Utilities Guide

Sybase Event Stream Processor 5.1

Contents

CHAPTER 1 Overview of Event Stream Processor Executables

Sybase® Event Stream Processor provides executables that give you command line control of your Event Stream Processor projects.

Event Stream Processor Server Executables

There are four Event Stream Processor executables: **esp_server**, **esp_cluster_admin**, **esp_monitor**, and **esp_playback**.

- **esp_server –** initiates the ESP Server.
- **esp_cluster_admin –** interacts with the cluster manager, in either interactive mode or command line mode, and sets up your project environment.
- **esp_monitor –** receives and displays performance data from a running instance of the Event Stream Processor. The performance data is written to standard output.
- **esp_playback –** records in-flowing data to a playback file, and plays the captured data back into a running Event Stream Processor instance.

Command and Control Executables

There are two Command and Control executables: **esp_cnc**, and **esp_client**.

- **esp_cnc –** executes individual control commands. This utility is supplied with the source, to serve as an example of writing client applications.
- **esp_client –** queries metadata, injects rows of data, makes stream snapshots, and controls a running Event Stream Processor Server instance. It also stops the ESP Server, fetches the list of streams and their definitions, and determines the host and port of the Gateway interface.

Publish and Subscribe Executables

There are six publisher and subscriber executables: **esp_convert** , **esp_kdbin**, **esp_kdbout**, **esp_rapexport**, **esp_subscribe**, and **esp_upload** .

- **esp_convert –** reads XML or delimited records from standard input and produces binary format records on standard output.
- **esp_kdbin –** reads data from a KDB database table into an Event Stream Processor stream.

- **esp_kdbout –** feeds streaming data from the Event Stream Processor to a KDB database table.
- **esp_rapexport –** runs an adapter to publish data from the Event Stream Processor to Sybase RAP - The Trading Edition.
- **esp_subscribe –** connects to a running instance of the ESP Server and subscribes to streaming data. The received records are converted to XML (or optionally delimited format) and written to standard output.
- **esp_upload –** reads binary records from standard input and publishes them to a current instance of the Event Stream Processor through the Gateway interface.

Authoring Executables

There are two authoring executables: **esp_compiler** and **esp_studio**.

- **esp_compiler –** translates a given set of CCL statements to the corresponding CCX representation to be consumed by the Event Stream Processor. Also, verifies the correctness of the CCL statements, checks for datatype consistencies, and performs limited optimization.
- **esp_studio –** this shell script launches the ESP Studio, a graphical environment you can use to author Event Stream Processor projects, and start and monitor the ESP Server.

CHAPTER 2 Server Executables

Use the command-line utilities to start the ESP Server with the desired configuration.

esp_server

A shell script that starts the ESP Server.

Syntax

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

Options

- **`-cluster-node`**<***file***> **–** (optional) specifies which node configuration file to use. For information about configuring cluster nodes, see the *Administrators Guide*.
- **`-cluster-log-properties`**<***file***> **–** (optional) specifies the cluster logging property file to use. For information see the *Administrators Guide*
- **`-h`** **or** **`--help`** **–** (optional) prints a list of possible options on the screen along with a brief explanation for each option.
- **`-v`** **or** **`--version`** **–** (optional) prints the **esp_server** utility version.

Usage

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

esp_cluster_admin

A command line utility you can use to interact with the cluster manager, in either interactive mode or command line mode.

The **esp_cluster_admin** utility supports several commands you can use to set up your project environment, encrypt sensitive data, and deploy a keystore. To access the utility, provide your user name and password, unless you chose not to set up authentication.

Once you log into the ESP Server, you can run commands on the server side continually until you exit by executing **exit** or **quit**.

Note: In interactive mode, you can log in and execute commands continuously; the utility maintains the session with the cluster manager. In command line mode, the utility logs you out after each command is run; you must log in again to perform the next operation.

By default, the cluster generates `TIMEOUT` after 20 seconds. You can set timeout in interactive mode or command line mode to avoid the `TIMEOUT` option appearing after 20 seconds. The SDK provides the timeout value (in seconds) for the **START PROJECT** or **STOP PROJECT** commands. If the command returns `TIMEOUT`, obtain the project status using the **GET PROJECT** or **GET PROJECTS** command.

Syntax

Call **esp_cluster_admin**:

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

If the cluster node is SSL enabled, call **esp_cluster_admin**:

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

To administer clusters with RSA authentication, call the **esp_cluster_admin**:

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

To administer clusters with Kerberos or LDAP authentication, call the **esp_cluster_admin**:

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

Commands

- **get managers –** lists the cluster managers.
- **get controllers –** lists the cluster controllers.
- **get workspaces –** lists all existing workspaces.
- **get projects –** lists all existing projects independent of a workspace.
- **get project <workspace-name>/<project-name> –** shows a project within an associated workspace.

  Enter the command followed by the workspace name, a forward slash, and the project name. Do not use spaces in the variable entries.

  For example, to show the project `sample` from the workspace `tradespace`, enter:

  ```
  get project tradespace/sample
  ```
- **get streams <workspace-name>/<project-name> –** shows all streams within a project.

  Enter the command followed by the workspace name, a forward slash, and the stream name. Do not use spaces in the variable entries.

  For example, to show the stream `tradedata` from the workspace `tradespace`, enter:

  ```
  get streams tradespace/tradedata
  ```
- **get schema <workspace-name>/<project-name> <stream-name> –** shows the schema of an associated stream.

  Enter the command followed by the workspace name, a forward slash, and the schema name. Do not use spaces in the variable entries.

For example, to show the schema (`dataformat`) for `tradedata`, enter:

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

- **add workspace <workspace-name> –** adds a new workspace.

  Enter the command followed by the workspace name. Do not use spaces in the variable entries.

  For example, to add the workspace `stockspace`, enter:

  ```
  add workspace stockspace
  ```

- **add project <workspace-name>/<project-name> <ccx> [<ccr>] –** adds a project to a workspace.

  Enter the command, followed by the workspace name to which to add the project, a forward slash, followed by the project name and file name with extension. Do not use spaces in the variable entries.

  For example, to add the project `tradeanalysis` with the same file name to the workspace `tradespace`, enter:

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

  The `ccr` file is the project configuration file for the `ccx` file.

- **remove workspace <workspace-name> –** removes a workspace.

  Enter the command, followed by the name of the workspace to remove. Do not use spaces in the variable entries.

  For example, to remove the workspace `stockspace`, enter:

  ```
  remove workspace stockspace
  ```

- **remove project <workspace-name>/<project-name> –** removes a project.

  Enter the command, followed by the name of the project to remove. Do not use spaces in the variable entries.

  For example, to remove the project `tradeanalysis` from the workspace `tradespace`, enter:

  ```
  remove project tradespace/tradeanalysis
  ```

- **start project <workspace-name>/<project-name> [timeout(sec)] [<instance-index>] –** starts a project.

  Enter the command, followed by the workspace name, a forward slash, and the name of the project to start. Do not use spaces in the variable entries.

  For example, to start the project `sample` from the workspace `tradespace`, enter:

  ```
  start project tradespace/sample
  ```

- **stop project <workspace-name>/<project-name> [timeout(sec)] [<instance-index>] –** stops a project.

  Enter the command, followed by the workspace name, a forward slash, and the name of the project to stop. Do not use spaces in the variable entries.

For example, to stop the project `sample` from the workspace tradespace, enter:

```
stop project tradespace/sample
```

- **stop node <node-name> –** stops a node (controllers or managers, or both).

  Enter the command, followed by the node name.

  Example

  ```
  stop node node1
  ```

- **encrypt <clear-text> –** encrypts plain text data.

  Enter the command, followed by the sensitive data to encrypt. When you run the command, the utility produces encrypted text you can use to replace the sensitive data in the associated file.

  For example, to encrypt the password `1234,` enter:

  ```
  encrypt 1234
  ```

- **deploykey <new-username> <keystore> <storepass> <key-alias> [<store-type>] –** adds a new user by deploying a new user key to the keystore.

