作成されると、各オートマトンはユニークなインスタンス番号を受け取ります。 インスタンス番号は、Event Stream Processor が再起動されない限り、繰り返されません。 オートマトンの実行中は、**pnum**, **instance** でオートマトンを識別します。
  - **var –** バインドされている変数の名前。 これらの変数の他に、バインドされているイベントと定数もリストされます。 定数は、ユニークなコンパイラ生成の名前で表示されます。
  - **value –** バインドされている変数の値 (文字列フォーマット)。 ここでは、バインドされたイベントの値は NULL としてレポートされます。 イベント・データの種類を検査してイベント・ローの内容を確認します。

- **`events` `*patternStream*` `[*patternNum*]` –** パターン・ストリームにおける各オートマトンの、これまでにそのオートマトンによって解析されたイベント。 このデータ種類では、異なるタイプのレコードの混合が返されます。 レコードのタイプには、フロー元の入力ストリームの名前が付けられます。

  オプション・パラメータ *patternNum* がある場合は、そのパターンのデータのみが表示されます。 フィールドは次のとおりです。

  - **ESP_Pnum –** このオートマトンによって解析されるパターンの番号 (0 から始まります)。
  - **ESP_Instance –** オートマトンのインスタンス番号。新しいオートマトンのクローンが作成されると、各オートマトンはユニークなインスタンス番号を受け取ります。 インスタンス番号は、Event Stream Processor が再

起動されない限り、繰り返されません。 オートマトンの実行中は、**pnum, instance** でオートマトンを識別します。

- **ESP_Var –** イベント変数の名前。
- **as in input stream –** その他のフィールドは、フロー元の入力ストリームのロー・タイプと同じ名前を保持します。

- **`expect` `*patternStream*` `[*patternNum*]` –** パターン・ストリームにおけるオートマトンごとの、オートマトンを次の状態に移行すると予想されるレコード。 このデータ種類では、異なるタイプのレコードの混合が返されます。 レコードのタイプには、フロー元の入力ストリームの名前が付けられます。

  オプション・パラメータ *patternNum* がある場合は、そのパターンのデータのみが表示されます。 フィールドは次のとおりです。

  - **ESP_Pnum –** このオートマトンによって解析されるパターンの番号 (0 から始まります)。
  - **ESP_Instance –** オートマトンのインスタンス番号。新しいオートマトンのクローンが作成されると、各オートマトンはユニークなインスタンス番号を受け取ります。 インスタンス番号は、Event Stream Processor が再起動されない限り、繰り返されません。 オートマトンの実行中は、**pnum, instance** でオートマトンを識別します。
  - **ESP_Var –** イベント変数の名前。 先頭に "!" が付いているレコードを受け取ると、パターンの不一致になります。 それ以外の場合、オートマトンは次の状態に移行します。
  - **as in input stream –** その他のフィールドは、フロー元の入力ストリームのロー・タイプと同じ名前を保持します。 値にバインドされているフィールドのみが表示されます。その他のフィールドは NULL と表示されます。

- **exf `*kind*` `[*stream*` `[*object*]] `*filter*` – ex** と似ていますが、Event Stream Processor で評価するフィルタ SPLASH 式を指定します。 フィルタが true (0 以外、NULL 以外) 値に評価するレコードのみが返されます。 フィルタを指定すると、トランザクション・ペア境界と更新ペア境界は失われます。各レコードは単独で戻ります。

  フィルタは、出力時にローの XML タグと一致する名前を持つ事前定義変数を参照することがあります。 大半のデータ種類では、変数のローにはフィルタリング対象の現在のレコードがあります。 入力データの種類では、複数の変数が定義され、各変数にはターゲット・ストリームの入力ストリームの名前が付けられます。この場合にレコードを評価すると、送信元のストリームと一致する変数にそのレコードがあり、その他のすべての変数は NULL に設定されます。 条件は、**"currow.field"** のように、通常どおりレコード内のフィールドを参照することがあります。

- **eval `*stream*` `*block*`** – ストリームで SPLASH 文 (式ではない) を評価して、ストリームのローカル変数 (ローカルの **DECLARE** ブロックまたはグローバルの **DECLARE** ブロックで定義されている変数) の内容を変更します。 ストリームの SPLASH ブロックの内側で定義されている変数は、該当するメソッドが実行される場合にのみ存在するので、変更できません。 グローバル変数の変更には、計算ストリームのコンテキストでの評価が使用されます。

  SPLASH 文は、";" で終わる単純な文か、中カッコ "{}" で囲まれたブロックにしてください。複数の文は必ず 1 つのブロックで囲みます。 中カッコを使用してブロック引数を囲む場合、外側の中カッコはブロック・デリミタと見なされません (単なる **esp_client** の引用符です)。

正しい例：

```
`a := 1;`
    {a := 1;}
    `{ typeof(input) r := [ a=9; |
    b= 's1'; c=1.; d=intDate(0);];
    keyCache(s0, r); insertCache(s0, r); }`
    {{ typeof(input) r := [ a=9; |
        b= 's1'; c=1.; d=intDate(0);];
    keyCache(s0, r); insertCache(s0, r); }}
```

間違った例：

```
`a := 1`
    {a := 1}
    `typeof(input) r := [ a=9; |
    b= 's1'; c=1.; d=intDate(0);];
    keyCache(s0, r); insertCache(s0, r);`
    { typeof(input) r := [ a=9; |
        b= 's1'; c=1.; d=intDate(0);];
    keyCache(s0, r); insertCache(s0, r); }
```

ブロック内でテンポラリ変数を定義する場合の構文を含め、通常の SPLASH 構文はすべて適用されます。 ストリームの変数とグローバル変数はすべて参照できるので、文で確認したり変更したりできます。ストリームとストリーム反復子は文で参照できません。

複数行の文を入力する場合は、逆引用符も中カッコも使用できません。前述の例では、行を分けることで端末での行のラッピングを表しています。 一般的には、次のように複数行の引用符付きフォーマットの方が便利です。

```
eval {stream} <<!
    { typeof(input) r := [ a=9; |
    b= 's1'; c=1.; d=intDate(0);];
    keyCache(s0, r); insertCache(s0, r); }
!
```

eventCache でのオペレーションには特別な準備が必要です。通常、eventCache のキーは現在の入力レコードによって決まります。この例では、入力レコード

がないので、キーは設定されず、eventCache でのオペレーションは効果がありません。 オペレーションを機能させるには、eventCache で集約オペレーションを実行する前に、演算子 **keyCache(ec-variable, record)** を使用してキーを手動で設定します。

出力制御フラグ

- **hdr *[on|off]*** – 引数がない場合は、"include column name header line" フラグの状態が表示されます。それ以外の場合は、"include column name header line" フラグを有効または無効にします。 **hdr** が有効な場合、表形式のデータの前にカラム名の見出しが出力されます。スナップショットの場合は、フィールドの位置、名前、フィールド・タイプが表示されます。 フィールドがキー・フィールドの場合は、フィールド名の前にアスタリスク ("*") が付きます。
- **sphdr *[on|off]*** – 引数がない場合は、"include header/data prefix" フラグの状態が表示されます。それ以外の場合は、"include header/data prefix" フラグを有効または無効にします。 **sphdr** が有効な場合は、表形式のスナップショット・データの各行の前に StreamName 値と opcode 値が出力されます。 さらに、**hdr** フラグが有効な場合は、ヘッダ行に Event Stream Processor の [StreamName] フィールド名と [opcode] フィールド名が含まれます。 Event Stream Processor のヘッダ／プレフィクスは **esp_client** の **putd** コマンドと **putx** コマンドです。
- **txb *[on|off]*** – 引数がない場合は、"include TRANSACTION BLOCK content" フラグの状態が表示されます。それ以外の場合は、"include TRANSACTION BLOCK content" フラグを有効または無効にします。 **txb** が有効な場合は、スナップショット・コマンドで生成される出力に、ゲートウェイ I/O トランザクション・ブロック内にあるすべてのメッセージが含まれます。 **txb** が無効な場合は、トランザクション・ブロックの INSERT メッセージのみを使用して、表形式のスナップショット出力が生成されます。
- **verbose *[on|off]*** – 引数がない場合は、"verbose" 出力フラグの状態が表示されます。それ以外の場合は、"verbose" 出力フラグを有効または無効にします。 **verbose** が有効な場合は、スナップショット・コマンドの処理時にこのフラグによって追加出力が生成されます。 特に、start_sync、end_sync、トランザクション・ブロック・インジケータ、最後のスナップショット、レコード／ロー数が生成されます。
- **xml *[on|off]*** – 引数がない場合は、"XML" 出力フラグの状態が表示されます。それ以外の場合は、"XML" 出力フラグを有効または無効にします。 **xml** が有効な場合は、スナップショット・コマンドによって生成される出力は ESP XML レコード・フォーマットになります。 XML フォーマットは、**putx** コマンドで使用されます。

put コマンドに関する注意事項

putd コマンドと **putx** コマンドは、**sphdr** の [StreamName] と [OpCode] のプレフィクスを使用します。日付文字列のフォーマットは次のとおりです。

%Y-%m-%dT%H%M%S

TZ 環境変数が "UTC" に設定されてから、レコードがゲートウェイ・インタフェースにアップロードされます。

使用法についての注意事項

コンソール・モードでは、コンソールから、コマンド・ラインの編集とコマンド履歴の取得を実行できるコマンドを発行できます。 このモードに入るには、次のコマンドを使用します。

```
esp_client -p localhost:19011/default/prj1 -c user:password
```

コマンド文字列モードでは、1 つ以上のコマンドを含む二重引用符付き文字列を入力します。二重引用符付き文字列で複数のコマンドを指定する場合は、各コマンドの終わりにセミコロン文字を挿入します。 コマンド・ラインからフィールド・セパレータを設定する場合は、新しいセパレータ文字を一重引用符で囲み、終了一重引用符とセミコロンの間にスペース文字を挿入します。

第 4 章 パブリッシュとサブスクライブの実行可能プログラム

コマンド・ライン・ユーティリティを使用して、Event Stream Processor のデータのサブスクライブ、変換、パブリッシュを実行します。

esp_iqloader

1 つ以上の Event Stream Processor ストリームから Sybase IQ にデータをアーカイブします。

esp_iqloader コマンドは、リアルタイム・モードまたはバッチ・モードのどちらかで 1 つ以上の Event Stream Processor ストリームから Sybase IQ にデータをアーカイブします。

esp_iqloader は、アーカイブを必要とするストリームをサブスクライブしてから、挿入の場合はバルク・ロードに適した区切りフォーマットで、更新と削除の場合は SQL DML 文として、データを中間ファイルに書き込みます。**esp_iqloader** では、挿入は Sybase IQ の **LOAD TABLE** 文機能を使用してアーカイブされ、更新と削除は ODBC を介してアーカイブされます。

注意： データ・ウェアハウス・モード (`-I` 引数) で **esp_iqloader** を実行している場合、更新は挿入として処理され、削除は無視されます。

エラーが検出されると、**esp_iqloader** は終了します。**esp_iqloader**、Event Stream Processor、または Sybase IQ が停止すると、再起動時に、**esp_iqloader** は最後にアーカイブされたトランザクションからデータのアーカイブを開始します。**esp_iqloader** は、最新のアーカイブされたトランザクションを追跡します。

注意： アーカイブ対象の各ストリームが永続サブスクライブ・パターンを使用していることを確認してください。各ストリームを 2 つの別のストリームに関連付けます。1 つ目のストリームはログ・ストリームで、アーカイブするストリームのトランザクション・ログがあります。2 つ目のストリームはログ・ストリームへの入力である制御ストリームです。これは、ログ・ストリームへのゲートウェイの役目を果たし、アーカイブされたトランザクション・ログを消去します。永続サブスクライブ・パターンを使用するログ・ストリームと制御ストリームの両方とも、ESP スタジオを使用して作成します。

制御ストリームとログ・ストリームは、Event Stream Processor で障害が発生した場合にトランザクションを失わないように永続化します。 データ・ストリームが自分自身を確実に再生成できるようにするには、制御ストリームとログ・ストリームの両方を格納するログ・ストアを 1 つ以上作成します。

実行中の **esp_iqloader** プロセスを停止するには、コマンド・ラインで [Ctrl] キーを押しながら [C] キーを押します。

制限事項と既知の問題：
- 更新と削除をアーカイブする場合、**esp_iqloader** は Event Stream Processor に追いつけないことがあります。これは、個々の更新と削除の SQL 文は ODBC を介して適用されて、データをアーカイブするためです。

構文

```
esp_iqloader -f configFile -p [<host>:]<port>/workspace-name/
project-name
```

必須の引数
- **`-f` *configFile*** – (必須) アーカイブ対象のストリーム、Sybase IQ への接続情報、Sybase IQ の LoadTable オプション、その他の情報が記述されている XML スタイルの設定ファイルを指定します。
- **`-p` *[<host>:]<port>/workspace-name/project-name*** – (必須) *host:<port>/<workspace name>/ <project name>* のすべての引数で、ESP サーバ (クラスタ・マネージャ) に接続する場合の URI を指定します。たとえば、ESP クラスタ・サーバを起動したときのポートが [19011] で、ホスト名が [localhost] として設定され、デフォルトのワークスペースで [prj1] プロジェクトを実行している場合は、`-p` を次のように指定します。 `-p localhost:19011/default/prj1`

オプション
- **`-b`** – (省略可能) **byteswap** モードを設定します。**esp_iqloader** を実行しているサーバは、バイト順序が異なるマシンに接続できます。

  注意： **esp_iqloader** を使用して、バイト順序のアーキテクチャが異なるマシンには接続できますが、アドレス・サイズが異なるマシンには接続できません。
- **`-B` *batchsize*** – (省略可能) ODBC と SQL のデータ操作文を使用してデータをアーカイブする場合のコミット・バッチ・サイズを指定します。 このオプションの適正な設定を選択します。 バッチ・サイズが小さすぎると、小さい中間ファイルが数多く生成されて、アーカイブの効率が低下します。 バッチ・サイズが大きすぎると、データベースにデータをコミットするときに問題が発生することがあります。デフォルト値は 1000 です。

  注意： *batchsize* では、バルク・ロード・トランザクションのコミット・バッチ・サイズは変更されません。この変更を行うには、**configFile** で

"LoadStatement" に適切なオプションを指定します。 バルク・ロード・トランザクションのデフォルトのバッチ・サイズ値は 100,000 です。

- **-c** ***user[:password]*** **–** (省略可能) 認証のクレデンシャルを Event Stream Processor に渡します。パスワードが指定されていないと、パスワードの入力を求めるプロンプトが表示されます。 Event Stream Proccessor でこれらのクレデンシャルを使用して認証が正常に行われた場合は、接続が維持されます。 それ以外の場合は、Event Stream Proccessor はすぐに接続を切断します。この引数が、Event Streaming Processor の起動時に指定した認証のタイプに対応していることを確認してください。
- **-d –** (省略可能) **esp_iqloader** が中間ファイルを作成する場合に使用するデリミタを指定します。 アーカイブ対象のデータに使用されているデリミタは選択しないでください。 このようなデリミタを選択すると、アーカイブされたデータが壊れたり、バルク・ロード実行可能プログラムで拒否されたりします。 デフォルトのデリミタ値は HEX 31 です。
- **-D –** (省略可能) デルタ・モードを有効にします。指定されていないと、**esp_iqloader** は、指定されたストリームの既存のすべてのデータをアーカイブして、終了します。
- **-e –** (省略可能) OpenSSL ソケットを介する暗号化を有効にします。
- **-k** ***privateRsaKeyFile*** **–** (省略可能) パスワード認証の代わりに、RSA プライベート・キー・ファイルのメカニズムを使用して認証を実行します。 *privateRsaKeyFile* には、プライベート RSA キー・ファイルのパス名を指定します。

  注意： ESP サーバの起動時に `-k` オプションで RSA キーの保存先のディレクトリを指定したことを確認してください。
- **-l** ***linedelimiter*** **–** (省略可能) **esp_iqloader** の 1 文字のデリミタを指定します。 **esp_iqloader** が中間ファイルを作成するときに行の終わりをマークします。デフォルトでは、**esp_iqloader** は改行文字をデリミタとして使用します。

  注意： アーカイブ対象のデータに使用されているデリミタは選択しないでください。 このようなデリミタを選択すると、アーカイブされたデータが壊れたり、バルク・ロード実行可能プログラムで拒否されたりします。
- **-p** ***[<host>:]<port>/workspace-name/project-name*** **–** (必須) *host:<port>/<workspace name>/ <project name>* のすべての引数で、ESP サーバ (クラスタ・マネージャ) に接続する場合の URI を指定します。たとえば、ESP クラスタ・サーバを起動したときのポートが [19011] で、ホスト名が [localhost] として設定され、デフォルトのワークスペースで [prj1] プロジェクトを実行している場合は、`-p` を次のように指定します。 `-p localhost:19011/default/prj1`

- **`-P` *precision*** – 浮動小数点数の小数点以下の桁数を定義する 0 ～ 6 の間の整数値を指定します。 デフォルト値は 6 です。
- **`-q` *queueSize*** – (省略可能) 最大サブスクリプション・バッファ・サイズを設定します。データを大量に受信する場合やサブスクリプション・バッファが満杯であることが Event Stream Proccessor から報告された場合は、このパラメータの値を大きくします。 最小値は 1001 です。デフォルト値は 8000 です。

  注意： 最大バッファ・サイズを設定する場合は、設定ファイルの Block Size パラメータが 12000 以下に設定されていることを確認してください。
- **`-R`** – (省略可能) リカバリ・モードを有効にします。このモードでは、**esp_iqloader** は Event Stream Processor に接続せず、これ以上データを取得しません。 代わりに、**esp_iqloader** が中断されたため、アーカイブされたデータではなく、以前にディスクに書き込まれたデータをアーカイブします。 データのアーカイブが完了すると、終了します。

  注意： リカバリ・モードを有効にすると、`-D` 引数は無視されます。
- **`-T` *interval*** – (省略可能) リアルタイム・モードでアーカイブする場合に **esp_iqloader** がデータの差分をアーカイブするおおよその間隔を指定します。間隔は、前回の差分アーカイブの操作が完了した時間を基準とします。
- **`-W` *wait_time*** – (省略可能) **esp_iqloader** が基本データを消費するまで Event Stream Processor が待つ時間をミリ秒単位で指定します。最小値は 1000 です。デフォルト値は 30000 です。
- **`-v`** – (省略可能) **esp_iqloader** ユーティリティのバージョンを出力します。

esp_iqloader 環境の設定

次に、必要な **esp_iqloader** 環境コンポーネントの設定を示します。

コンポーネント	設定
ODBC ドライバ (SybaseIQ)	• ODBC ドライバ・マネージャを介して設定します。または、**esp_iqloader** が検出できるパスに追加します。 • ターゲット・データベースの接続情報がある `odbc.ini` ファイルを作成します。
ユーザ・アカウント (esp_iqloader)	• **esp_iqloader** を使用するユーザ・アカウントが、ディレクトリの作成権限 (ディレクトリがない場合) を含め、作業ディレクトリとログ・ディレクトリに対するすべてのパーミッションを確実に所有するようにします。

コンポーネント	設定
作業ディレクトリ (esp_iqloader)	• **esp_iqloader** の作業ディレクトリに Sybase IQ が確実にアクセスできるようにします。Sybase IQ を実行しているサーバが **esp_iqloader** を実行しているサーバと異なる場合は、2 台のサーバ間でこのディレクトリを (たとえば、SAN や NAS を介して) 共有します。 これは、**LOAD TABLE** 文を正しく動作させるために必要です。
データのアーカイブ (esp_iqloader)	• 作業ディレクトリにボリュームの急増に対応できる十分なディスク領域を確実に用意しておきます。データをアーカイブする場合、特に SQL DML を適用してデータをアーカイブしていると、アーカイブ実行可能プログラムが Event Stream Processor に追いつけないことがあります。この場合は、大量の中間ファイルがディスクに書き込まれる可能性があります。
テーブル (SybaseIQ)	• 送信先データベースにターゲット・テーブルを事前定義し、それらのテーブルが該当するデータ型を使用して同じ順序で存在するようにします。 • アーカイブ実行可能プログラムが datetime カラムにデータを挿入するように指示されている場合は、ターゲット・テーブルにさらにこのカラムを追加します。 このカラムは、1 つ目または最後のカラムとしてのみ表示されるようにしてください。

esp_iqloader アーカイブの設定

XML 設定ファイルの要素定義について説明します。

PlatformArchive

```
<?xml version="1.0" encoding="UTF-8"?>
<PlatformArchive xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
    xsi:noNamespaceSchemaLocation="file:///install/etc/
esp_iqloader.xsd">
    archive definition
    ..
</PlatformArchive>
```

PlatformArchive 要素はヘッダ要素としての役目を果たします。上記の例では、`esp_iqloader.xsd` スキーマ・ファイルが `/install/etc` ディレクトリにあることを前提としています。 このディレクトリにない場合は、そのパスをスキーマ・ファイルの実際のパスに置き換えます。

注意： ヘッダの後では、アーカイブ設定で使用するオブジェクトを任意の順序で定義できます。次に、含めることのできる各オブジェクトの構文について説明します。

SybaseIQ

```
<SybaseIQ id="DestinationName" dsn="DsnName">
    <Option...../>
    <Option..../>
    ..
</SybaseIQ>
```

要素	定義
DestinationName	送信先の名前。DestinationName が設定ファイル内でユニークな名前であることを確認してください。
DsnName	odbc.ini ファイルにあるデータ・ソースの名前。この要素は Sybase IQ のターゲットを表し、ターゲット固有のバルク・ロード・オプションを指定する方法として使用できます。 この種類の要素は複数指定されることがありますが、所定の **esp_iqloader** インスタンスに使用できるターゲットは 1 つのみです。 この名前は、大文字と小文字の区別があります。 odbc.ini ファイルのエントリと正確に一致することを確認してください。

オプション

```
<Option name="OptionName" value="OptionValue"/>
```

要素	定義
OptionName	大文字と小文字の区別がある、**LOAD TABLE** 文オプションの名前。
OptionValue	**LOAD TABLE** 文オプションの値。指定されるオプションの数が多かったり、特定のオプションが複数回繰り返されたりすることがありますが、選択されるのは最新の値のみです。 **esp_iqloader** はオプションの検証を試みません。 オプションは **LOAD TABLE** 文で検証されます。エラーがあると、**LOAD TABLE** 文とアーカイブ・プロセスは終了します。

ストリーム

```
<Streams id="CollectionName">
    <Stream .../>
    <Stream .../>
    ..
</Streams>
```

要素	定義
CollectionName	アーカイブするストリームのコレクションの名前。これは、設定ファイル内でユニークである限り、どのような名前でもかまいません。 少なくとも 1 つの <Streams> 要素が定義され、その要素には少なくとも 1 つの <Stream> 要素があることを確認してください。

ストリーム

```
<Stream sourceName="SourceName"
    targetName="TargetName"
    logStreamName="LogStreamName"
    controlStreamName="ControlStreamName"
    [timestampLocation="{first | last}"]
/>
```

要素	定義
SourceName	Event Stream Processor のストリームの名前。この名前は、大文字と小文字の区別があります。たとえば、この名前が、XML プロジェクトで指定されているストリームの名前と一致することを確認してください。
TargetName	送信先データベースのテーブル名。この名前の大文字と小文字の区別は、送信先データベースのタイプによって異なります。
LogStreamName	(必須) アーカイブするストリームのトランザクション・ログがある、Event Stream Processor のストリームの名前。
ControlStreamName	(必須) ログ・ストリームへの入力として使用する制御ストリームの名前。このストリームを使用して、ログ・ストリーム内のアーカイブされたトランザクション・ログの消去を制御します。 Event Stream Processor スタジオの永続サブスクライブ・パターンを使用してログ・ストリームと制御ストリームを生成している場合、デフォルトの制御ストリームの名前は sourceName です。
TimestampLocation	(省略可能) アーカイブに含めるタイムスタンプのロケーションを示します。このタイムスタンプは、ターゲット・テーブルの 1 つ目または最後のカラムに設定できます。または、none に設定することもできます。1 つ目のカラムと最後のカラムのロケーションは追加されますが、none の timestamp ロケーションはアーカイブ・プロセスでは使用されません。

アーカイブ

```
<Archive target="DestinationName"
archiveStreams="CollectionName"
    [dbtempDir="WindowsDirPath"]
    [tempDir="WorkingDirPath"]
    [timestampName="TimestampColumnName"]
    [logDir="LogDirPath"]
/>
```

要素	定義
`DestinationName`	`<SybaseIQ>` 要素の ID。この要素は、送信先データベース・サーバを表します。
`CollectionName`	`<Streams>` 要素の ID。この要素で、アーカイブ対象の Event Stream Processor ストリームに関する情報を定義します。
`WindowsDirPath`	**esp_iqloader** が Windows の表記規則を使用して中間アーカイブ・データ・ファイルの管理に使用するロケーション。このディレクトリは Sybase IQ と Event Stream Processor の間で共有されるため、この要素は、Sybase IQ が Windows サーバで実行され、Event Stream Processor が UNIX または Linux サーバで実行されている場合に使用されます。
`WorkingDirPath`	**esp_iqloader** が中間アーカイブ・データ・ファイルの管理に使用するロケーション。 ディレクトリが存在しない場合は、このディレクトリが自動的に作成されます。ディレクトリ名が指定されていない場合、デフォルト値は現在のディレクトリ内の `archiveTemp` です。
`TimestampColumnName`	ターゲット・テーブル内の timestamp カラムの名前。**esp_iqloader** はこのカラムを使用して、アーカイブ実行可能プログラムがレコードを処理した時間を格納します。
`LogDirPath`	バルク・ロード実行可能プログラムの **LOAD TABLE** 文がログ・ファイルを入れるパス名。ディレクトリが存在しない場合は、このディレクトリが作成されます。 このディレクトリにログを生成するには、**LOAD TABLE** 文の `<SybaseIQ>` 要素に、**LOAD TABLE** 文の適切なオプションを指定します。

Sybase IQ アダプタのデータ型マッピング

Event Stream Processor のデータ型は Sybase IQ のデータ型にマップされます。

必要に応じて、Event Stream Processor のデータ型を他の互換性のある Sybase IQ のデータ型にマップするためのテーブルを Sybase IQ に作成できます。デフォルトのデータ型マッピングは、次のとおりです。

注意： Event Stream Processor の date と timestamp のデータ型は Sybase IQ の `datetime` のデータ型のみと互換性があります。

Event Stream Processor のデータ型	Sybase IQ のデータ型
`bigdatetime`	`timestamp`
`binary`	`binary`
`boolean`	`varchar(5)`
`date`	`datetime`
`float`	`float`
`integer`	`integer`
`interval`	`bigint`
`long`	`bigint`
`money`	`decimal(38,4)`
`money(n)`	`decimal(38,n)`
`string`	`varchar(n)`
`timestamp`	`datetime`