  Enter the command, followed by the new user name, keystore file path, and storepass key-alias.

  All nodes in a cluster must share the same keystore file path. The node to which the deploy command is sent updates the keystore and the other nodes then reload that file. To test if the deploy key is working properly, log in to the cluster with the new key, but through a different node.

  For example, to deploy the new key `123456` under the new user name `jdoe`, with store key-alias `jdoe`, enter:

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

- **reload policy –** reloads the `policy.xml` file in a running cluster.

  If you have recently updated the existing policy file, the cluster is reverified against the new policy configuration upon reload.

- **connect –** reconnects the utility to the cluster it was started with if the connection has expired. This command is in interactive mode only.
- **quit or exit –** logs you out of interactive mode. To reaccess the utility, provide your user name and password, unless you chose not to set up authentication.
- **help –** retrieves a plain-text description of utility commands and usage information for interactive mode.

esp_cluster_admin in Command Line Mode

Run command line operations for the **esp_cluster_admin** utility supported commands.

Run one command for each instance of the **esp_cluster_admin** call. Repeat the **esp_cluster_admin** call as many times as needed.

Syntax

Run **esp_cluster_admin** commands for command line mode:

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

If the cluster node is SSL enabled, call **esp_cluster_admin**:

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

To administer clusters with RSA authentication, call **esp_cluster_admin**:

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

To administer clusters with Kerberos or LDAP authentication, call **esp_cluster_admin**:

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

Options

- **--get_managers –** lists the cluster managers.
- **--get_controllers –** lists the cluster controllers.
- **--get_workspaces –** lists all existing workspaces.
- **--get_projects –** lists all existing projects independent of a workspace.
- **--get_projectdetail –** enter the command followed by the workspace name, a forward slash, and the desired project name.

  For example, to show project details of project `project1` from workspace `ws1`, enter:

  ```
  esp_cluster_admin --uri=esp://localhost:19011 --username=user --
  password=pass --get_projectdetail --workspace-name=ws1 --project-
  name=project1
  ```
- **--get_streams –** enter the command, followed by the workspace name, then the desired project name.

  For example, to show a stream of project `project1` from workspace `ws1`, enter:

  ```
  esp_cluster_admin --uri=esp://localhost:19011 --username=user --
  password=pass --get_streams --workspace-name=ws1 --project-
  name=project1
  ```
- **--get_schema –** enter the command, followed by the workspace name, the project name, then the stream name.

  For example, to show a schema of a stream`win` of project `project1` from workspace `ws1`, enter:

  ```
  esp_cluster_admin --uri=esp://localhost:19011 --username=user --
  password=pass --get_schema --workspace-name=ws1 --project-
  name=project1 --stream-name win
  ```
- **--add_workspace –** enter the command, followed by the workspace name.

For example, to add a workspace `ws2`, enter:

```
esp_cluster_admin --uri=esp://localhost:19011 --username=user --
password=pass --add_workspace --workspace-name=ws2
```

- **--add_project –** enter the command, followed by the workspace name, the project name, then the project file name with extension.

  For example, to add project `project1` to workspace `ws2`, enter:

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

- **--remove_project –** enter the command, followed by the workspace name, then the project name.

  For example, to remove project `project1` from workspace `ws1`, enter:

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

- **--start_project –** enter the command, followed by the workspace name, then the project name.

  For example, to start project `project1` in workspace `ws1`, enter:

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

- **--stop_project –** enter the command, followed by the workspace name, then the project name.

  For example, to stop project `project1` in workspace `ws1`, enter:

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

- **--stop_node –** enter the command, followed by the node name.

  For example, to stop node `node1`, enter:

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

- **--encrypt_text –** enter the command, followed by the sensitive data you want to encrypt.

  For example, to encrypt text `1234`, enter:

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

- **--reload_policy –** reloads the `policy.xml` file.
- **--deploy_key –** enter the command, followed by the new username, keystore filepath, and storepass key-alias.

  For example, to deploy a new storepass key-alias `user` to keystore `mykeystore.jks`, enter:

```
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 –** retrieves plain-text description of utility commands and usage information.

esp_monitor

Reads performance data from a running instance of Event Stream Processor and prints it out in XML format on standard output.

Monitoring data is available only if the time granularity option is set in the project configuration (CCR) file. The time granularity option specifies, in seconds, how often the set of performance records —one per stream and one per gateway connection— is obtained from the running Event Stream Processor. The **_ESP_Clients_Monitor** stream contains basic information about the connected clients but performance-related fields are populated only with the monitoring option.

A record in this format is produced for each stream.

```
<_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 –** holds the opcode for the record.
- **stream –** contains the name of the stream for which statistics are reported.
- **cpu_pct –** the CPU utilization percentage over the last reporting interval.
- **trans_per_sec –** the transaction rate for this interval.
- **rows_per_sec –** the rate of row arrivals for this interval.
- **inc_trans –** the number of transactions for this interval.
- **inc_rows –** the number of new rows for this interval.
- **queue –** the number of records in the queue for the stream.
- **store_rows –** the number of rows in the table.
- **last_update –** the date/time of the last update.
- **sequence –** the sequence number of the update (redundant, since the stream name and last_update already provide unique identification).
- **posting_to_client –** the handle of the gateway client where the stream was trying to post data at the moment, or -1 if none.

A record in this format is produced for each gateway client.

```
<_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 –** the opcode for the record.
- **handle –** the handle of this gateway client.
- **user_name –** the user name of this client.
- **ip –** the address from which this client is connected.
- **host –** the host name from which this client is connected, if resolvable.
- **port –** the port from which this client is connected.
- **login_time –** the timestamp when this client logged in.
- **conn_tag –** the connection tag, if any.
- **cpu_pct –** the CPU utilization percentage over the last reporting interval by this client's gateway thread.
- **last_update –** the date/time of the last update.
- **subscribed –** (1) if this client has subscribed or (0) if not.
- **sub_trans_per_sec –** the subscription transaction rate for this interval; envelopes and service messages are included in the count.
- **sub_rows_per_sec –** the subscription row rate for this interval.
- **sub_inc_trans –** the number of subscription transactions/envelopes/messages for this interval.
- **sub_inc_rows –** the number of subscription rows for this interval.
- **sub_total_trans –** the total number of subscription transactions/envelopes/messages sent.
- **sub_total_rows –** the total number of subscription rows sent.
- **sub_dropped_rows –** the number of subscription rows dropped due to the client not keeping up.
- **sub_accum_size –** for pulsed subscriptions, the current number of rows collected in the accumulator, to be sent in the next pulse.

- **sub_queue –** the number of records in the "proper queue" for this client (the total amount of data buffered consists of **sub_accum_size**, **sub_queue**, and **sub_work_queue**).
- **sub_queue_fill_pct –** contains the size of **sub_queue** in percent relative to its limit.
- **sub_work_queue –** the number of records being transferred from the queue to the socket buffer.
- **pub_trans_per_sec –** the publish transaction rate for this interval; the envelopes and service messages are also counted equal to transactions.
- **pub_rows_per_sec –** the publish row rate for this interval.
- **pub_inc_trans –** the number of publish transactions, envelopes, or messages for this interval.
- **pub_inc_rows –** the number of publish rows for this interval.
- **pub_total_trans –** the total number of publish transactions, envelopes, or messages received.
- **pub_total_rows –** the total number of publish rows received.
- **pub_stream_id –** -1 if the publisher cannot write to the stream to which it is currently trying to publish, otherwise it is the numeric Id of that stream.

Syntax

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

Required Arguments

- **`-c`** ***user[:password]* –** (required) authenticates with a *user* ID and, optionally, a *password*. If you do not provide either the *password* or the **-k** option, the user is prompted for the password. If Event Stream Processor successfully authenticates with these credentials, the connection is maintained, otherwise Event Stream Processor immediately closes the connection.
- **`-p`** ***[<host>:]<port>/workspace-name/project-name* –** (required) together, the *host:<port>/<workspace name>/<project name>* arguments specify the URI to connect to the ESP Server (cluster manager). For example, if you have started your ESP cluster server in the port **19011**, the host name is set as **localhost**, and you are running a project called **prj1** in the default workspace, specify `-p` as: `-p localhost:19011/default/prj1`

Options

- **`-k`** ***privateRsaKeyFile* –** (optional) performs authentication using the RSA private key file mechanism instead of password authentication. The **privateRsaKeyFile** must specify the pathname of the private RSA key file.

Note: Ensure that the ESP Server has been started with the `-k` option specifying the directory in which to store the RSA keys.

Examples

- **Monitoring an Instance of Event Stream Processor –** To monitor an instance of the Event Stream Processor running on the host "myhost.sybase.com" with a Command and Control port of 31415, use:

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

esp_playback

Loads data into the Event Stream Processor from a variety of sources at the specified rate.

Use esp_playback to test data models. It can send data with a play rate, and upload data to a server from different sources.

Currently supported formats are ESP XML, comma-separated values, ESP binary, ESP recorder, KDB, and ODBC sources.

Note: The ODBC driver supports Sybase IQ and ASE databases. You can also connect to other ODBC-compatible data sources such as Oracle, DB2, and MySQL.

This tool automatically displays updates regarding the status of the data upload into ESP when the command has been initiated. Each update displays (in order): a time stamp (in square brackets) in microseconds, the number of events uploaded into ESP, if any errors have occurred, and the total percentage of events uploaded so far. By default, the esp_playback tool displays an update every 5 seconds, but you can use the -r option to specify a custom interval. Below is an example where output is set to report every 2 microseconds:

```

[1329258744] started
[1329258744] complete: 0 (success) 0 (errors) 0 (%)
[1329258746] complete: 1 (success) 0 (errors) 33 (%)
[1329258748] complete: 2 (success) 0 (errors) 66 (%)
[1329258750] complete: 3 (success) 0 (errors) 100 (%)
[1329258750] stopped
```

When a timestamp accompanied with a ‘stopped’ output is displayed, the utility has finished uploading events into ESP. If any errors occur during the upload, an error message indicates the issue.

You can also slow down or speed up the rate of the playback. Use the `-R` option to specify the desired rate, either in rows/millisecond, or at a rate determined by the values in a timestamp/ datetime column in input data. If you use the latter method, you can specify a timescale rate to speed up or slow down the playback.

Because **esp_playback** is a testing tool, there are various reasons for slowing down or speeding up the playback. Typically, you adjust the playback rate to verify your calculations and collection of data. When you're confident that your project is configured correctly, you can speed up the playback rate to simulate the anticipated incoming rate of actual data to ensure your project can handle the throughput.

Note that the input speed may influence your calculations. The example below demonstrates how modifying input speed can change the results of calculations due to timing discrepancies. The CCL code below sums the number of shares, `numShares`, and averages the price, `price`, of the input from the `cbret_006` window.

```
create schema sharesSchema (
    tradeId integer,
    symbol string,
    tradeDate date,
    numShares integer,
    price float
);

CREATE  MEMORY  STORE "store" PROPERTIES  INDEXTYPE ='tree',
INDEXSIZEHINT =8;

create input window cbret_006a
    schema sharesSchema
    primary key(tradeId)
    STORE "store"
;
create output window cbret_006
    primary key deduced
    as
    select a.tradeId, a.symbol, a.tradeDate,
        sum(a.numShares) as numShares,
        avg(a.price) as price
        from cbret_006a a
        keep 1 seconds per (symbol)
        group by a.symbol
;
```

A sample input of three lines of XML records to be read.

```
<cbret_006a ESP_OPS="i"  tradeId="1" symbol="EBAY"
tradeDate="2000-05-04T12:00:01" numShares="500" price="150.0" />
<cbret_006a ESP_OPS="i"  tradeId="2" symbol="EBAY"
tradeDate="2000-05-04T12:00:02" numShares="100" price="50.625" />
<cbret_006a ESP_OPS="i"  tradeId="3" symbol="EBAY"
tradeDate="2000-05-04T12:00:04" numShares="1000" price="16.875" />
```

esp_playback is run below without modifying the playrate and the output is written by executing **esp_subscribe** connected to the `cbret_006` window.

```
esp_playback -p remoteBox:19011/ws1/project1 -c sybase:sybase -C
espxml:sum/input.xml
esp_subscribe  -p remoteBox:19011/ws1/project1 -c sybase:sybase  -s
cbret_006:
```

The output is a single XML record with the correct share sum and average price of the three input records.

```
<cbret_006 ESP_OPS="i"  tradeId="3" symbol="EBAY"
tradeDate="2000-05-04 12:00:04" numShares="1600" price="72.500000"/
>
```

esp_playback is run below with the `-R` option changing the playrate to 1 record per 4000 milliseconds (4 seconds). The output is written to the standard output by executing **esp_subscribe** connected to the `cbret_006` window.

```
esp_playback -p remoteBox:19011/ws1/project1 -c sybase:sybase  -C
espxml:sum/input.xml -R 1:4000 -t1
esp_subscribe -p remoteBox:19011/ws1/project1 -c sybase:sybase  -s
cbret_006:
```

The output of this execution of **esp_playback** is 3 XML records identical to the input file.

```
<cbret_006 ESP_OPS="i"  tradeId="1" symbol="EBAY"
tradeDate="2000-05-04 12:00:01" numShares="500" price="150.000000"/>
<cbret_006 ESP_OPS="i"  tradeId="2" symbol="EBAY"
tradeDate="2000-05-04 12:00:02" numShares="100" price="50.625000"/>
<cbret_006 ESP_OPS="i"  tradeId="3" symbol="EBAY"
tradeDate="2000-05-04 12:00:04" numShares="1000" price="16.875000"/
>
```

Note that the server only keeps each symbol for a total of 1 second, as specified in the CCL code by the line

```
keep 1 seconds per (symbol)
```

As a result of slowing down the playrate to 1 record per 4 seconds, the share sum and average price were calculated incorrectly since the server only ever stored the share sum and average price of 1 record at a time.

Syntax

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

Required Arguments

- **`-c`** ***user[:password]* –** (required) authenticates access to an ESP server with a user ID and, optionally, a *password*. If you do not provide either the *password* or the **-k** or **-G** option, you are prompted for the password. If the Event Stream Processor successfully authenticates with these credentials, the connection is maintained, otherwise the Event Stream Processor immediately closes the connection.

Options

- **`-a`** **–** (optional) uses asynchronous publishing. In this mode, the publication does not wait for ESP Server to acknowledge the received data. The default is synchronous publishing.
- **`-p`** ***[<host>:]<port>/workspace-name/project-name* –** (required) together, the *host:<port>/<workspace name>/ <project name>* arguments specify the URI to connect to the ESP Server (cluster manager). For example, if you have started your ESP cluster server in the port **19011**, the host name is set as **localhost**, and you are running a project called

prj1 in the default workspace, specify `-p` as: `-p localhost:19011/default/prj1`

- **`-D` *user [:password]* –** (optional) authenticates access to a database with a userID and, optionally, a *password*. This option is only required for ODBC sources.
- **`-G` –** (optional) authenticates access to the Event Stream Processor with credentials held within a Kerberos authentication ticket. Environment variables determine where the system will look to find authentication tickets (See *Administrators Guide > Security in the Event Stream Processor > Authentication > Setting Environment Variables for Kerberos* for more information). You must use the **-c** option to specify a user name in order to locate the proper authentication ticket; if you do not use **-c**, the client prompts for a user name and password.
- **`-h` –** (optional) prints a list of possible options on the screen along with a brief explanation for each option.
- **`-i` –** (optional) turns on the shine-through flag. In this mode, if there are any missing columns in an update, they are filled in with the previous values for the updated row. The default behavior is to fill any missing columns with NULLS.
- **`-e` –** (optional) specifies that communications to ESP Server should be encrypted. Note that Event Stream Processor must be started in encrypted mode for this to work.
- **`-k` *privateRsaKeyFile* –** (optional) authentication is performed using the RSA private key file mechanism instead of password authentication. The **privateRsaKeyFile** must specify the pathname of the private RSA key file.

  Note: Ensure that the ESP Server has been started with the `-k` option specifying the directory in which to store the RSA keys.
- **`-L` –** (optional) resets the server time to match the time of the data source when it was being recorded. `-L` must be immediately followed by `-R`*columnName*, which sets the specified time at which playback should occur (according to the timestamp column).

  Note: If `-R` is not defined, the `-L` option does not function and reverts to use the current server time for playback.
- **`-C` *connStr* –** (dependent required) specifies the connection string for the source. The connection string varies from source to source.

  Supported sources formats include: ODBC, ESP XML, ESP DLM, Binary, Recorder, and KDB.

  The connection string for these are formats are as follows:

  - ODBC:`odbc:<dsnName>:<sql>|<file>:<fileName>|<query>`
    - **odbc –** the ODBC source string identifier. An ODBC 3.0 compliant driver for the required source must be installed and configured on the machine running this utility to use this source option.
    - ***dsnName* –** the ODBC data source name. For a Sybase IQ data source, the data source name resides in the `.odbc.ini` file (found in `<homedirectory>/.odbc.ini)`. For other database sources, the data

source names reside in the odbc.ini and odbcinst.ini files (Found in an ODBC folder such as /etc/odbc.ini and /etc/odbcinst.ini). The example below has a dsnName entry of "SybaseIQ".