アーカイブ設定ファイルのサンプル

次は、`archive.xml` ファイル設定のサンプルです。

```
<PlatformArchive
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="file:///install/etc/
esp_iqloader.xsd" >
    <SybaseIQ id="Warehouse" dsn="ArchiveDsn">
        <Option name="FORMAT" value="BINARY"/>
        <Option name="ESCAPES" value="ON"/>
        <Option name="BLOCK SIZE" value="100000"/>
    </SybaseIQ>
```

```
<Streams id="MyArchive">
        <Stream sourceName="Titles" targetName="Titles" ¥
            timestampLocation="last"
controlStreamName="Titles_truncate"
            logStreamName="Titles_log"/>
        <Stream sourceName="Books" targetName="Books_Archive" ¥
            controlStreamName="Books_truncate"
logStreamName="Books_log"/>
        <Stream sourceName="Sales" targetName="Sales" ¥
            timestampLocation="last"
controlStreamName="Sales_truncate"
            logStreamName="Sales_log"/>
    </Streams>

    <Archive target="Warehouse" archiveStreams="MyArchive" ¥
        tempDir="/tmp/workdir" timestampName="ArchiveTime" ¥
        logDir="/tmp/logDir/"/>
</PlatformArchive>
```

ODBC DSN エントリのサンプル

次は、odbc.ini ファイル内の ODBC DSN エントリのサンプルです。

```
[ArchiveDSN]
Description                          = Sybase ODBC Data Source
UID                                     = dba
PWD                                     = sql
Driver                                 = Adaptive Server Enterprise
ENG                                     = datawarehouse_server
DBN                                     = books
LINKS                                = tcpip(host=dbServer;port=2638)
```

パラメータ	説明
UID	Sybase IQ のユーザ ID。
PWD	Sybase IQ のユーザ ID のパスワード。
Driver	ODBC ドライバ・マネージャを使用している場合はドライバの名前。ドライバの名前が odbcinst.ini ファイルのドライバ情報セクションの名前と一致していることを確認してください。Sybase IQ の ODBC ドライバを直接使用している場合、このパラメータは必要ありません。 odbcinst.ini ファイルの場所は、次のとおりです。 • /etc ディレクトリ (Linux の場合) • /windows ディレクトリ (Windows の場合)
ENG	Sybase IQ サーバの名前。

パラメータ	説明
DBN	データベースの名前。
LINKS	データベースへの接続に使用するネットワーク・プロトコル。

esp_convert

XML レコードと区切られたレコードを Event Stream Processor と互換性のある 32 ビットのバイナリ・レコードに変換します。ストリームを記述するメタデータは、Event Stream Processor の実行インスタンスへの (C & C インタフェースを介する) 接続から、または Event Stream Processor の互換性のある設定ファイルを使用して、取得します。

構文

```
esp_convert -f configFile -p [host:]port [OPTION...]
```

必須の引数

- **-f *configFile*** – (依存的に必須) Event Stream Processor のフォーマットに変換するデータが記述されている XML スタイルの設定ファイルを指定します。
- **-p *[<host>:]<port>/workspace-name/project-name*** – (必須) *host:<port>/<workspace name>/ <project name>* のすべての引数で、ESP サーバ (クラスタ・マネージャ) に接続する場合の URI を指定します。たとえば、ESP クラスタ・サーバを起動したときのポートが [19011] で、ホスト名が [localhost] として設定され、デフォルトのワークスペースで [prj1] プロジェクトを実行している場合は、`-p` を次のように指定します。 `-p localhost:19011/default/prj1`

オプション

- **-b** – (省略可能) サーバ (データの消費側) が実行されているマシン・アーキテクチャのバイト順序が、**esp_convert** が実行されているマシン・アーキテクチャの逆であることを示します。
- **-c *user[:password]*** – (省略可能) *user* ID と、オプションで *password* を使用して認証します。 *password* が指定されていないと、パスワードの入力を求めるプロンプトが表示されます。 Event Stream Processor でこれらのクレデンシャルを使用して認証が正常に行われた場合は、接続が維持されます。それ以外の場合は、Event Stream Processor はすぐに接続を切断します。
- **-d *separator*** – (省略可能) 標準入力から、デフォルトの XML フォーマットではなく、区切られたレコードを読み取って変換します。
- **-e** – (省略可能) openSSL ソケットを介する Event Stream Processor とのトラフィックを暗号化します。

注意： このオプションを使用するには、Event Stream Processor が暗号化モードで起動されていることを確認してください。

- **-f *configfile*** – (依存的に必須) 標準入力を介して読み取るデータの場所が記述されている CCX 設定ファイルを指定します。
- **-F *path*** – (省略可能) XML スキーマ・ファイルのフル・パス名を指定します (デフォルトは `$ESP_HOME/etc/Platform.xsd` です)。
- **-h** – (省略可能) 使用可能なオプションのリストを各オプションの簡単な説明とともに画面に出力します。
- **-k *privateRsaKeyFile*** – (省略可能) パスワード認証の代わりに、RSA プライベート・キー・ファイルのメカニズムを使用して認証を実行します。 *privateRsaKeyFile* には、プライベート RSA キー・ファイルのパス名を指定します。

注意： ESP サーバの起動時に `-k` オプションで RSA キーの保存先のディレクトリを指定したことを確認してください。

- **-m *datetimemask*** – (省略可能) 日付値のフォーマット文字列を (strptime フォーマットで) 指定します。 デフォルト値は "%Y-%m-%dT%H:%M:%S" です。
- **-v** – (省略可能) **esp_convert** ユーティリティのバージョンを出力します。

例

- **バイナリ・フォーマットへのレコードの変換** – XML ファイルまたはカンマ区切り値 (CSV) ファイルからレコードを変換します。

  ファイル `foo.xml` 内のすべての XML レコードをネイティブのバイナリ・フォーマットに変換し、それらを Event Stream Processor の実行インスタンスにポストするには、次のように入力します。

  ```
  cat foo.xml | esp_convert -p localhost:19011/default/prj1 |
  esp_upload -p localhost:19011/default/prj1
  ```

  ファイル `foo.csv` 内のすべてのカンマ区切りレコードをネイティブのバイナリ・フォーマットに変換し、それらを Event Stream Processor の実行インスタンスにポストするには、次のコマンドを使用します。

  ```
  cat foo.csv | esp_convert -d "," -p localhost:19011/default/prj1 |
  esp_upload -p localhost:19011/default/prj1
  ```

  ファイル `foo.xml` 内のすべての XML レコードをネイティブのバイナリ・フォーマットに変換し、**esp_upload** が実行されているマシンとはバイト順序が異なるターゲット・マシン HOST 上の Event Stream Processor の実行インスタンスにそれらをポストするには、次のコマンドを使用します。

```
cat foo.xml | esp_convert -b -p localhost:19011/default/prj1 |
esp_upload -b -p localhost:19011/default/prj1
```

ファイル `foo.xml` 内のすべての XML レコードをネイティブのバイナリ・フォーマットに変換し、**esp_upload** が実行されているマシンとはバイト順序が異なるターゲット・マシン HOST 上の Event Stream Processor の実行インスタンスにそれらをポストするには、次のコマンドを使用します。

```
cat foo.xml | esp_convert -b -p localhost:19011/default/prj1 |
esp_upload -b -p localhost:19011/default/prj1
```

入力フォーマット

次の例は、区切りフォーマットのレコードを示します。

```
StreamName<sep>Operation<sep>column_1..<sep>column_n
```

カラムはすべて区切りフォーマットで表し、ローの終わりに改行文字を付けます。 オペレーションは、それぞれ insert、update、delete、safe delete (レコードが存在する場合にのみ削除)、upsert に対する 1 文字 {i|u|d|s|p} です。

これは XML フォーマットのレコードです。

```
<StreamName [ESP_OPS="i|u|d|s|p"] [ESP_FLAGS="s"]
  column_name="value" ... column_name="value" />
```

ESP_OPS がない場合、オペレーションは upsert と見なされます。 存在するカラムの数についての要件はありません。 欠落しているカラムは NULL 値を持つと見なされます。 ESP_FLAGS がある場合は、レコードに対して SHINE フラグを設定することを示す "S" の値のみを割り当てることができます。

esp_kdbin

KDB データベース・テーブルから Event Stream Processor ストリームにデータを読み取ります。

esp_kdbin アダプタは、KDB データベースから Event Stream Processor のストリームにデータを読み取ります。 設定パラメータに基づいて、クエリ対象のデータまたはストリーミング・データのどちらかを読み取るようにアダプタを設定できます。

デフォルトでは、アダプタはフィールド名のマッチング (大文字と小文字は区別されません) を実行して、ソースの KDB テーブルとターゲット・ストリームの間の

マッピングを決定します。 また、マッピングを明示的に指定することもできます。

構文

```
esp_kdbin -H [kdbhost:]kdbport -p [host:]port -q source -s stream
[OPTION...]
```

必須の引数

- **-H *kdbhost:kdbport* –** KDB が受信待機するポート番号、またはホスト名とポート番号を指定します。 デフォルト値は localhost:5001 です。
- **-p *[<host>:]<port>/workspace-name/project-name* –** *host:<port>/<workspace name>/<project name>* のすべての引数で、ESP サーバ (クラスタ・マネージャ) に接続する場合の URI を指定します。たとえば、ESP クラスタ・サーバを起動したときのポートが [19011] で、ホスト名が [localhost] として設定され、デフォルトのワークスペースで [prj1] プロジェクトを実行している場合は、`-p` を次のように指定します。 `-p localhost:19011/default/prj1`
- **-q *source* –** ストリーミング・モードで実行している場合は、KDB テーブルを指定します。 非ストリーミング・モードで実行している場合は、有効なクエリ文字列を指定します。
- **-s *stream* –** ターゲット・ストリームを指定します。 読み取るデータのパブリッシュ先のストリームです。

オプション

- **-a –** 非同期モードの転送を使用します。この場合、アダプタは、データを受信したことを示す Event Stream Processor からの受信確認を待ちません。 プライマリとホット・スペアの両方がデータを確実に受信するようにホット・スペア設定を使用する場合に、このオプションは必要です。 デフォルト値は 'non async' です。
- **-b *blocksize* –** 転送する場合に 1 つのデータ・ブロックにまとめるレコードの数を指定します。 値が高いほど、スループットが増加する可能性がありますが、同時に遅延も増加します。 使用できるデータが十分にない場合は、ブロック内のレコード数が、指定された数より少ない可能性があります。 デフォルト値は 64 です。
- **-c *user[:password]* –** 認証のクレデンシャルを Event Stream Processor に渡します。 Event Stream Processor でこれらのクレデンシャルを使用して認証が正常に行われた場合は、接続が維持されます。それ以外の場合は、Event Stream Processor はすぐに接続を切断します。
- **-d –** デバッグ・メッセージを出力します。
- **-e –** アダプタと Event Stream Processor 間のすべての通信に暗号化方式の OpenSSL ソケットを使用します (Event Stream Processor を暗号化モードで起動す

ることが必要です)。 このオプションを指定しないと、暗号化は行われません。 デフォルトでは、暗号化されません。

- **-g *gatewayhost*** – 指定されたゲートウェイ・ホストを使用します。 Event Stream Processor から返されたホスト名は無視されます。
- **-h** – 詳細なヘルプを出力します。
- **-k *privateRsaKeyFile*** – パスワード認証の代わりに、RSA プライベート・キー・ファイルのメカニズムを使用して認証を実行します。 *privateRsaKeyFile* には、プライベート RSA キー・ファイルのパス名を指定します。

注意： ESP サーバの起動時に `-k` オプションで RSA キーの保存先のディレクトリを指定したことを確認してください。

- **-I *interval*** – 待機時間を秒数で指定します。非ストリーミング・モードで実行している場合、この時間を過ぎると、指定されたクエリが再度実行されます。値 `0` は、クエリの実行回数を 1 回のみとすることを示します。ポーリングは実行されません。 デフォルト値は 0 です。
- **-M *mapping*** – ターゲット・システムのカラム名と KDB データベース・テーブルのカラム名の間のマッピングを指定します。 マッピングは、コロンで区切られた一連の `SPColumn=KDBColumn` 文です。 このパラメータが指定されていないと、コネクタは、ターゲット・システムのカラム名がソース・テーブルのカラム名と一致する (大文字と小文字は区別されません) カラムのデータのみを吸収します。
- **-m** – このオプションが指定されていないと、アダプタは KDB データベースに接続して、ストリーミング・データを読み取ります。 指定されていると、アダプタは、指定されたデータベース・クエリを実行して、その結果を Event Stream Processor にフィードします。 デフォルトでは、ストリーミング・モードを使用します。
- **-T *attempts*** – 操作中に接続が切断された場合に KDB データベースへの再接続を試みる回数を指定します。 デフォルト値は 1 です。
- **-t** – トランザクション・ブロックを使用します。 これによってパフォーマンスは向上しますが、1 つのレコードが失敗すると、ブロック内のすべてのレコードが拒否されます。 デフォルトでは、エンベロープを使用します。
- **-u *user:password*** – 認証のクレデンシャルを KDB データベースに渡します。

例

サーバ myServer 上の KDB データベースの KdbTrades テーブルからデータを読み取り (ここで、KDB はポート 9200 で受信待機します) ローカル・サーバ上の Event Stream Processor の SpTrades ストリームにそのデータを書き込む (ここで、C & C インタフェースのポートは 1190 です) 基本的なストリーミング・モードのクエリを実行するには、次のコマンドを使用します。

```
esp_kdbin -p 1190 -H myServer:9200 -q KdbTrades -s SpTrades
```

同じクエリを実行して、KDB データベースのフィールドを Event Stream Processor ストリームのカラムに明示的にマッピングするには、次のコマンドを使用します。

```
esp_kdbin -p 1190 -H myServer:9200 -q KdbTrades -s SpTrades ¥
-M SpId=KId:SpSymbol=KSymbol:SpPrice=KPrice:SpCount=KCount
```

指定されたクエリをサーバ myServer 上の KDB データベースに 5 秒ごとに発行して (ここで、KDB はポート 9200 で受信待機します) サーバ outputServer 上の Event Stream Processor にそのデータを書き込む (ここで、C & C インタフェースのポートは 1221 です) プル・モードのオペレーションを実行するには、次のコマンドを使用します。

```
esp_kdbin -p outputServer:1221 -H myServer:9200 -q 'select Id,
Symbol, Price, Count from KdbTrades' ¥
-s SpTrades -m -I 5
```

esp_kdbout

Event Stream Processor から KDB データベース・テーブルにストリーミング・データをフィードします。

デフォルトでは、アダプタはフィールド名のマッチングを実行して (大文字と小文字は区別されません)、Event Stream Processor ストリームと KDB テーブルの間のマッピングを決定します。 また、マッピングを明示的に指定することもできます。

構文

```
esp_kdbout -H [kdbhost:]kdbport -p [host:]port -q source -s
table[OPTION...]
```

必須の引数

- **-p *[<host>:]<port>/workspace-name/project-name*** – *host:<port>/<workspace name>/<project name>* のすべての引数で、ESP サーバ (クラスタ・マネージャ) に接続する場合の URI を指定します。たとえば、ESP クラスタ・サーバを起動したときのポートが [19011] で、ホスト名が [localhost] として設定され、デフォルトのワークスペースで [prj1] プロジェクトを実行している場合は、-p を次のように指定します。 `-p localhost:19011/default/prj1`
- **-H *kdbhost:kdbport*** – KDB が受信待機するポート番号、またはホスト名とポート番号を指定します。 デフォルトのホスト名は localhost です。

- **-q** <***query***> – Event Stream Processor のストリームの名前、または KDB テーブルに書き込むデータを取得するための有効な SQL クエリのどちらかを指定します。
- **-s** <***stream***> – データの書き込み先の KDB テーブルの名前を指定します。

オプション

- **-a** – 非同期モードの転送を使用します。この場合、アダプタは、データを受信したことを示す Event Stream Processor からの受信確認を待ちません。
- **-B** – 削除可能なサブスクリプションを使用します。 アダプタがデータに追いつけなくなると、Event Stream Processor はサブスクリプションを削除します。
- **-b** <***blocksize***> – KDB テーブルへの 1 回のバッチ書き込みに取り込む最大レコード数を設定できます。 デフォルトは 5000 です。
- **-c** <***user:password***> – 認証のクレデンシャルを Event Stream Processor に渡します。 Event Stream Processor でこれらのクレデンシャルを使用して認証が正常に行われた場合は、接続が維持されます。それ以外の場合は、Event Stream Processor はすぐに接続を切断します。
- **-d** – デバッグ・メッセージをログに記録します。
- **-e** – アダプタと Event Stream Processor 間のすべての通信に暗号化された OpenSSL ソケットを使用します (Event Stream Processor を暗号化モードで起動することが必要です)。 このオプションを指定しないと、暗号化は行われません。
- **-h** – 使用可能なオプションのリストを各オプションの簡単な説明とともに画面に出力します。
- **-I** – 値を無視する KDB フィールド名のカンマ区切りリストを指定します。 これらのフィールドはメッセージには含まれますが、常に NULL が挿入されます。
- **-k** <***privateRsaKeyFile***> – パスワードの代わりに、RSA プライベート・キー・ファイルのメカニズムを使用して認証を実行します。 *privateRsaKeyFile* には、プライベート RSA キー・ファイルのパス名を指定します。 このオプションが有効な場合、ユーザ名は `-c` オプションで指定する必要がありますが、パスワードは必要ありません。 また、ESP サーバの起動時に `-k` オプションで RSA キーの保存先のディレクトリを指定したことを確認してください。
- **-L** <***interval***> – Event Stream Processor に接続しているときにパルス・サブスクリプションを使用します。 パルス *interval* は秒単位で指定します。
- **-l** – “損失を伴う”サブスクリプションを使用します。
- **-M** <***permutation***> – ターゲット・システムのカラム名と KDB データベース・テーブルのカラム名の間のマッピングを指定します。 マッピングは、コロンで区切られた一連の SPColumn=KDBColumn 文です。 このパラメータが指定されていないと、コネクタは、ターゲット・システムのカラム名がソース・テー

ブルのカラム名と一致するカラムのデータのみを吸収します (大文字と小文字は区別されません)。

- **`-m`** – “アップサート”オペレーションを使用してデータをテーブルに書き込みます。 このオプションが使用されていない場合、アダプタはストリーミング・モードで動作し、**u.upd** オペレーションを使用して KDB データベースにデータを書き込みます。
- **`-n`** – トランザクションのみで、基本データは受信されません。デフォルト値は false です。
- **`-O`** – メッセージから除外する KDB フィールド名のカンマ区切りリストを指定します。 メッセージに含まれるが常に NULL が挿入される無視対象のフィールドと異なり、これらのフィールドはメッセージに含まれません。
- **`-Q`** – クワイエット・モードで実行します。メッセージは表示されません。
- **`-R`** – シャインスルー (可能な場合) を使用してサブスクライブします。この場合、更新に新しいデータが入っていないフィールドでは以前の受信情報が保持されます。
- **`-T`** <***numtries***> – オペレーション中に接続が切断された場合に KDB データベースへの再接続を試みる回数を指定します。 デフォルトは 1 です。
- **`-t`** – KDB テーブルをターゲットにします。
- **`-u`** <***user:password***> – 認証のクレデンシャルを KDB データベースに渡します。
- **`-v`** – **esp_kdbout** ユーティリティのバージョンを出力します。

例

サーバ `myserver` 上の Event Stream Processor のプロジェクト `default/prj1` の `SpTrades` ストリームをサブスクライブして (ここで、クラスタ・マネージャのポートは 1221 です)、サーバ `outputserver` 上の KDB データベースの KdbTrades にデータをストリーミングするには (ここで、KDB はポート 9200 で受信待機します)、次のコマンドを使用します。

```
esp_kdbout -p myserver:1221/default/prj1 -H outputserver:9200 -q
SpTrades -s KdbTrades
```

KDB テーブルのフィールド XXX と YYY に (Event Stream Processor にそれらに対応するデータがないために) NULL を挿入するには、次のコマンドを使用します。

```
esp_kdbout -p myserver:1221/default/prj1 -H outputserver:9200 -q
SpTrades -s KdbTrades -I XXX,YYY
```

KDB のテーブルでは、データ更新メッセージで指定できないフィールド (たとえば、XXX、YYY など) が計算されていることがあります。 このような場合は、通

常、データベースで長さエラーが発生します。 更新メッセージからこれらのフィールドを完全に除外するには、次のコマンドを使用します。

```
esp_kdbout -p myserver:1221/default/prj1 -H outputserver:9200 -q
SpTrades -s KdbTrades -O XXX,YYY
```

esp_rapexport

Event Stream Processor によるデータ出力を変更して、RAP - The Trading Edition による入力として受け入れられるようにするアダプタとして機能します。 RAP が処理できるのは挿入のみなので、削除は削除され、更新は挿入に変換されます。

構文

```
esp_rapexport -f configFile -t templateDir -p publisherConfigDir
```

必須の引数

- **`-f` *config_file*** – RAP にデータを提供する Event Stream Processor 上のストリームに関する情報があるファイルのフル・パスを指定します。
- **`-p` *Dir*** – パブリッシャ・ファイルがあるディレクトリのフル・パスを指定します。このファイルで、RAP のサブスクライバが使用するマルチキャスト・アドレスを定義します。
- **`-t` *Dir*** – RDS テンプレートがあるディレクトリのフル・パスを指定します。このテンプレートで、Event Stream Processor のストリームのカラムを適切なデータベース・テーブルとカラムにマッピングします。

例

`config.xml` 設定ファイルと、`templates` と `publish` の各サブディレクトリを使用してアダプタを実行する方法を示します。これらのファイルとサブディレクトリはすべて、RAP_HOME 環境変数にフル・パスがあるホーム・ディレクトリの下にあります。

```
              esp_rapexport -f $RAP_HOME/config.xml -t
$RAP_HOME/templates -p $RAP_HOME/publish
```

esp_subscribe

C & C インタフェースとゲートウェイ・インタフェースを介して Event Stream Processor のインスタンスに接続し、トランザクション・ストリーミング・データ

をサブスクライブします。 受信レコードは XML (または必要に応じて区切りフォーマット) に変換されて、標準出力に書き込まれます。

構文

```
esp_subscribe  -p host:port/workspace/project [OPTION...]
```

必須の引数

- **-p *host:port/workspace/project*** – *host:port/workspace/project* のすべての引数で、ESP サーバ (クラスタ・マネージャ) に接続する場合の URI を指定します。たとえば、ESP クラスタ・サーバを起動したときのポートがローカル・マシンの [19011]で、デフォルトのワークスペースで [prj1] プロジェクトを実行している場合は、-p を次のように指定します。 `-p localhost:19011/default/prj1`

オプション

- **-A** – この *conn_handle* 接続ハンドルのサブスクリプションを追加します。**esp_subscribe** は要求を処理した後で終了します。
- **-b** – **esp_subscribe** が実行されているマシン・アーキテクチャとはバイト順序が逆のサーバに接続します。
- **-B** – **esp_subscribe** がデータを吸収できる速度よりデータを提供する速度が速くなった場合、接続を切断します。
- **-c *user[:password]*** – *user* ID と、オプションで *password* を使用して認証を実行します。 *password* が指定されていないと、ユーザにパスワードの入力を求めるプロンプトが表示されます。 Event Stream Processor でこれらのクレデンシャルを使用して認証が正常に行われた場合は、接続が維持されます。それ以外の場合は、Event Stream Processor はすぐに接続を切断します。
- **-d *separator*** – サブスクライブ・クライアント区切り出力モードに設定し、指定された *separator* を区切り文字として使用します。
- **-D *conn_handle*** – この *conn_handle* 接続ハンドルのサブスクリプションを削除します。 この **esp_subscribe** は要求を処理した後で終了します。
- **-e** – openSSL ソケットを介する Event Stream Processor とのトラフィックを暗号化します。

  注意： このオプションを使用する場合は、Event Stream Processor が暗号化モードで起動されていることを確認してください。
- **-f *configfile*** – 標準入力を介して読み取るデータの場所が記述されている CCX 設定ファイルを指定します。
- **-F *schemafile*** – XML スキーマ・ファイルを設定します。デフォルトのスキーマ・ファイルは `$ESP_HOME/etc/Platform.xsd` です。

- **-g** ***gateway_host*** – `get_gateway()` 呼び出しから返されたホストではなく、指定されたゲートウェイ・ホストを使用します。
- **-h** – 使用可能なオプションのリストを各オプションの簡単な説明とともに画面に出力します。
- **-H** – Event Stream Processor のハートビート・メッセージ (一定の間隔で送信される (sID,Qd)) をサブスクライブします。
- **-i** ***streamId[,...]*** – 各ストリームの整数ハンドルを使用して、サブスクライブする 1 つ以上のストリームを指定します。
- **-k** ***privateRsaKeyFile*** – パスワード認証の代わりに、RSA プライベート・キー・ファイルのメカニズムを使用して認証を実行します。 *privateRsaKeyFile* には、プライベート RSA キー・ファイルのパス名を指定します。

  注意： ESP サーバの起動時に `-k` オプションで RSA キーの保存先のディレクトリを指定したことを確認してください。
- **-l** – サブスクライブ・クライアントを「損失を伴う」モードに設定します。このモードでは、サブスクライブ・クラアイントが Event Stream Processor で生成されたストリーム・データに追いつけないと、データは廃棄されます (失われます)。
- **-L** ***N*** – パルス・サブスクリプションを指定します。この場合は、レコードが収集されて、*N* 秒ごとに配信されます。
- **-m** ***conn_name*** – 接続のシンボリックなタグ名を設定します。 これによって、**esp_subscribe** は _ESP_Clients メタデータ・ストリームで接続を簡単に検索できます。 このタグは *conn_tag* フィールドに格納されます。 タグ名で接続を強制終了するには、**esp_client** を使用して `kill every` コマンドを実行します。
- **-M** ***number*** – 確立する (すべて同じストリームの) 複数の接続の数を指定します。
- **-P** ***precision*** – `FLOAT` の出力での小数点以下の桁数を設定します。デフォルト値は小数点以下 6 桁です。
- **-Q** ***SQL statement*** – アウトバウンド・レコードに適用する SQL select 文を指定します。

  SQL 文にストリームのキー・カラムが含まれていない場合でも、アウトバウンド・レコードには挿入、更新、削除のマークが付きます。 これらの場合に挿入／更新／削除の意味を提供するために、特殊なカラム ESP_SEQNO を指定できます。 たとえば、文 `select *, ESP_SEQNO from Vwap order by Price` では、ESP_SEQNO カラムの値に自動的にデータが挿入されます。 `order by` を指定すると、ESP_SEQNO は出力セットにローの順序を反映します。

- **-R** – シャインスルー (可能な場合) を使用してサブスクライブします。この場合、以前受信したフィールドの新しいデータが更新に入っていないと、それらのフィールドの情報は保持されます。
- **-s** ***stream_1 [,...,stream_N]*** – サブスクライブする 1 つ以上のストリームを、各ストリームの論理名を使用して指定すると、テーブルのスナップショットが作成されます (SYNC レコードを受信した後で終了します)。
- **-S** – テーブルのスナップショットを取得します。 ストリームの初期状態を受信した後、すぐに終了します。
- **-t** – サブスクライブ・クライアントをトランザクション専用モードに設定します。この場合は、Event Stream Processor への接続時に、ストリームの初期状態ではなく、ストリームのトランザクション更新を受信します。
- **-T** – トランザクション・ブロック開始マーカ `<block>` とトランザクション・ブロック終了マーカ `</block>` を出力します。
- **-v** – 特殊なイベント (同期の開始、同期の終了、ストリームの終了など) をレポートします。
- **-W** ***baseDrainTimeout*** – クライアントがサブスクライブ・ストリームからすべての基本データを読み取るときの時間制限をミリ秒単位で設定します。この時間を過ぎると、基本データは削除されます。 このパラメータが指定されていない場合、デフォルト値は 8000 ミリ秒 (8 秒) です。
- **-X** – Event Stream Processor が接続を切断していない場合でも、すべてのサブスクリプションが終了したらプロセスを終了します。
- **-z** ***queueSize*** – 出力ゲートウェイ接続のサブスクライバ・キューのサイズを *queueSize* に設定します。 このキューの先頭に達すると接続されたソケットが終了し、クライアントへ配信するデータがバッファされます。 *queueSize* の最小値は 1000 レコードです。 デフォルト値は 8192 レコードです。
- **-V – esp_subscribe** ユーティリティのバージョンを出力します。