```
[SybaseIQ]
         Description  = Sybase IQ database
         ServerName   = iqhost_iqdemo
         DatabaseName = iqdemo
         CommLinks    = tcpip{host=iqhost;port=2638}
         Driver       = /iqhome/IQ-15_1/lib64/libodbc.so
```

  - **file|sql –** specifies whether the following argument is a file name or a SQL query. One of these values is required.
  - ***fileName|query* –** the *fileName* is the name of the file containing a SQL statement. The query is the SQL query that needs to be executed to retrieve the data. One of these values is required.
- ESPXML: `espxml:<inputFile>`
  - **espxml –** is an identifier for an ESP XML file source.
  - ***inputFile* –** is the full path and name of the file containing data in ESP XML format.

  For information on how a record in an XML file should be configured, see *Publish and Subscribe Executables > esp_convert > Input Formats*.
- ESPDLM: `espdlm:<inputFile>[:<delimiter>]`
  - **espdlm –** is an identifier for ESP DLM file source.
  - ***inputFile* –** is the full path and name of the file containing data in ESP DLM format.
  - ***delimiter* –** this optional parameter is a single-character field delimiter; the default is a comma.

  For information on how a record in a DLM file should be configured, *Publish and Subscribe Executables > esp_convert > Input Formats*.
- Binary: `binary:<inputFile>`
  - **binary –** is an identifier for an ESP binary file source. The advantage to this format is that the loads are faster because there is no conversion required because the data is already in a format that the Event Stream Processor can absorb. The disadvantage to this format is that it is machine-architecture specific.

    One can use **esp_convert** to convert data from either XML or DML to binary format.

    Note: Binary files recorded in previous releases cannot be played back unless they are first converted to the new binary format using **esp_convert**. See *Publish and Subscribe Executables > esp_convert* for information on how to to convert binary files.
  - ***inputFile* –** is the full path and name of the file containing data in binary format.
- Recorder: `recorder:<inputFile>`

    - **recorder –** is an identifier for a file generated by the recorder. The recorder can be started using the ESP Studio or the through the recorder examples in `$ESP_HOME/client/pubsub/` folder.
    - ***inputFile* –** is the full path and name of the file containing data generated by the recorder.
  - KDB: `kdb:<host>:<port>:<sql>|<file>:<filename>|<query>:[<user>:<password>]`
    - **file|sql –** specifies whether the following argument is a file name or a SQL query. One of these values is required.
    - ***fileName|query* –** the *fileName* is the name of the file containing a SQL statement. The query is the SQL query that needs to be executed to retrieve the data. One of these values is required.

- **-R *playRate* –** specifies how fast/slow the data must be played back. If this parameter is not supplied, the record plays as fast as possible. The following examples show the two ways that the playback rate can be supplied.

  ```
  records:milliseconds
  ```

  where:

  - **records –** is the number of records to publish in the given number of milliseconds. This value can be 0 only if the *millisecond* component is also 0. A value of 0 indicates play as fast as possible.
  - **milliseconds –** is the number of milliseconds to play back the given number of records.

    This property is not supported for the source type recorder.

  ```
  columnName
  ```

  where:

  - **columnName –** is the column name in the target stream that controls how fast the record is played back. The column name is case-sensitive and an error is reported if the provided column name does not exist in the target stream.

    The **columnName** property is ignored for the source type Recorder and is currently not supported for binary file sources.

- **-T *timeScaleRate* –** specifies a multiplication factor for the delta between the times for two consecutive records. This is used with the playback rate, when the playback rate is controlled by a column in the source. It takes an `integer` value between -N to +M. A positive value greater than +1 speeds up the time and a negative value less than -1 slows down the rate. A value of +1 or -1 plays back at the rate specified in the column and a value of 0 specifies that the column values are ignored. Default value is 1.

- **-r *interval* –** specifies the minimum number of seconds to wait between each reporting of the published statistics. The default is 5 seconds. A value of 0 indicates no reporting. The time interval check is triggered when there is a record to be published. This means that even when the reporting interval has been exceeded the statistics are reported if there are no records to trigger the check.
- **-B *bufferSize* –** specifies the internal read and write buffer sizes. The default buffer size is 32K, which is also the maximum allowed value. Smaller buffer sizes use less memory but may run slower in some situations.
- **-t *size* –** specifies that transaction blocks must be used to publish data to ESP Server, with each block *size* large. ESP Server processes data faster if transaction blocks deliver the data. The performance boost is achieved in two ways: the network is used more efficiently because a larger number of records are packed into a single network packet; and because ESP Server treats the records as a single block, which fails or passes in its entirety. This results in less processing overhead. Depending on the nature of the application, this option may not be suitable.
- **-w *size* –** specifies that envelopes must be used to publish data to the Event Stream Processor, with each envelope of size *size*. If neither the *-t* nor *-w* option is specified, the default value is -w64. The value of *size* must be between 1 and 1024. Using this option ensures network efficiency in delivering data to ESP Server by modifying it to treat the records as individual records.
- **-s *streamName* –** (required) specifies the target stream name for this utility. This option is required for ODBC and KDB sources. All other sources have the target stream-name/ stream-ID embedded in them.

  Note: XML and delimited sources may require configuration prior to use. See *Server Executables > esp_playback > Input Formats* for information on how to configure XML and delimited sources.

  The *streamName* variable is case-sensitive.
- **-S *maxStringSize* –** specifies the maximum string size that can be processed. This option is used in ODBC sources. The default value is 1024.

  This value is global for all strings in the source. Specifying a large value for this option may result in increased memory usage depending on the number of strings in the record and the value specified with the `-B` option.
- **-m *dateMask* –** the date mask to use for XML and delimited files. The default is "%Y-%m-%dT%H:%M:%S'. The date mask is common for all date columns. This option is meaningful only for the ESP XML and delimited file sources.

Examples

- **Reading and playing back data –** To read data from an ODBC source with a DSN name of 'myODBC_connection' into a stream called 'InputWindow1', using a command line SQL statement:

```
esp_playback -D dbuser:dbpassword -p myserver:9786/default/
```

```
odbc_playback_test -C ODBC:myODBC_connection:SQL:"select * from
mytable" -s InputWindow1
```

To read data from an ESP XML source `trades.xml`, play the data at a rate of 10,000 rows per second, and report the progress every 15 seconds:

```
esp_playback -c user:pass -p localhost:19022/w1/p1 -C "espxml:/
tmp/trades.xml" -R 10000:1000 -r 15
```

To read data from an ESP DLM source `trades.csv`, play the data at a rate of 10,000 rows per second, and report the progress ever 15 seconds.

```
$ESP_HOME/bin/esp_playback -c user:pass -p localhost:19011/w1/p1
-C "espdlm:qa/trades.csv:," -R 10000:1000 -r 15
```

CHAPTER 3 Command and Control Executables

Use these command-line utilities to connect to and issue commands to Event Stream Processor.

Each Event Stream Processor instance has a single Command and Control server thread that handles requests for information (metadata) or instructions to perform tasks such as **quiesce** or **shutdown**.

The **esp_cnc** and **esp_client** utilities send directives to Event Stream Processor via this process. These utilities enable you to control the running Event Stream Processor from the command-line.

The **esp_query** On-Demand SQL Query utility uses this process to send requests for information to Event Stream Processor. See the *On-Demand Queries* topic for more details.

esp_cnc

Connects to the Event Stream Processor via the Command and Control and Gateway interfaces, and issues simple command and control commands to the server. It prints the results on the standard output.

Syntax

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

Required Arguments

- **`-C`** ***command*** **–** (required) may be one of these literals: **getGateway**, **getBaseStreams**, **getDerivedStreams**, **getStreamDefinition**, **getAddressSize**, **getDateSize**, **sendStreamsExit**, **augmentSubscriber**, **removeSubscriber**, **isBigEndian**, **isQuiesced**, **isQuiescedNow**, **returnWhenQuiesced**, **setParam**, **getVersion**, **getStreamHandle**
- **`-c`** ***user[:password]*** **–** (required) authenticates with a user ID and, optionally, a *password*. If you do not provide either the *password* or the **-k** or **-G** option, the user is prompted for the password. If the Event Stream Processor successfully authenticates with these credentials, the connection is maintained, otherwise the Event Stream Processor immediately closes the connection.
- **`-p`** ***[<host>:]<port>/workspace-name/project-name*** **–** (required) together, the *host:<port>/<workspace name>/<project name>* arguments specify the URI to connect to the ESP Server (cluster manager). For example, if you have started your ESP cluster server in the port **19011**, the host name is set as **localhost**, and you are running a project called

prj1 in the default workspace, specify `-p` as: `-p localhost:19011/default/prj1`

Options

- **`-e` –** (dependent required) encrypts traffic via openSSL. When this option is not present, no encryption occurs.
- **`-G` –** (optional) authenticates access to the Event Stream Processor with credentials held within a Kerberos authentication ticket. Environment variables determine where the system will look to find authentication tickets (See *Administrators Guide > Security in the Event Stream Processor > Authentication > Setting Environment Variables for Kerberos* for more information). If the user name differs from the default principal name in the ticket cache, specify an alternate user name with the **–c** option to use the corresponding authentication ticket.
- **`-h` –** (optional) a list of possible options on the screen along with a brief explanation for each option.
- **`-H` *handle* –** (dependent required) specifies the client handle to direct an **augmentSubscriber** or **removeSubscriber** command. This option is required for those commands.
- **`-k` *privateRsaKeyFile* –** (optional) performs authentication using the RSA private key file mechanism instead of password authentication. The *privateRsaKeyFile* must specify the pathname of the private RSA key file. This option is required if the ESP Server was started with the `-V rsa` option. With this option enabled, the user name must be specified with the `-c` option, but the password is not required.

  Note: Ensure that the ESP Server has been started with the `-k` option specifying the directory in which to store the RSA keys.
- **`-P` *name:value* –** (dependent required) specifies the new value to be associated with a variable. Required for the **setParam** option. You cannot set the value of a parameter in a running project; use the `-P` option to set new values for variables only.
- **`-s` *stream* –** (dependent required) specifies a single stream. Required for the commands **getStreamDefinition**, **augmentSubscriber**, **getStreamHandle** and **removeSubscriber**.

esp_client

Controls and gets information from a running Event Stream Processor instance.

Syntax

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

Required Arguments

- **`-c` *user[:password]* –** authenticates with a *user* ID and, optionally, a *password*. If you do not provide either the *password* or the **-k** or **-G** option, you are prompted for the password. If the Event Stream Processor successfully authenticates with these credentials, the connection is maintained, otherwise the Event Stream Processor immediately closes the connection.

Options

- **`-G` –** (optional) authenticates access to the Event Stream Processor with credentials held within a Kerberos authentication ticket. Environment variables determine where the system will look to find authentication tickets (See *Administrators Guide > Security in the Event Stream Processor > Authentication > Setting Environment Variables for Kerberos* for more information). If the user name differs from the default principal name in the ticket cache, specify an alternate user name with the **`–c`** option to use the corresponding authentication ticket.
- **`-h` –** (optional) prints a list of possible options on the screen along with a brief explanation for each option.
- **`-i` *file* –** (optional) runs the commands in the given file.
- **`-k` *privateRsaKeyFile* –** (optional) performs authentication using the RSA private key file mechanism instead of password authentication. Ensure that the *privateRsaKeyFile* pathname of the private RSA key file is specified. This option is required if the Event Stream Processor was started with the `-V` RSA option. With this option enabled, the user name must be specified with the `-c` option, but the password is not required.

  Note: Ensure that the ESP Server has been started with the `-k` option specifying the directory in which to store the RSA keys.

- **`-p`*[<host>:]<port>/workspace-name/project-name* –** (required) together, the *host:<port>/<workspace name>/<project name>* arguments specify the URI to connect to the ESP Server (cluster manager). For example, if you have started your ESP cluster server in the port **19011**, the host name is set as **localhost**, and you are running a project called **prj1** in the default workspace, specify `-p` as: `-p localhost:19011/default/prj1`
- **`-q` –** (optional) disables the prompt. This option is useful when reading commands from a file piped to standard input on Windows.
- **`-x` –** (optional) enables echoing of commands before execution. This can also be changed later with the command **echo on|off**.
- **`-v` –** (optional) prints the **esp_client** utility version.

Commands

Enclose arbitrary values that a command takes as parameters in single back quotes(`` ` ``) for Windows and curly braces `{}` for Linux or Solaris. For example, `` `parameter` `` or `{parameter}`. Separate commands with a semicolon.

Note: Back quotes (`) are used to avoid confusion with the normal single (') and double quotes (") that are used in the syntax of the shell and SPLASH expressions.

Strings enclosed in back quotes cannot use back quotes within the string itself. You cannot use double quotes on **esp_client** commands on the command line, because the UNIX shell attempts to interpret them. To prevent this, shield back quotes with curly braces ({}). For example:

```
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}"
```

The strings enclosed in curly braces ({}) must have a balanced number of braces inside of them. For example:

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

You cannot use back ticks and curly braces for multiline values. There is another quoting style that is intended for the large multiline parameters, such as the configuration files. These parameters start with `<<!`, then the inline text of the parameter, and finally a line containing only `!` (with no whitespace before or after). This syntax is similar to the shell's "<<" syntax but the terminating word "`!`" cannot be changed. The line breaks after the "`<<!`" and before the terminating "`!`" are not part of the parameter, so you can specify single-line values through the syntax as well. You can specify multiple inline parameters by separating them with lines containing `!<<!`. For example:

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

Note: This syntax can be used with any command.

Many commands can redirect output to a file using:

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

The ">" operator overwrites an existing file, and ">>" appends to an existing file. The operator "`|`" pipes the output to a UNIX command pipeline. Use either back quotes or braces to quote the file name or the filter program.

General commands are:

- **addrsize –** prints the address or pointer size (in bytes) of the connected Event Stream Processor instance (4 bytes for 32-bit addressing, 8 bytes for 64-bit addressing). This value reflects how the connected platform instance is compiled (32-bit vs. 64-bit). For example, a 32-bit platform can be running on a 64-bit host, in which case the **addrsize** command returns a value of 4, not 8.
- **backup –** creates a backup of all the log stores. The backup files are created with the suffix ".bak", so, for example, `dynamic.log` is backed up into the file `dynamic.bak`. The backup files are created with compacted contents.
- **clear base stream `*stream*` –** deletes the contents of the specified base stream.
- **clock –** prints the current state of the Event Stream Processor logical clock. For example:

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

  The time is printed as both the number of seconds since the UNIX epoch and a user-readable value. Rate is the clock rate relative to real time: 10 means "10 times faster", 0.1 means "10 times slower". The real flag shows whether the clock matches the system time of the machine where the Event Stream Processor runs (1), or if the clock has been changed artificially (0). Stop depth shows how many times the clock has been stopped recursively (how many times start clock would have to be called to actually resume the flow of time). When the clock is running, the stop depth is 0. Max sleep is the period of time, in real milliseconds, that guarantees that all the sleepers discover the changes in the clock rate or time. Calls that change the clock rate sleep automatically (with the logical clock stopped) to ensure that their effects have been applied cleanly.
- **clock [rate `*number*`] –** changes the rate of the platform clock. Rate is specified as a floating-point number, the minimum rate is 0.001.
- **clock [time `*number*`] –** changes the current time of the platform clock. Time can be specified as a floating point number of seconds since the UNIX epoch or in the format year-month-day Thour:min:sec. The same can be done with milliseconds: year-month-dayThour:min:sec.NNN. The letter "T" is literal, as in the default Event Stream Processor time format. If the time value is prefixed with **add**, it specifies a change to the current time. In this case it must be a floating point number of seconds. Prints the previous state of the clock as it was before executing the command. See the description in the **clock** command.
- **clock [rate `*number*`] [time [add] `*number*`] –** changes the current time and/or rate of the logical clock. Rate is specified as a floating-point number, the minimum rate is 0.001. Time can be specified as a floating point number of seconds since the UNIX epoch or in the format year-month-day Thour:min:sec. The same can be done with milliseconds: year-month-dayThour:min:sec.NNN. The letter "T" is literal, as in the default Event Stream Processor time format. If the time value is prefixed with **add**, it specifies a change—floating point number of seconds—to the current time. Prints the state of the clock as it was before executing the command. See the description in the **clock** command.
- **clock real –** returns the logical clock of the Event Stream Processor to the real time (use the system clock of the machine where the Event Stream Processor is running). You cannot set

the clock to real time while it is stopped. Prints the previous state of the clock as it was before executing the command. See the description in the **clock** command.

- **clock stop on pause *[`0|1`]* –** shows or changes the flag that controls whether the Event Stream Processor logical clock stops when the Event Stream Processor is paused in trace mode. Prints the state of the clock as it was before executing the command. See the description in the **clock** command.
- **datesize –** prints the size of a date field (in bytes) of the connected Event Stream Processor instance (8 bytes for instances using the Win32 filetime, 4 bytes for instances using the 32-bit integer time_t filetime).
- **echo *`string`* –** prints the string to standard output.
- **echo –** enables or disables the printing of commands to standard output before execution.
- **endian –** prints the endian value ("big" or "little") of the machine on which the connected Event Stream Processor instance is running.
- **fd –** displays the field delimiter value.
- **fd *delimiter* –** sets a new field delimiter value. Do not quote delimiter. Any non-space character is taken as the new delimiter. The default delimiter is the pipe (|) character.
- **gateway –** prints the host and port number of the Gateway interface.
- **get_config –** gets the current running XML configuration and prints it to standard output or to a local file by specifying *>file* after the command.
- **help –** prints the general help message.
- **help flags –** prints help for the set of output control flags.
- **history *`number` [`stream`]* –** changes the maximum history size of a stream. The history is collected only with trace mode on. Every time the trace mode is turned off, the history is discarded. As the history collects, only the last number of input and output transaction pairs are kept, the older ones are discarded. The default limit on the Event Stream Processor is 100.
- **history ex *`stream`* –** displays the current maximum history size of a stream.
- **idx *`streamName`* –** outputs the index for the stream specified.
- **immediate stop –** stops the Event Stream Processor immediately, without shutting down any streams.
- **immediate pause –** forces the Event Stream Processor into a paused state. This is a last-resort way to examine the Event Stream Processor if it is internally deadlocked or frozen. This command corrupts the Event Stream Processor state; use it only when the processor is indefinitely stuck.
- **kill *`handle`* –** kills the client connection with the specified handle. The metadata stream _ESP_Clients contains a list of all open connections.
- **kill every *`name`* –** kills all the client connections with the specified tag name. The metadata stream _ESP_Clients contains a list of all open connections and their tag names. You can specify tag name using option `-m` of such commands as **esp_subscribe** and **esp_upload**.
- **lock timeout *[`seconds`]* –** shows or changes the value of exclusive lock timeout of the Event Stream Processor. Most commands (except those that wait for certain events) are

serialized using the lock, with one command executing at a time. The timeout prevents the subsequent commands from hanging forever waiting for this lock if an error occurs. The default timeout is 60 seconds.

- **loglevel `*level*` –** sets the logging level of the Event Stream Processor. For more information, see the *Administrators Guide*.

- **moneyprecision –** prints the precision of the 'money' datatype (in digits to the right of the decimal).
- **putd `*delimited record*` –** puts a single delimited record (within back quotes or braces) to the Gateway I/O process. If you do not specify a delimiter using the **fd** *delimiter* command, the default delimiter, |, is assumed. For example, to enter a comma-delimited list containing an input window name followed by three fields, enter:

  ```
  esp_client> fd,
  esp_client> putd `ExampleInputWindow,i,1,abc`
  ```

  Without the initial **fd** command, the correct syntax would be:

  ```
  esp_client> putd `ExampleInputWindow|i|1|abc`
  ```

  See *Put Command Notes* for additional information.
- **putx `*XML record*` –** puts a single XML formatted record (within back quotes or braces) to the Gateway interface. See *Put Command Notes*.
- **quiesced –** prints the quiesced state (1 for true or 0 for false). The state is 1 if there are no publisher connections and all input data has fully propagated through the model.
- **quit –** exits the **esp_client** utility.
- **refresh_calendars –** refreshes calendar data from files; this command does nothing if the Event Stream Processor has not loaded any calendars via its calendar functions.
- **save_config ` *remoteFile*` –** saves the current running XML configuration to this file on the ESP Server. The file cannot yet exist, and you must quote the file name.
- **setparam ` *variable*` `*value*` –** sets the specified variable to a specified value. You cannot set the value of a parameter in a running project.
- **settings –** prints the field separator, flag values, and so forth.
- **snapshot *[`streamName`]* –** prints the current content of the stream in tabular form, using the output control flag settings. See *Put Command Notes*.
- **start adapters initial –** starts all the adapters as specified in the Event Stream Processor **ADAPTER START** statement. Any adapters that are already running continue to run, and any adapters that are not currently running are restarted. This command waits for all the started adapters to complete their initial loading.
- **start clock –** resumes the logical clock of the Event Stream Processor. Prints the state of the clock as it was before executing the command. See the description in the **clock** command.
- **start adapter `*adapter-or-group*` –** starts a named adapter, or all adapters from a named group, from the **ADAPTER START** statement. For the adapters that are already running, this command has no effect. This command does not wait for the full start of the adapters,

but returns immediately. Use **wait connector initial** to wait for completion of initial loading.

- **stop –** issues **exit streams** command to the Event Stream Processor Command and Control interface, causing the Event Stream Processor engine to exit.
- **stop clock –** stops the logical clock of the Event Stream Processor. Records continue processing, but timestamps do not change and timer events do not occur. While the clock is stopped, you can change the time and rate but cannot change the clock to real time. You may call **stop clock** multiple times; however, you must also then call **start clock** an equal number of times to resume flow. The Event Stream Processor may internally stop and resume the clock; however, you should resume only if they initiated the stop. Prints the state of the clock as it was before executing the command. See the description in the **clock** command.
- **stop adapter *`adapter-or-group`* –** stops a named adapter, or all adapters from a named group, from the **ADAPTER START** statement. For the adapters that are not running, this command has no effect. The output adapters are allowed to complete the processing of their output queue before they stop (however, no new commands are added to the queue). This command does not wait for the adapters to be stopped, but returns immediately. Use **wait connector** to wait for completion of the adapters.
- **stop adapter immediate *`adapter-or-group`* –** stops a named adapter, or all adapters from a named group, from the **ADAPTER START** statement. Similar to **stop adapter** but the output queue from output adapters is discarded and the adapters are requested to stop immediately. For input adapters, **stop adapter** and **stop adapter immediate** are equivalent because these adapters have no output queue. This command does not wait for the adapters to stop. Use **wait adapter** to wait for completion of the adapters.
- **stream *`streamName`* –** prints the definition of the specified stream, using the **hdr** and **sphdr** output control flags. See *Output Control Flags*.
- **streams –** prints the list of base and derived streams.
- **throttle *`number` [`stream`]* –** changes the input queue throttle value for one or all streams. Any writes to the stream's input queue are blocked when the queue size reaches double the throttle value. The throttle value cannot be increased beyond the default value (it can only be reduced). Reducing this value may be useful for tracing records during debugging.
- **throttle ex *`stream`* –** displays the current throttle value of a stream.
- **trace_mode *[on|off]* –** changes or gets the current state of the trace mode. Without any argument, this command prints the current state; *on* enables the trace mode and *off* disables it. The trace mode is a prerequisite for the commands that relate to single-stepping, breakpoints, and examining debugging information.
- **wait adapter *`adapter-or-group`* –** waits for a named adapter, or for all adapters from a named group, from the **ADAPTER START** statement to terminate. They may terminate naturally (running out of data in the data source) or as a result of **stop adapter**.
- **wait adapter initial *`adapter-or-group`* –** waits for a named adapter, or for all adapters from a named group, from the **ADAPTER START** statement to complete the initial loading. This command waits for the adapter state to change to something other than "initial".

- **wait quiesced –** waits until all input fully propagates through the model. Input received after this command is buffered until the propagation of the previous data completes. Then the Event Stream Processor resumes normal operation.
- **wait quiesced gateway –** waits until all publishing clients disconnect and all input fully propagates through the model. This command waits for the condition when the command **quiesced** returns 1. If any new clients connect while this command is waiting for the data to propagate, the data from these clients is buffered until the wait completes.

Commands Requiring Trace Mode

- **pause –** pauses the Event Stream Processor execution. Returns after the pause begins. All data examination and single-stepping commands require the Event Stream Processor to be paused first, explicitly with this command, or on a breakpoint or exception on bad data. If the Event Stream Processor is already paused, this command returns success immediately.
- **check_pause –** shows whether or not the Event Stream Processor is currently paused.
- **wait_pause –** waits until the Event Stream Processor is paused by a breakpoint or from another instance of **esp_client**. The wait cannot be interrupted (other than by ending **esp_client**).
- **run –** continues the normal Event Stream Processor execution.
- **step *[`stream`]* –** executes a single step when the Event Stream Processor is paused. If a stream name is given as an argument, a single step is executed on this stream. Otherwise, a stream to be stepped through is picked at random among the streams ready to process data. If no streams are ready to process (all are waiting for input or output), this command is not performed and returns success immediately.
- **step timeout *[`number`]* –** sets the timeout in milliseconds for the automatic stepping. The default timeout is 0.3s. Using a negative or zero value resets the timeout to default.
- **step trans *`stream` [`limit`]* –** automatically steps the stream at least once, and then to just before the end of transaction (the "PUT" location on the stream state diagram). Specify the second argument to limit the number of steps to be made, to limit the running time of large transactions. The default limit is 10000. If the stream has no input pending, or if it blocks on the output for more than timeout, the stepping also stops and returns an error.
- **step quiesce stream *`stream` [`limit`]* –** automatically steps the stream and all its descendants until all their input queues are empty. The first stream in this command is a literal and the second one represents a parameter—the stream name. Specify the second argument to limit the number of steps to be made, to limit the running time of large amounts of data collected in the queues. The default limit is 100000. If the stream is a base stream, and the input on it comes quickly, the call returns only when the limit of steps is achieved. This is not an issue for derived streams; since the Event Stream Processor is paused when stepping, any inputs to this stream are also paused. If a derived stream's input queue is full, and inputs are waiting to deposit their already processed data on it, the waiting inputs add their transactions as the input queue is processed. If none of the streams has input pending, or if they all block on the output longer than the timeout, stepping also stops and returns an error.

- **step quiesce downstream *`stream` [`limit`]* –** similar to **step quiesce stream**, except the stream itself is not stepped through; only its descendants are. This command is useful for clearing out descendant streams' input queues. When the argument stream produces its output, the data progression through the descendant streams can be easily traced.
- **step quiesce from base *[`limit`]* –** automatically steps all derived (non-base) streams until their input queues are empty. Specify this argument to limit the number of steps to be made, to limit the running time for large amounts of data in the queues. The default limit is 100000. If none of the streams has input pending, or if all of them block on the output for more than the timeout value, the stepping also stops and returns an error. This command may be useful for cleaning out the queues of derived streams before processing an inconsistent record through the base stream. You can easily watch the progression of data through the derived streams.
- **dump *`filePrefix` [`stream`]* –** dumps the contents of each stream, or one specific stream, to a file. Each file is named `filePrefixdump_streamName.xml`
- **bp add *`stream` `inputStream` [`condition`]* –** adds a breakpoint on a stream, before it begins processing an input record from another stream (*inputStream*). Optionally, use a SPLASH expression to specify a condition to trigger the breakpoint. The breakpoint is only triggered when the condition evaluated on the input record is true. The expression may refer to either or both of these predefined variables:

  - **currow –** the current input record.
  - **oldrow –** the previous value of the record with this key, that is being updated or deleted.

  The condition may refer to the fields in the records, *"row.field"*. The stream's local and global variables may be used as well. This command prints the ID of the newly created breakpoint.
- **bp add *`stream` any* –** adds a breakpoint on a stream, before it starts processing an input record from any stream. You cannot specify the condition.

  This command prints the ID of the newly created breakpoint.
- **bp add *`stream` out [`condition`]* –** adds a breakpoint on a stream, after it has processed an input record and produced some (possibly empty) output. Optionally, use a SPLASH expression to specify a condition to trigger the breakpoint. The breakpoint is only triggered when the condition evaluated on the input record is true. The expression may refer to one predefined variable, *currow*, which is the current output record. Since one input record may produce multiple output records, the condition is evaluated for each of them sequentially. If there is no output produced, the condition still evaluates once with *currow* set to NULL. The condition may refer to the fields in the records, *"row.field"*.

  This command prints the ID of the newly created breakpoint.