例

localhost:11180 のクラスタ・マネージャで実行されている default/prj1 プロジェクトの PreprocessorTransactions と DebitMovements の 2 つのストリームをサブスクライブして、すべてのストリーム・データを XML フォーマットで標準出力に出力するには、次のように入力します。

```
esp_subscribe -c user:pass -s
PreprocessorTransactions,DebitMovements -p localhost:11180/default/
prj1
```

PreprocessorTransactions と DebitMovements の 2 つのストリームをサブスクライブして、すべてのストリーム・データをパイプ区切りフォーマットで標準出力に出力するには、次のように入力します。

```
esp_subscribe -c user:pass -d "|" -s
PreprocessorTransactions,DebitMovements -p localhost:11180/default/
```

```
prj1
```

HOST マシン (サブスクライブが実行されているマシンとはバイト順序が逆) で実行されているサーバで生成されたデータがある 1 つのストリーム PreprocessorTransactions をサブスクライブして、すべてのストリーム・データをパイプ区切りフォーマットで標準出力に出力するには、次のように入力します。

```
esp_subscribe -c user:pass -d "|" -s PreprocessorTransactions -p
localhost:11180/default/prj1
```

1 つのストリーム baseInput をサブスクライブして、SQL 文を適用するには、次のように入力します。

```
esp_subscribe -c user:pass -Q   "select intData_1,
10*intData_1+dblData_1 from baseInput where intData_1 > 20" -p
localhost:11180/default/prj1
```

エラー・ストリーム ErrorStream をサブスクライブするには、次のように入力します。

```
esp_subscribe -p localhost:11180/default/prj1 -Q "select e.*,
recordDataToString(e.sourceStreamName, e.errorRecord ) errorRecord
from ErrorStream e"
```

esp_upload

標準入力からのバイナリ・レコードを記録し、ゲートウェイ・インタフェースを介して Event Stream Processor の実行インスタンスにそれらを転送します。

データのフォーマットは、0 個以上の `<Stream Handle><Raw Binary Record>` で構成されます。`<Stream Handle>` は、レコードの送信先ストリームを示す `uint32_t` です。 通常、このツールは **esp_convert** ツールと一緒にパイプラインの終わりで使用します。

構文

```
esp_upload -p [<host>:]<port>/workspace-name/project-name
[OPTION...]
```

オプション

- **-b** – バイトスワップ・モードを設定します。 **esp_upload** に (**esp_convert** `-b` を介して) フィードされるロー・レコードと **esp_upload** がデータを送信するサーバは、**esp_upload** クライアントが実行されているアーキテクチャとはバイト順序が異なります (データのバイト順序は常にサーバのバイト順序と一致する必要があります)。

- **-c *user[:password]*** – *user* ID と、オプションで *password* を使用して認証を実行します。 *password* が指定されていないと、ユーザにパスワードの入力を求めるプロンプトが表示されます。 Event Stream Processor でこれらのクレデンシャルを使用して認証が正常に行われた場合は、接続が維持されます。それ以外の場合は、Event Stream Processor はすぐに接続を切断します。
- **-d *N*** – レコード間またはトランザクション・ブロック間に *N* ミリ秒の遅延を挿入します。
- **-e** – openSSL ソケットを介する Event Stream Processor とのトラフィックを暗号化します。

  注意： このオプションを使用するには、Event Stream Processor が暗号化モードで起動されていることを確認してください。
- **-f *timeout:finalizer*** – 実行するファイナライザを設定します。 **esp_upload** から *timeout* ミリ秒以内にメッセージを受信しないと、ESP サーバは SQL ファイナライザ文 (セミコロンで区切られた insert 文、update 文、または delete 文の組み合わせ) を実行します。 SQL 文は、**esp_upload** が停止した場合も実行されます。
- **-h** – 使用可能なオプションのリストを各オプションの簡単な説明とともに画面に出力します。
- **-k *privateRsaKeyFile*** – パスワード認証の代わりに、RSA プライベート・キー・ファイルのメカニズムを使用して認証を実行します。 *privateRsaKeyFile* には、プライベート RSA キー・ファイルのパス名を指定します。 ESP サーバの起動時に `-k` オプションで RSA キーの保存先のディレクトリを指定したことを確認してください。

  注意： このオプションが有効な場合、ユーザ名は `-c` オプションで指定する必要がありますが、パスワードは必要ありません。
- **-m *conn_name*** – 接続のシンボリックなタグ名を設定します。 これによって、**esp_upload** は _ESP_Clients メタデータ・ストリームで接続を簡単に検索できます。 タグ名で接続を強制終了するには、**esp_client** コマンドを使用します。
- **-p *[<host>:]<port>/workspace-name/project-name*** – (必須) *host:<port>/<workspace name>/ <project name>* のすべての引数で、ESP サーバ (クラスタ・マネージャ) に接続する場合の URI を指定します。たとえば、ESP クラスタ・サーバを起動したときのポートが [19011] で、ホスト名が [localhost] として設定され、デフォルトのワークスペースで [prj1] プロジェクトを実行している場合は、`-p` を次のように指定します。 `-p localhost:19011/default/prj1`
- **-r *N*** – レコードまたはトランザクション・ブロックを秒あたり *N* 個の速度でアップロードします。

- **-s** *N* – *N* 個のレコードまたはトランザクション・ブロックごとにソース・ストリームを同期します。 これで、レコードはソース・ストリームに確実に吸収されます。
- **-t** *size* – トランザクション・モードで実行します。 **esp_upload** が読み取る各レコードは、ストリーム単位でバッファされます。 バッファ内のレコード数が指定された数に達すると、1 つのトランザクションとしてラップされて Event Stream Processor に送信されます。 読み取られたレコードがすべて 1 つのストリームからのものである場合は、これにより、そのストリームは *size* のレコード・チャンクに効果的にバッファされ、それらのレコード・チャンクがトランザクションとしてコミットされます。 バッファされたレコードは、EOF が読み取られるとストリームごとに 1 つのトランザクションとして送信されます。
- **-w** *size* – エンベロープ・モードで実行します。 **esp_upload** が読み取る各レコードは、ストリーム単位でバッファされます。 バッファ内のレコード数が指定された数に達すると、1 つのエンベロープとしてラップされて Event Stream Processor に送信されます。 読み取られたレコードがすべて 1 つのストリームからのものである場合は、これにより、そのストリームは *size* のレコード・チャンクに効果的にバッファされます。 バッファされたレコードは、EOF が読み取られるとストリームごとに 1 つのエンベロープとして送信されます。
- **-x** – 標準入力で EOF を受信すると、データのアップロードが終了した各ストリームに <END OF STREAM> マーカを送信します。 Event Stream Processor のすべてのソース・ストリームが <END OF STREAM> マーカを受信すると、Event Stream Processor はシャットダウンして終了します。
- **-X** – アップロードが完了した時点でプラットフォームを強制終了します。
- **-Y** *<beat>* – *<beat>* マイクロ秒内にデータを受信しなかった場合に、プラットフォームを強制終了します。
- **-v** – **esp_upload** ユーティリティのバージョンを出力します。

例
CSV と XML の各入力ファイルのフォーマットについては、「**esp_convert**」を参照してください。

ファイル `foo.xml` 内のすべての XML レコードをネイティブのバイナリ・フォーマットに変換し、それらを Event Stream Processor の実行インスタンスにポストするには、次のように入力します。

```
cat foo.xml | esp_convert -p localhost:11180/default/prj1 |
esp_upload -c user:pass -p localhost:11180/default/prj1
```

ファイル `foo.csv` 内のすべてのカンマ区切りレコードをネイティブのバイナリ・フォーマットに変換し、それらを Event Stream Processor の実行インスタンスにポストするには、次のように入力します。

```
cat foo.csv | esp_convert -d "," -p localhost:11180/default/prj1 |
esp_upload -c user:pass -p localhost:11180/default/prj1
```

ファイル `foo.xml` 内のすべての XML レコードをネイティブのバイナリ・フォーマットに変換し、**esp_upload** が実行されているマシンとはバイト順序が異なるターゲット・マシン HOST 上の Event Stream Processor の実行インスタンスにそれらをポストするには、次のように入力します。

```
cat foo.xml | esp_convert -b -p localhost:11180/default/prj1 |
esp_upload -c user:pass -b -p localhost:11180/default/prj1
```

第 5 章 オーサリングの実行可能プログラム

esp_compiler

指定された一連の SQL 文を、Event Stream Processor で使用される対応する XML 表現に変換します。 また、SQL 文が正しいことを確認し、データ型の整合性を検査して、限られた最適化を実行します。

構文

```
esp_compiler -i<CCL File> -o<CCX File>
```

必須の引数

- **`-i` *CCL File* –** 指定されたファイルを入力として使用します。このファイルが CCX に変換されます。

オプション

- **`-o` *CCX File* –** 指定した名前のファイルに CCX 出力を書き込みます。このファイルが既に存在する場合は、上書きされます。 指定しないと、デフォルトで、CCX は標準出力に書き込まれます。
- **`-v` – esp_compiler** ユーティリティのバージョンを出力します。

esp_studio

ESP スタジオを起動します。スタジオを使用してプロジェクト (つまり、継続クエリ) をグラフィックで作成し、ESP サーバによるこれらのサービスの実行を開始してモニタします。

注意： `esp_studio` のバージョン情報を調べるには、ESP スタジオを起動して [About] ダイアログにアクセスします。他のユーティリティで使用されている `-v` オプションは、`esp_studio` では使用できません。

構文

```
esp_studio
```

第 6 章　拡張デバッグ

概要

Sybase Event Stream Processor には、ご使用のプロジェクトで問題を見つけて修正するためのデバッグ機能が含まれています。

これらの機能は、Event Stream Processor スタジオと **esp_client** コマンド・ライン・ユーティリティの 2 つのインタフェースから使用できます。デバッグ・ツールはプロジェクトの開発時に使用するものです。Event Stream Processor が運用モードの場合は使用しないでください。デバッグ・ツールは Event Stream Processor にかなりの負荷をかけるため、通常は無効になっています。

デバッグ・ツールを有効にするには、Event Stream Processor をトレース・モードで実行します。

トレース・モード

トレース・モードでは、Event Stream Processor は有効なデバッグ操作とブレークポイントの追加チェックを実行して、実行履歴に関する追加情報を収集します。

esp_client ユーティリティを使用して、トレース・モードの有効化、無効化、トレース・モード状況の確認を実行します。

たとえば、**esp_client** のインスタンスでは、トレース・モードの状況を確認し、トレース・モードをオンにして、状況を再確認してから、オフにします。

たとえば、**esp_client** のインスタンスでは、トレース・モードの状況を確認し、トレース・モードをオンにして、状況を再確認してから、オフにします。

```
esp_client> trace_mode
trace mode is off
esp_client> trace_mode on
esp_client> trace_mode
trace mode is on
esp_client>trace_mode off
esp_client>trace_mode
trace mode is off
```

トレース・モードは、**esp_client** のインスタンスには関連付けられません。トレース・モードをオンにしたまま、**esp_client** を終了できます。**esp_client** の別のイン

スタンスでトレース・モードをオフにするまで、Event Stream Processor はトレース・モードを維持します。

Event Stream Processor のステップ

システムが一時停止しているときに Event Stream Processor をステップしてプラットフォームの状態を進めることができます。

ストリームはシングル・ステップまたはマルチ・ステップで進めることができます。変更を大幅に加えるには、ストリームにマルチ・ステップが必要になることがあります。そのため、ユーザはシステムでトランザクション全体を実行するか、ストリームがクワイス状態になるまで実行することができます。また、ユーザはシステムの再実行を選択することもできます。この場合、システムは通常、別のブレークポイントがトリガされるまで実行します。

自動シングル・ステップ
自動ステップは、ブレークポイントや不正ロー例外では機能しません。ブレークポイントや不正ロー例外が検出されると、スタジオには報告されますが、ステップは停止されません。

1 つ目の自動ステップ・コマンド **step trans** は、トランザクションの終わりまでステップします。このコマンドは、少なくとも 1 つの共通ステップを実行してから、ストリームが COMPUTE ロケーションにいる限りステップを続行します。 ストリームが PUT ロケーションまたは BAD_ROW ロケーションに移動すると、ストリームは停止するので、ユーザはトランザクションの影響を検査してから、トランザクションをコミットまたは破棄できます。 **step trans** を繰り返し呼び出して、以前のトランザクションをステップします。

実行による INPUT ロケーションまたは OUTPUT ロケーションのブロックが 0.3 秒を超えると、**step trans** は停止します。

その他の自動ステップ・コマンドはクワイスの概念に関連します (使用可能なすべての入力の処理が完了するまで、ストリームを実行します)。 それらのコマンドは次のとおりです。

コマンド	機能
step quiesce stream *{streamName}*	クワイス状態になるまで (その入力キューすべてが空になるまで)、ストリームとその直接と間接の下位ストリームすべてを自動的にステップします。
step quiesce downstream *{streamName}*	ストリームの下位ストリームのみをステップします。 ストリーム自体のステップは実行されません。このコマンドを使用して、下位ストリームの入力キューを空にします。 引数のストリームが出力を生成する場合に、下位ストリーム内のデータの通過を簡単にトレースできます。

コマンド	機能
step quiesce from base	入力キューが空になるまで、すべての派生 (非ソース) ストリームを自動的にステップします。 このコマンドを使用して派生ストリームのキューを空にしてから、ソース・ストリームからの矛盾するレコードを処理します。

注意： Event Stream Processor が一時停止しているときに **esp_client** またはスタジオを終了すると、Event Stream Processor は一時停止したままになります。 **esp_client** の別のインスタンスと接続しても、一時停止したままです。Event Stream Processor がトレース・モードになっている場合は、そのモードのままになります。 ブレークポイントや例外を検出すると、一時停止し、一時停止が解除されるまですべての処理を停止します。

Event Stream Processor が一時停止していても、**esp_client** から Event Stream Processor を停止できます。Event Stream Processor の一時停止が解除され、トレース・モードが無効になって、プラットフォームが停止し、通常どおり終了します。

トレース・モードを無効にした場合もプラットフォームの一時停止は解除されます。

Event Stream Processor の一時停止

トレース・モードでは、一時停止コマンドを発行してストリーム処理ループを一時停止できます。ループを一時停止すれば、ストリームを静的な状態で検査できます。

Event Stream Processor がトレース・モードである間、ストリーム処理メカニズムは、処理グループのロケーションに入るたびに一時停止要求を確認します。一時停止コマンドを発行すると、ストリームは一時停止し、ストリームの続行が許可されるまでループを再開しません。Event Stream Processor が一時停止すると、計算は実行されません。ストリームの出力バッファ内のトランザクションは、サブスクライバで引き続き消費できます。一時停止されたストリームへのパブリッシュは、ストリームの入力バッファが満杯になるまで続きます。

一時停止が要求されたときにストリームが実際の処理に取り組んでいた場合は、ストリームが次のロケーションに入るまで処理は続行されます。

ストリームは、I/O ロケーションにある間は一時停止要求の影響を受けません。たとえば、ストリームは、INPUT ロケーションでは入力キューにトランザクションがなくなった場合に、OUTPUT ロケーションでは出力キューが満杯になった場合に、自動的に一時停止します。

ストリームは、一時停止している場合でも、I/O ロケーションと処理ロケーションの間で移動することがあります。ストリームのサブスクライバが低速であると、ストリームの出力は満杯になり、バッファ領域が使用できるようになるまで、ス

トリームは OUTPUT ロケーションに残ります。このロケーションにあるストリームが一時停止要求を受信した場合、その要求は無視されます。ストリームのサブスクライバがこの要求の後で出力バッファからトランザクションを取得すると、ストリームは現在の出力トランザクションをバッファに預けて、INPUT ロケーションに進みます。この時点では、INPUT ロケーションに入ってから、ストリームは一時停止要求を認識します。一時停止要求後の新しいデータは処理されませんが、ストリームはロケーションを変更します。

他のストリームと同様にメタデータ・ストリームも一時停止します。Event Stream Processor が一時停止している間は、_ESP_RunUpdate を除き、これらのストリームから更新を受け取ることはできません。

esp_client コマンド・ライン・ユーティリティを使用して、Event Stream Processor の一時停止状況の確認、一時停止、一時停止解除を実行します。

```
esp_client> check_pause
Platform is not paused
esp_client> pause
esp_client> check_pause
PAUSED
esp_client> run
esp_client> check_pause
Platform is not paused
```

ストリーム処理のループ

Event Stream Processor のストリームの内部ロジックは、Event Stream Processor のデータの処理方法に対応する状態を持つループとして表すことができます。

通常の処理シーケンスは、次のように進みます。

1. INPUT
   ストリームは、入力キューが空以外の状態になるまで待ち、入力キューの先頭からトランザクションを選択します。このトランザクションは inTrans (現在の入力トランザクション) として参照できます。トランザクションはロー単位で処理されます。
   現在の出力トランザクションは、処理の結果を収集できるように、空に設定されます。
2. COMPUTE
   現在の入力トランザクションから次のレコードが選択されます。これは inRow (現在の入力ロー) として参照できます。場合によっては、現在の入力レコードが実際は 2 つのレコードで、1 つの UPDATE_BLOCK にまとめられていることもあります。
   これがループの最初の反復ではなければ、直前の入力レコードの処理から生成されたレコードを outRow として従来どおり参照できます。

ストリームに入力ブレークポイントが定義されている場合は、それらのブレークポイントが現在の入力レコードに対して評価されて、Event Stream Processor の一時停止をトリガする可能性があります。

Event Stream Processor が一時停止しているかどうかの確認が実行されます。一時停止している場合、ストリームはここで一時停止し、続行するためのパーミッションを待ちます。

最後に、現在の入力レコードに対して実際の計算が実行されます。0 個以上の出力レコードが生成されます。これらのレコードは outRow として参照でき、outTrans の終わりにも追加されます。これらのレコードは特定の内部ルールに従っているので、外部にパブリッシュされる場合とまったく同じとは限りません。たとえば、通常、この時点の更新レコードのオペレーション・タイプは UPSERT で、削除レコードは SAFEDELETE です。

ストリームに定義されている出力ブレークポイントが現在の入力レコードに対して評価されて、Event Stream Processor の一時停止をトリガする可能性があります。

入力トランザクションにさらにレコードが残っていれば、計算ループは続行します。それ以外の場合、ストリームは、ゼロによる除算などの例外が発生しない限り、計算済みのデータをストアに入れる処理に進みます。このような例外が発生した場合は、BAD_ROW 処理に進みます。

3. PUT

Event Stream Processor が一時停止しているかどうかの確認が実行されます。一時停止している場合、ストリームはここで一時停止し、続行するためのパーミッションを待ちます。

新しい結果がストリームのストアに挿入されます。これは単純なプロセスではありません。結果のトランザクションはクリーニングされ、ストアに既に含まれていた情報に従って変換されます。このため、これ以降、現在の出力トランザクションは参照できません。現在の出力ローもありません。これらの変換には、次のようなものがあります。

- SAFEDELETE：破棄される (ストアにこのようなレコードがなかった場合) か、DELETE に変換されます (ストアにあったすべてのデータが挿入されてから削除されます)。
- UPSERT：INSERT または UPDATE_BLOCK のどちらかに変換されます。その他の UPDATE はすべて UPDATE_BLOCK に変換されます。または、レコード内のデータが以前の状態から変更されていない場合は破棄されることもあります。UPDATE_BLOCK はレコードのペアです。1 つ目のレコードは、オペレーション・タイプが UPDATE_BLOCK で、新しい値があります。2 つ目のレコードは、オペレーション・タイプが DELETE で、古い値があります。UPDATE_BLOCK が Event Stream Processor の外側にパブリッシュされると、2 つ目のレコードは破棄され、1 つ目のレコードが UPDATE に変換さ

れます。Event Stream Processor の内側では、更新ブロック全体を参照できます。

PUT は、たとえば、ストアに既にあるキーを持つレコードを挿入しようとした場合に、例外をトリガすることがあります。この場合は、トランザクション全体が中止されて、ストリームは BAD_ROW の位置に移動します。

ストリームの履歴には、現在の入力トランザクションと現在の出力トランザクション (既に変換済みのもの) が挿入されます。入力トランザクションは inHist の終わりに追加され、出力トランザクションは outHist の終わりに追加されます。これで処理が完了したので、inTrans と inRow は参照できなくなります。outTrans と outRow はこの時点では既に参照できません。

4. OUTPUT

   結果トランザクションはキューに入れられて、クライアントにパブリッシュされます。一部のクライアントの速度が遅すぎて、出力バッファが満杯になると、ストリームはバッファ領域が使用可能になるまで待ちます。

   結果トランザクションは、このストリームを入力とするストリームに配信されます。繰り返しますが、入力キューのいずれかが満杯になると、このストリームはそれらが使用可能になるまで待ちます。

   ストリームは、次のトランザクションの INPUT に進みます。

このメイン・ループの他に、サイド・ブランチもあります。有効期限のあるストリームの場合は、1 秒おきに次のサイド・ブランチが発生します。

- EXPIRY

ブレークポイントと例外

ブレークポイント機能は、データ・モデルに問題があると Event Stream Processor を一時停止させます。ブレークポイントは、Event Stream Processor を停止させて、ユーザが問題のトラブルシューティングを実行できるようにします。

システムを一時停止させる方法は 2 つあります。明示的な一時停止コマンドを送信する方法と、ストリームまたはウィンドウにブレークポイントを設定する方法です (これにはローカル・ストリームも含まれます)。ブレークポイントは、ストリームの入力または出力に設定できます。追加できるブレークポイントの数に制限はありません。タプルがブレークポイントを検出すると、実行が一時停止されます。

ストリームまたはウィンドウのブレークポイントには特定の条件を設定できます。たとえば、特定のレコードのみが一時停止をトリガするようにブレークポイントにフィルタを設定できます。また、指定されたレコード数を超えたらブレークポイントをトリガするように、時間条件を設定することもできます。

ストリームのブレークポイントには次の 2 つの種類があります。

- 入力時 - 入力トランザクションから処理対象のローを取得したときにブレークポイントを確認してから、計算を実行します。 Event Stream Processor は COMPUTE ロケーションで一時停止します。
  - すべての入力時 - すべてのストリームからの入力を確認します。
  - 特定のストリーム時 - 特定のストリームから入力トランザクションを受け取った場合にのみ確認します。
- 出力時 - ローを処理してから、ブレークポイントを確認します。Event Stream Processor は次のロケーション (次の入力ローのための COMPUTE、PUT、または BAD_ROW) で一時停止します。

無条件ブレークポイントと例外
単純なブレークポイントは無条件です。ストリームが適切なロケーションにあると、ブレークポイントがトリガされて、Event Stream Processor は一時停止します。複数のブレークポイントを同時にトリガできます。まったく同じロケーションに複数のブレークポイントを定義できます。プラットフォームは、ブレークポイントの作成時に割り当てられたユニークな ID でブレークポイントを区別できます。同じブレークポイントを 2 つ作成すると、Event Stream Processor はそれぞれに別々の ID を割り当てて、両方のブレークポイントを同時にトリガできるようにします。

ブレークポイントが作成されたら、ブレークポイントを有効、無効、またはその情報の一部を変更できます。ブレークポイントを別のロケーションに移動したり、その条件式を変更したりすることはできません。大きな変更を加えるには、ブレークポイントを削除してから、新しいブレークポイントを作成します。

プラットフォームを任意のレコードで一時停止させることもできます。たとえば、ストリームから渡された 1000 番目のレコードでバグが表面化した場合は、999 個のレコードを渡してから、一時停止し、そのポイントからシングル・ステップを実行できます。このためには、n 番目のローごとにトリガするようにブレークポイントを設定します。プラットフォームを再起動すると、ブレークポイントは次に 2000 番目のローでトリガされます。 `bp list` で、フィールド `enabledEvery` はブレークポイント番号 (N) を示し、`leftToTrigger` はブレークポイントがトリガされるまで表示させておくレコードの数を示します。ブレークポイントがトリガされるたびに、 `leftToTrigger` は `enabledEvery` の元の値にリセットされます。デフォルトでは、ブレークポイントの作成時に `enabledEvery` は 1 に設定されて、ローごとにトリガします。 これは、**esp_client** コマンド **bp every** を使用して変更できます。

ユーザは、0 番目のレコードごとにトリガするようにブレークポイントを設定するか、**bp on** コマンド (レコードごとにブレークポイントをトリガするコマンド) を使用して、ブレークポイントを一時的に無効にできます。

条件ブレークポイントと例外
ストリームから渡された特定の内容を持つレコードを確認するには、条件ブレークポイントを使用します。

Event Stream Processor で条件ブレークポイントを適用するには、ブレークポイントのフィルタ式を指定します。 最初にフィルタ式が評価され、その結果が false (0 または NULL) 値であると、ブレークポイントはスキップされます。 式の結果が true 値の場合 (またはブレークポイントの `leftToTrigger` カウントが減った場合) にのみブレークポイントはトリガされます。

フィルタ式は SPLASH では標準です。2 つの事前定義レコード変数 `currow` と `oldrow` からのデータを使用します。変数 `currow` には現在のレコードが入ります。`oldrow` は入力時のブレークポイントに対してのみ定義されます。 INSERT と単純な UPDATE では、`oldrow` の値は NULL です。UPDATE_BLOCK では、`oldrow` にはブロックの 2 つ目のレコード、つまり置き換える古いレコードが入ります。DELETE と SAFEDELETE では、`oldrow` には `currow` と同じデータが入ります。 特定のフィールドは、通常の `currow.field` 構文を使用して評価できます。 ローのオペレーション・コードを取得するには、**getOpcode(currow)** を使用します。

これらの事前定義変数を提供するロー定義は、ブレークポイントのタイプによって異なります。

出力時のブレークポイントの場合、ブレークポイントはロー定義で定義されます。式は、前の COMPUTE 時に生成された出力ローで評価されます。 複数のローが生成される可能性があるので、式はローごとに評価されます。 ローが生成されなかった場合、式は 1 回評価され、`currow` は NULL に設定されます。 この場合、COMPUTE の出力時に UPDATE_BLOCK は生成されないため、`oldrow` は使用できません。

特定のストリームからの入力時のブレークポイントの場合、ブレークポイントはその入力ストリームのロー定義で定義されます。式は、計算する予定のレコードまたは更新ブロックで評価されます。この種類のブレークポイントではフィルタ式は使用できません。

ソース・ストリームは、他のストリームではなく、プラットフォームの外側からデータを受け取ります。ソース・ストリームの入力に条件ブレークポイントを追加するには、**bp add** *{filterInput} {filterInput}* で、入力ストリームにソース・ストリームの独自の名前を使用します。

デバッガ・イベントの通知

通知を受信できるのは、Event Stream Processor が実行と一時停止の間で切り替えた場合、シングルステップを実行する場合、プラットフォームがブレークポイントと例外を検出した場合です。

これらの更新を受信するには、_ESP_RunUpdates ストリームをサブスクライブできます。このストリームにコンテンツは保持されません。通知はストリームのストアをバイパスします。オペレーション・タイプは常に UPDATE です。 詳細については、『CCL Reference Guide』を参照してください。

サンプル・デバッグ：Event Stream Processor の一時停止

esp_client コマンド・ライン・ユーティリティを使用して、Event Stream Processor の一時停止状態の確認、一時停止、一時停止解除を実行します。

1. Event Stream Processor を一時停止するには、次のコマンドを使用します。

   ```
   esp_client> pause
   ```

2. Event Stream Processor を実行し、実行を継続するには、次のコマンドを使用します。

   ```
   esp_client> run
   ```

3. Event Stream Processor が一時停止しているかどうかを確認するには、次のコマンドを使用します。

   ```
   esp_client> check_pause
   ```

   Event Stream Processor は、一時停止しているかどうかをユーザに通知します。一時停止していない場合は、`Platform is not paused`が表示されます。一時停止している場合は、`PAUSED` が表示されます。

サンプル・デバッグ：Event Stream Processor のステップ

esp_client コマンド・ライン・ユーティリティを使用して、シングル・ステップでストリームを進めます。
シングル・ステップでストリームを進めるには、次のコマンドを使用します。

```
esp_client> step [`stream`]
```

Event Stream Processor によって、処理を必要とするストリームを選択するには、次のコマンドを使用します。

```
esp_client> step
```

自動ステップ

esp_client コマンド・ライン・ユーティリティには、自動ステップのための多くのオプションが用意されています。これらのコマンドを使用すれば、連続してシングル・ステップを実行する必要がありません。

1. ストリームをトランザクションの終わりまで進めるには、次のコマンドを使用します。

   ```
   esp_client> step trans [`stream`]
   ```

2. ストリームとそのストリームの直接と間接のすべての下位ストリームを、その入力キューのすべてが空になるまでステップするには、次のコマンドを使用します。

   ```
   esp_client> step quiesce stream {streamName}
   ```

3. ストリームの下位ストリームのみをその入力キューが空になるまでステップするには、次のコマンドを使用します。

   ```
   esp_client> step quiesce downstream {streamName}
   ```

4. すべての派生ストリームをその入力キューが空になるまでステップするには、次のコマンドを使用します。

   ```
   esp_client> step quiesce from base
   ```

サンプル・デバッグ：ブレークポイントの追加

esp_client コマンド・ライン・ユーティリティを使用して、ストリームまたはウィンドウのブレークポイントを追加または削除します。Event Stream Processor を一時停止してから、ブレークポイントを追加します。

1. ブレークポイントをストリームに追加してから、任意のストリームからの入力レコードの処理を開始するには、次のコマンドを使用します。

   ```
   bp add `stream` any
   ```

2. 指定した ID を持つブレークポイントを削除するには、次のコマンドを使用します。

   ```
   bp del `id`
   ```

   ブレークポイントの ID を取得するには、**bp add** コマンドまたは **bp list** コマンドのどちらかを使用します。

3. ブレークポイントをすべて削除するには、次のコマンドを使用します。

   ```
   bp del all
   ```

4. 指定した ID を持つブレークポイントを有効または無効にするには、次のコマンドを使用します。

```
bp on|off `id`
```

5. ブレークポイントをすべて有効または無効にするには、次のコマンドを使用します。

```
bp on|off all
```

6. 指定した ID を持つブレークポイントを n 回目ごとにトリガするには、次のコマンドを使用します。

```
bp every `count` `id`
```

7. すべてのブレークポイントを n 回目ごとにトリガするには、次のコマンドを使用します。

```
bp every `count` all
```

8. 作成されたすべてのブレークポイントをリストするには、次のコマンドを使用します。

```
bp list
```

サンプル・デバッグ：条件ブレークポイントの追加

esp_client コマンド・ライン・ユーティリティには、ブレークポイントに条件を追加するオプションが用意されています。これらのブレークポイントは、条件が true に評価された場合にのみトリガされます。

1. ストリームにブレークポイントを追加してから、別のストリームからの入力レコードの処理を開始するには、次のコマンドを使用します。

```
bp add `stream` `inputStream` [`condition`]
```

2. 入力レコードを処理して、出力を生成してから、ストリームにブレークポイントを追加するには、次のコマンドを使用します。

```
bp add `stream` out [`condition`]
```

データの検査

examine コマンドでは、各種ストリームによって生成されたさまざまなデータ型を確認できます。

検査コマンドでは、各種ストリームで生成されたさまざまなデータ型を確認できます。このコマンドは、Event Stream Processor が一時停止している場合にのみ動作します。

検査コマンドは、通常のサブスクライバへの更新の送信に使用されるフォーマットと同じフォーマットでレコードを返します。**esp_client** コマンドでは、レコードは XML フォーマットで生成されます。 検査対象のデータで発生するオペレーション・タイプには、通常のサブスクライバで確認される標準タイプと、Event Stream Processor の内側でのみ使用されるタイプがあります。

検査コマンドから返されるレコードは、次の 2 つの方法でグループ化されることがあります。

- 2 つのレコードを 1 つの更新ブロックにグループ化できます。これらは、**esp_client** によって XML 要素 <pair> 内に出力されます。
- 複数のレコードと更新ブロックを 1 つのトランザクションにグループ化できます。これらのトランザクションは、**esp_client** によって XML 要素 <trans> 内に出力されます。 トランザクション内のレコードが 1 つのみの場合、<trans> ラッピングは使用されず、1 つのレコードとして出力されます。

ストリームで入力ウィンドウが使用されている場合は、このウィンドウが満杯になると、以前のレコードに対する SAFEDELETE の生成が開始されます。 入力ストリームから送信される DELETE とこれらのレコードを区別するため、**esp_client** は、各レコードに擬似フィールド ESP_RETENTION=1 を挿入します。

次の引数によって、検査対象のデータが選択されます。

引数	機能
kind	検査対象のデータの種類を指定します。
stream	データの取得元のストリームを指定します。すべてのストリームからデータを取得するには、このフィールドを空にしておきます。
object	特定のデータ単位を選択します。この種類のデータの単位が多い場合 (たとえば、変数の場合)、これを使用します。

examine コマンドで指定されたデータの種類に当てはまらない場合は、ストリームかデータまたはその両方を空のままにするか除外できます。 種類とストリームは一致する必要があります。ストリームにはプラットフォームワイドなデータを要求できません。また、stream 引数に値が指定されていない場合はストリームごとのデータを要求できません。

一部の種類のデータは、特定のストリームからのみ入手できます。 たとえば、パターン状態はパターン・ストリームからのみ読み取ることができます。 要求された種類のデータを特定のストリームで入手できないと、他のストリームでその種類のデータがサポートされていても、エラー `No such kind of data` が返されます。

stream 引数の有無に関係なく使用できるデータの種類もあります。たとえば、グローバル変数を検査する場合はストリームを指定せずに "var" を使用し、ストリームの変数を検査する場合はそのストリームを指定して "var" を使用できます。

入力キューに関連するデータのグループは異種です。各レコードは、そのレコードを生成したストリームの名前と一致するタグを受け取ります (ソース・ストリームから生成されたレコードは、ソース・ストリーム自体の名前を持つタグを受け

取ります)。 タグには、queue、inTrans、inRow、queueHead、queueTail、inHist、lastInTrans、inHistEarliest、inHistLatest があります。

また、入力データと出力データの混合からなる履歴データのグループもあります。これらの各グループには、トランザクションのペアが 1 つ以上含まれます。 各ペアの 1 つ目のレコードが入力トランザクションで、2 つ目のレコードが対応する出力トランザクションです。 各入力トランザクション・レコードには、そのレコードを生成したストリームの名前のタグが付きます。各出力トランザクション・レコードは、タグ currow を取得します。 入力ストリーム・レコードにローのタグも付いている場合は、表示される順序を見て識別できます。

処理された入力トランザクション、生成された出力など、履歴を扱う特定の種類のデータがあります。 これらのデータ・セットは別々に (入力履歴と出力履歴として) 取得することも、組み合わせて取得することもできます。入力履歴と出力履歴を別々に検査する場合、トランザクションの一致にはそのインデックスが使用されます。 たとえば、1 つ目の入力トランザクションは 1 つ目の出力トランザクションと一致します。

ストリームで保持される履歴データの量は、ストリームの履歴サイズの設定によって決まります。この設定は、**esp_client** コマンド履歴を使用して、すべてのストリームに対しグローバルに設定することも、個々のストリームに対し設定することもできます。デフォルトの履歴サイズ制限は 100 トランザクションです。 使用する履歴制限が大きいと、Event Stream Processor のメモリ使用量が増加します。

トレース・モードがオフの場合、履歴はすべて破棄されますが、制限は維持されます。トレース・モードを次に有効にしたときに、履歴の収集が再開されます。

examine コマンドから返されたレコードは空のプレースホルダに格納します。トランザクション境界に (たとえば、outTrans に) あいまいさがないと、**esp_client** はデータを返しません。 **hist** など、その他の場合は、トランザクションは発生したが、出力は生成されなかったことがプレースホルダに示されます。 プレースホルダ・レコードには、キー・フィールドを含め、すべてのフィールドが含まれます。各値には NULL が設定されます。

データ・レコードの一部のタイプでは、自然なフィールド名が使用されています。"pause"などのレコードはメタデータを返します。 このタイプのレコードのフィールド名は、Event Stream Processor で定義されます。その他のタイプには、ユーザで定義されたフィールドと Event Stream Processor で追加されたフィールドの両方があります。 このタイプのレコードでは、Event Stream Processor で追加されたフィールドにプレフィクス ESP_ が付きます。 ユーザ定義のレコードにはこのプレフィクスを使用しないでください。

フィルタ

データの量が多い場合は、フィルタを使用して特定のローを検査します。デバッグ・コマンドから返されたデータを絞り込み、関連するデータのみを選択できます。

データは Event Stream Processor でフィルタリングされてから送信されます。

esp_client コマンド **exf** は、次のようにフィルタリングを実行します。

```
exf {kind} [ {stream} [ {object} ] ] {expr}
```

ex コマンドと同様に、ストリームとオブジェクトの名前はオプションです。その他に、フィルタ式を入れる引数があります。 現在のローを入れるブレークポイント・フィルタ式と同様に、フィルタ式は事前定義変数を参照します。この式は、ローと式を比較し、そのローを返すかどうかを決めます。 フィルタ式が `true (non-0, non-NULL)` を返す場合は必ずレコードがユーザに返されて表示されます。

定義済み変数のルールは、次のとおりです。

- 返されたデータ・セット内のすべてのローが同じタイプの場合、それらのローは 1 つの変数ローにラップされます。
- データ種類にあるローが混合タイプの場合 (入力データまたは履歴データ)、複数の変数が定義され、XML タグと一致する名前がこれらのレコードに対して出力されます。 値があるのは、現在のレコードのタイプと一致する 1 つの変数のみです。 その他にはすべて NULL が設定されます。

ストア・データ

ストア・データは、Event Stream Processor のデバッグ・ツールを使用してファイルに入れます。

この処理は、ストリームの要素にある属性 `ofile` と似ています。この属性では、Event Stream Processor が終了すると、ストリームの内容がファイルにダンプされます。

ストア・データをダンプするには、次のコマンドを使用します。

```
dump { FileName} {streamName}
```

データ操作

esp_client では、**eval** コマンドを使用して、デバッグ・インタフェースを介して Event Stream Processor 内のデータを変更できます。この機能を使用して、グローバル変数とストリーム・ローカル変数の内容 (eventCache を含む) を変更できます。

データ操作機能は、Event Stream Processor がトレース・モードで、一時停止している場合にのみ動作します。

eval コマンドは、ストリームで SPLASH 文 (またはブロック) を評価してデータを変更します。 変数のみを参照でき、ストリームとストリーム反復子は参照できません。 分岐化、ループなど、どの SPLASH 文でも使用できますが、無限ループを作成すると Event Stream Processor はいつまでも停止します。 テンポラリ変数を SPLASH ブロックの内側で使用することもできます。

通常、eventCache のキーは現在の入力レコードによって決まります。 入力レコードがないので、キーは設定されず、eventCache でのオペレーションは効果がありません。 このため、演算子 *keyCache(ec-variable, record)* を使用してキーを手動で設定する必要があります。 必ずこの演算子を設定してから、eventCache でオペレーションを実行するようにします。 キーは、ループ内でも、複数回変更できるので、複数のキーでオペレーションを実行できます。

eval コマンドで変更できるのは、ストリームのローカル変数 (ローカルの **DECLARE** ブロックで定義されている変数) とグローバル変数のみです。ストリームの SPLASH ブロックの内側で定義されている変数は、該当するメソッドが実行される場合にのみ存在します。 これらの変数は変更できません。

あるコード・ユニットを評価できれば、それは式ではなく SPLASH 文です。 SPLASH 文は、";" で終わる単純な式、または中カッコ "{}" で囲まれたブロックのどちらかです。 複数の文は必ず 1 つのブロックで囲みます。 中カッコを使用してブロック引数を囲む場合、外側の中カッコはブロック・デリミタと見なされません。 単なる **esp_client** の引用符です。

次に、正しいブロックと間違ったブロックの例を示します。

正しいブロック

```
eval `stream` `a := 1;`

eval {a := 1;}

eval `stream` `{ typeof(input) r := [ a=9; | b= 's1'; c=1.;
d=intDate(0);];
        keyCache(s0, r); insertCache(s0, r); }`

eval {stream} {{ typeof(input) r := [ a=9; | b= 's1'; c=1.;
d=intDate(0);];
        keyCache(s0, r); insertCache(s0, r); }}
```

間違ったブロック

```
eval `stream` `a := 1`

eval {a := 1}
```

```
eval `stream` `typeof(input) r := [ a=9; | b= 's1'; c=1.;
d=intDate(0);];
        keyCache(s0, r); insertCache(s0, r);`

eval {stream} { typeof(input) r := [ a=9; | b= 's1'; c=1.;
d=intDate(0);];
        keyCache(s0, r); insertCache(s0, r); }
```

第 7 章　オンデマンド・クエリ

Event Stream Processor の SQL 構文

ESP オンデマンド・クエリ・インタフェースでは、esp_query ツールを使用して、実行中の ESP プロジェクトのアクティブなウィンドウにクエリを実行することができます。

これらの SQL クエリはいつでも実行でき、アドホックです (たとえば、事前に定義する必要はありません)。また、クエリの実行時のウィンドウの内容に基づいてスナップショットを返します。これは重要な特徴です。継続クエリである CCL クエリと異なり、オンデマンド・クエリ・インタフェースを介して発行されるクエリはスナップショット・クエリです。

クエリは、実行中の ESP プロジェクトの出力ウィンドウにのみ実行できます。ストリームとデルタ・ストリームには状態がないため、オンデマンド・クエリは実行できません。また、ローカル・ウィンドウにはクエリを実行できません。これは、ローカル・ウィンドウはプロジェクトの内部にあり、プロジェクトの外側では参照できないためです。

esp_query と、オンデマンド・クエリでサポートされる SQL 構文を使用して、ESP プロジェクトにクエリを実行します。

DELETE 文

入力ウィンドウから、指定されたローを削除します。

構文

```
DELETE FROM window  [ WHERE whereclause ]
```

コンポーネント

window	入力ウィンドウ。
whereclause	true の場合は 1、false の場合は 0 に評価される式。

使用法

DELETE 文は、入力ウィンドウから指定されたローを削除します。ストリームでは使用できません。**WHERE** 句は、削除対象のローを絞り込むフィルタの役目を果

たします。**WHERE** 句が指定されなかった場合は、入力ウィンドウのすべてのローが削除されます。

例

```
DELETE Trades WHERE Shares < 100 and Symbol = 'SAP';
```

INSERT 文

1 つ以上のローを作成し、指定されたストリームまたはウィンドウにそれらのローを挿入します。

構文

```
INSERT INTO StreamWindow ( column [,...] ) VALUES ( value [,...] )
```

コンポーネント

StreamWindow	入力ストリームまたは入力ウィンドウの名前。
column	指定されたストリームまたはウィンドウにあるカラムの名前。
value	挿入対象の対応する値。指定されたカラムの数と同じ数の値が必要です。

使用法

INSERT 文は、データのローを生成し、送信先ストリームまたはウィンドウにそれらのローをパブリッシュします。文が実行されて出力が生成されるたびに、指定された送信先にローがパブリッシュされます。ステートメントの実行時期を制御するには、省略可能な **OUTPUT** 句を追加できます。

INSERT 文は、データのパブリッシュ先のカラムを左から右に明示的に指定します。**INSERT INTO** 句に指定するカラムの数とデータ型は、**VALUES** 句のカラム式の項目の数とデータ型に対応します。

送信先ストリームまたはウィンドウに、指定した数よりも多いカラムが含まれていると、ローがパブリッシュされるときにリストにないカラムに NULL が設定されます。

例

```
INSERT INTO Trades (TradeId, Symbol, Shares, Price) VALUES (1000,
'SAP', 100, 75.50)
```

SELECT 文

入力ウィンドウまたは出力ウィンドウの内容に対してクエリを実行します。これは、実行中のプロジェクトの内容に対してクエリを実行する場合の構文です。プロジェクトを作成する場合の有効な構文ではありません。

SELECT 文には、1 つ以上の **SELECT** 句と 1 つ以上の **FROM** 句が必要ですが、**WHERE** 句、**GROUP BY** 句、**ORDER BY** 句など、その他の句を必要に応じて入れる場合もあります。

構文

```
SELECT { TOP N { col1[,...] | * } } | {[DISTINCT] { expression [[AS]
alias] } [, ...]}
FROM { out_window }
[ WHERE expression ]
[ GROUP BY expression [, ...] ]
[ ORDER BY column [ ASC[ENDING] | DESC[ENDING] ] [, ...] ]
```

コンポーネント

N	ローの数。
式	必要に応じて、選択条件、グループ化条件、またはフィルタリング条件を指定する SQL-92 式。詳細については、「使用法」を参照してください。
out_window	入力／出力ウィンドウの名前。ウィンドウがあるモジュールへのパスで、名前と区切り文字のドットを使用して、サブモジュールを指定します。 たとえば、module1.out_window1 のように入力します。
column	**GROUP BY** 句で出力の編成に使用するカラムの名前。

使用法

SELECT 文は、**FROM** 句にリストされている入力／出力ウィンドウの現在の内容に対してクエリを実行し、それぞれ決められた数のカラムがある結果ローを生成します。クエリの **WHERE** 句によるフィルタリング (指定されている場合) が実行されてから、**FROM** 句にリストされている入力／出力ウィンドウのローが **SELECT** 句に渡されます。これらの結果が、クエリのその他の句で処理されます。

SELECT 文には次の句があります。

- **SELECT** 句
- **FROM** 句
- (省略可能) **WHERE** 句
- (省略可能) **GROUP BY** 句

SELECT 句

各 **SELECT** 文には、1 つの **SELECT** 句が必要です。**SELECT** 句では select リストを指定します。このリストを使用してクエリの結果が生成されます。select リストには、次のような多くのさまざまな機能があります。

- 単純な **SELECT** 文の select リストにある項目の数によって、結果でのカラムの数が決まります。
- select リスト式は、**SELECT** 文の **FROM** 句で指定される出力ウィンドウのカラム名を参照できます。
- アスタリスク (*) 文字を使用すると、**FROM** 句で指定される入力／出力ウィンドウのすべてのカラムを選択できます。
- select リストの前に DISTINCT キーワードを挿入して、結果に含める各ローをユニークにすることができます。2 つ以上のローでクエリ対象のすべてのカラムの値が同じ場合、DISTINCT キーワードを指定すれば結果に含まれるローは 1 つのみになります。それ以外の場合は、基準を満たすすべてのローが結果に含まれます。

また、必要に応じて、`rowtime` または `rowid` を **SELECT** 句に含めて、出力に rowtime と rowid を表示することもできます。

FROM 句

FROM 句では、**SELECT** 文のクエリのデータ・ソースを指定します。データ・ソースは出力ウィンドウです。出力ウィンドウは、**CREATE WINDOW** 文での定義済みの名前で参照できます。

WHERE 句

省略可能な **WHERE** 句では、選択条件を指定できます。選択条件として、**WHERE** 句はデータ・ソースからローをフィルタリングしてから SELECT 句に渡します。**WHERE** 句の式では集合関数を使用できません。

GROUP BY 句

省略可能な **GROUP BY** 句を指定した場合は、結果の 1 つ以上のローが組み合わされて出力の 1 つのローになります。**GROUP BY** 句は、クエリ結果に集合関数がある場合によく使用されます。 この句では、定数を使用している式や入力ウィンドウまたは入力ストリームからの式は使用できますが、集合関数がある式は使用できません。

例

この例では、シンボル 'SAP' を持つすべての取引が Trades から選択されます。

```
SELECT TradeId, Symbol, Shares
FROM Trades
WHERE Symbol = 'SAP'
```

この例では、シンボル 'SAP' を持つすべての取引が選択され、シンボルを基準としてグループ化されてから、価格を基準としてグループ化されます。

```
SELECT Symbol, Price, sum(Shares)
From Trades
where Symbol = 'SAP'
GROUP BY Symbol, Price
```

UPDATE 文

入力ウィンドウ内の既存のローを更新します。

構文

```
UPDATE window SET { column=value [,...] } [ WHERE whereclause ]
```

コンポーネント

window	入力ウィンドウ。
whereclause	true の場合は 1、false の場合は 0 に評価される式。
value	指定されたカラムと同じデータ型の値に評価される式。
column	値のパブリッシュ先である送信先カラムの名前。

使用法

UPDATE 文は、入力ウィンドウ内の既存のローを更新します。ストリームでは使用できません。**UPDATE** 文は、**WHERE** 句が指定された場合はその句で識別されたローを更新し、この句が指定されなかった場合はすべてのローを更新します。**UPDATE** 文で指定されなかったカラムにはすべて自動的に NULL が設定されます。

例

```
UPDATE Trades SET Price = 0.0 WHERE Symbol <> 'SAP'
```

サポートされる SQL-92 式

SQL での式の参照でサポートされる構文について説明します。

SQL での式の参照は、特に明記されていない限り、次の構文を指します。

```
{expression binary expression} |{expression [NOT] LIKE expression}|
{unary expression} |(expression) |column |pub.column | literal |
parameter |
{function ( expression | * ) } |{ expression { IS NULL | IS NOT
NULL } } |
{expression [NOT] IN ( values )} |
{CASE [expression] { WHEN expression THEN expression } [...] [ELSE
expression] END}|
```

2 項

```
 | * | / | % | + | - | | | < | <= | >= | = | <> | IN | AND | OR
```

単項

```
- | + | NOT
```

esp_query

標準入力または -Q オプションで提供される SQL 文をサーバ上で実行し、その結果を標準出力に出力します。

構文

```
esp_query -p host:port/workspace/project [OPTION...]
```

必須の引数

- **-p *host:port/workspace/project*** – *host:port/workspace/project* のすべての引数で、ESP サーバ (クラスタ・マネージャ) に接続する場合の URI を指定します。たとえば、ESP クラスタ・サーバを起動したときのポートがローカル・マシンの [19011]で、デフォルトのワークスペースで [prj1] プロジェクトを実行している場合は、-p を次のように指定します。 `-p localhost:19011/default/prj1`

オプション

- **-c *user[:password]*** – 認証に使用する *user* ID と、オプションの *password* を指定します。 *password* が指定されていないと、ユーザにパスワードの入力を求めるプロンプトが表示されます。 Event Stream Processor でこれらのクレデンシャルを使用して認証が正常に行われた場合は、接続が維持されます。それ以外の場合は、Event Stream Processor はすぐに接続を切断します。
- **-d <*database*>** – Event Stream Processor に接続するときのデータベース名を指定します。 デフォルトは "database" です。
- **-e** – Event Stream Processor への暗号化された SSL 接続を使用することを指定します。
- **-G** – SASL/GSSAPI を使用する認証を指定します。
- **-h** – 使用可能なオプションのリストを各オプションの簡単な説明とともに画面に出力します。
- **-k <*path*>** – パスワード認証の代わりに、RSA プライベート・キー・ファイルのメカニズムを使用して認証を実行します。 *privateRsaKeyFile* には、プライベート RSA キー・ファイルのパス名を指定します。 ESP サーバの起動時に -k

オプションで RSA キーの保存先のディレクトリを指定したことを確認してください。

注意： このオプションが有効な場合、ユーザ名は `-c` オプションで指定する必要がありますが、パスワードは必要ありません。

- **`-m`** <***date format***> – `strptime` フォーマットを使用して、日付値のフォーマットを設定します。 デフォルトは "%Y-%m-%d %H:%M:%S+00" です。
- **`-P`** <***precision***> – 出力での小数点以下の桁数を指定します (デフォルトは 2 です)。
- **`-Q`** <***query***> – 実行するクエリを指定します (stdin を使用して指定されていない場合)。
- **`-q`** ***hostname:port*** – ターゲット ESP サーバの SQL リスナのホスト名とポートを指定します。 デフォルトのホストは "localhost" で、デフォルトのポートは "22200" です。
- **`-t`** <***table name***> – XML 出力のテーブルの名前を指定します。 デフォルト名は "Result" です。
- **`-v`** – **esp_query** ユーティリティのバージョンを出力します。

使用法

esp_query は、標準入力で SQL クエリを受け取り、それを Event Stream Processor の実行インスタンスに転送します。 その後、クエリの結果を標準出力に出力します。

UNIX または Linux のコマンド・ライン・プロンプトまたは ESP スタジオの [Query] パネルで SQL クエリを指定する場合は、二重引用符で囲みます。 DOS コマンド・ライン・プロンプトでは、SQL クエリを二重引用符で囲まないでください。

ストリームに対してクエリを実行しても意味はありませんが、エラー・ストリームの状態を保持するダウンストリーム・ウィンドウが定義されている場合は、**esp_query** を使用して、エラー・ストリームから情報を取得できます。

例

SQL ポート 11100 を使用して、UNIX マシン myhost でストリーム *Emp* の内容を出力するには、次のように入力します。

```
echo "select * from Emp" | ./esp_query -p myhost:11100/workspace/
project -c user:password
```

myhost が Windows マシンの場合、**esp_query** の構文は同じですが、クエリを二重引用符で囲まずに、次のように入力します。

```
echo select * from Emp | esp_query -p myhost:11100/workspace/project
-c user:password
```

Dept ストリームからエントリを削除し、それに応じて *Emp* ストリームを更新するには、次のように入力します。

```
echo "delete from Dept where dn='SWP'; update Emp set dn='' where
dn='SWP'" | ./esp_query -p myhost:11100/workspace/project -c
user:password
```

エラー・ストリーム ErrorStream の状態を保持する ErrorState ウィンドウに対してクエリを実行するには、次のように入力します。

```
echo "select e.*, recordDataToString(e.sourceStreamName,
e.errorRecord ) errorRecord from errorState e" |esp_query -p myhost:
11100/workspace/project -c user:password
```

SQL 構文

Event Stream Processor が受け入れるのは、SQL92 文のサブセットです。

Event Stream Processor が受け入れる SQL92 文は、**select** 文、**insert** 文、**update** 文、**delete** 文です。

select を使用するクエリは、1 つのストリームに制限され、ジョインもサブクエリもサポートされていませんが、**where**、**group by**、**order by** の各句が使用される場合があります。 **insert**、**update**、**delete** の各文はソース・ストリームに制限されます。 これらの変更に関する文はセミコロンで区切って順に挿入できます。

索引

E