- **bp del *`id`* –** deletes the breakpoint with the specified ID (as returned from **bp add** or reported by **bp list)**.
- **bp del all –** deletes all the breakpoints.
- **bp on|off *`id`* –** enables or disables the breakpoint with specified ID.

- **bp on|off all –** enables or disable all the breakpoints.
- **bp every *`count` `id`* –** makes the breakpoint with a specified ID trigger on every nth occasion. For example, to make the breakpoint with ID 8 trigger on every 100th record, use "**bp every** `100` `8`". Setting the count to 1 triggers the breakpoint one every record.
- **bp every *`count` all* –** triggers all breakpoints on every Nth occasion.
- **bp list –** lists the breakpoints. an alternative for **"ex `breakpoints`"**.
- **ex *`kind` [`stream` [`object`]]* –** examines data in the Event Stream Processor. **ex** takes the name of the kind of data, of the stream to which it belongs, and of the particular object. For some kinds of data, the stream and object arguments may not be applicable. The data is printed in XML format, with the element name for most data kinds set to "row". If the data represents a transaction, it is enclosed in a *<trans>* element. If the data represents an update pair, it is enclosed in a *<pair>* element. The exact fields depend on the data being examined.

  When you examine the input data kinds (input queue, current input transactions and row, input history), the data may be a mix of rows of different types, produced by different streams. The name of the XML element is set to the name of the stream that produced it (for base streams, this is the name of the base stream itself).

  Kinds of data currently supported are:

  - **`pause` –** state of the user streams when paused. The fields are:
    - **name –** name of the stream.
    - **loc –** location where the stream is paused.
    - **onbp –** if on a triggered breakpoint, the ID of that breakpoint. Otherwise, **onbp** is set to 0. If multiple breakpoints are triggered simultaneously, **onbp** contains the ID of one of them.
    - **throttle –** the input queue throttle value.
    - **history –** maximum size of the kept history.
    - **postSeq –** count of transactions posted to the input queue.
    - **inSeq –** count of transactions ever read from the input queue.
    - **outSeq –** count of transactions processed to the output (including the empty transactions that get discarded, and the expiry transactions).
    - **stepSeq –** the count of steps (as defined by the **step** command) made in trace mode. This includes both single-stepping and running. Use the changes in this count to determine which streams have changed state.
  - **`pauseall` –** same as **pause** but includes the metadata streams as well.
  - **`breakpoints` –** information about all currently registered breakpoints. The fields are:
    - **id –** ID of the breakpoint. Does not change throughout the breakpoint's life.
    - **stream –** name of the stream on which the breakpoint is defined.

  - **origin –** contains the name of the input stream for a breakpoint on a particular input stream. Use "*" for a breakpoint on input from any stream, and "" (empty) for a breakpoint on output.
  - **expr –** conditional expression.
  - **enabledEvery –** n to trigger the breakpoint on every nth matching record. See **bp every**.
  - **leftToTrigger –** the number of matches that are currently left for the breakpoint until triggering.
  - **onit –** set to 1 if the breakpoint is currently triggered, otherwise set to 0.
- **`var``` `*var-name*` –** contents of a global variable (one defined in the global **DECLARE** block). The fields depend on the type of variable. The indexes in the arrays are shown as **ESP_Index**. The keys in the dictionaries are shown as **ESP_Key_<field-name>**. The values of records are shown with fields as in the record definition. The simple variables are represented with the **ESP_Value** field. For structured values, this command may return multiple rows. If a variable is NULL, nothing is returned. For an array, only the elements with non-NULL values are shown.
- **`listVar` –** the list of all global variable names.
  - **name –** name of the variable.
  - **type –** type of the variable.
- **`store``*stream*` –** contents of a stream's store. The fields are as in the stream's row definition.
- **`outTrans` `*stream*` –** the current output transaction as it is being built. The fields are the same as in the stream's row definition.
- **`outRow``*stream*` –** output produced from processing the previous input row. May contain multiple or no rows. The fields are the same as in the stream's row definition.
- **`badRows``*stream*` –** when the Event Stream Processor is paused on a bad rows exception, contains these bad rows. The fields are the same as in the row definition of the stream that produced the data (or, for a base stream, of the current stream).
- **`badRowsReason``*stream*` –** for each bad row reported by **badRows**, this data contains an error message explaining why it is bad. The message is in the reason field.
- **`outHist``*stream*` –** the output transactions from the stream's history. The empty transactions are returned as records with all fields containing NULL. There is one-to-one match between the transactions returned by **ex `outHist`** and **ex `inHist`**. The fields are the same as in the stream's row definition.
- **`lastOutTrans``*stream*` –** the newest output transaction in the stream's history. Similar to **" ex `outHistLatest` `stream` `0`"**, but if the history is empty, returns a success with no rows, while **outHistLatest** returns an error. The fields are the same as in the stream's row definition.
- **`outHistEarliest``*stream*` `*index*` –** selects an individual output transaction from the stream's history. The index is a number, 0 selecting the earliest transaction saved in the history, and increasing index numbers indicate later transactions. If there is no

transaction with such an index, returns the "No such object" error. The fields are the same as in the stream's row definition.

- **`outHistLatest``*stream*` `*index*` –** select an individual output transaction from the stream's history. The index is a number, 0 selecting the latest transaction saved in the history, and increasing index numbers indicate earlier transactions. If there is no transaction with such an index, returns the "No such object" error. The fields are the same as in the stream's row definition.
- **`var` `*stream*` `*var-name*` –** contents of a stream's local variables. You can examine only the local variables (those defined in the stream's local **DECLARE** block). Local variables include the variables of array, dictionary, and eventCache. The variables defined inside the SPLASH blocks exist only when the appropriate methods run, and cannot be examined. The fields depend on the type of variable. The indexes in the arrays and eventCaches are shown as **ESP_Index**. The keys in the dictionaries and eventCaches are shown as **ESP_Key_<field-name>**. The values of records are shown with fields as in the record definition. The simple variables are represented with the **ESP_Value** field. For structured values, **var** may return multiple rows. If a variable is NULL, nothing is returned. For an array only the elements with non- NULL values are shown. You cannot access global variables this way; instead, use the empty stream name to access them.
- **`listVar` `*stream*` –** the list of all variable names defined on this stream. Applicable only to the streams that are allowed to have the local **DECLARE** block. Does not include the global variables.
  - **name –** name of the variable.
  - **type –** type of the variable.
- **`queue` `*stream*` –** an input data kind. Contents of the stream's input queue. The fields are the same as in the row definition of the stream that produced the data (or, for a base stream, of the current stream).
- **`inTrans` `*stream*` –** an input data kind. The current input transaction that is being processed. The fields are the same as in the row definition of the stream that produced the data (or, for a base stream, of the current stream).
- **`inRow` `*stream*` –** an input data kind. The current input row that is being processed. The fields are the same as in the row definition of the stream that produced the data (or, for a base stream, of the current stream).
- **`queueHead` `*stream*` `*index*` –** an input data kind. Select an individual transaction from the stream's input queue. The index is a number, 0 selecting the transaction at the head of the queue, increasing index numbers indicating following transactions. If there is no transaction with such an index, returns the "No such object" error. The fields are the same as in the row definition of the stream that produced the data (or, for a base stream, of the current stream).
- **`queueTail` `*stream*``*index*` –** an input data kind. Select an individual transaction from the stream's input queue. The index is a number, 0 selecting the last transaction at the tail of the queue, increasing index numbers indicating previous transactions. If there is no transaction with such an index, returns the "No such object" error. The fields are the

same as in the row definition of the stream that produced the data (or, for a base stream, of the current stream).

- **`inHist` `*stream*` –** an input data kind. The input transaction from the stream's history. There is a one-to-one match between the transactions returned by **ex `outHist`** and **ex `inHist`**. The fields are as in the row definition of the stream that produced the data (or, for a base stream, of the current stream).
- **`lastInTrans` `*stream*` –** an input data kind. The newest input transaction in the stream's history. Similar to **"`inHistLatest` `stream` `0`"**, but if the history is empty, returns a success with no rows, while **`inHistLatest** returns an error. The fields are the same as in the row definition of the stream that produced the data (or, for a base stream, of the current stream).
- **`inHistEarliest` `*stream*` `*index*` –** an input data kind. Select an individual input transaction from the stream's history. The index is a number, 0 selecting the earliest transaction saved in the history, increasing index numbers indicating later transactions. If there is no transaction with such an index, returns the "No such object" error. The fields are as in the row definition of the stream that produced the data (or, for a base stream, of the current stream).
- **`inHistLatest` `*stream*` `*index*` –** an input data kind. Select an individual input transaction from the stream's history. The index is a number, 0 selecting the latest transaction saved in the history, increasing index numbers indicating earlier transactions. If there is no transaction with such an index, returns the "No such object" error. The fields are the same as in the row definition of the stream that produced the data (or, for a base stream, of the current stream).
- **`hist` `*stream*` –** a mixed representation of history, including both input and output data. Each input transaction is followed by its matching output transaction. The rows in the input transactions are marked with the XML tag of their origin stream name, while the rows in the output transaction are marked with the XML tag "row".
- **`lastTrans` `*stream*` –** a mixed input-and-output data kind. See **`hist`**. The newest input and output transactions in the stream's history.
- **`histEarliest` `*stream*` `*index*` –** a mixed input-and-output data kind. See **`hist`**. Select an individual transaction pair from the stream's history. The index is a number, 0 selecting the earliest transaction saved in the history, increasing index numbers indicating later transactions. If there is no transaction with such an index, returns the "No such object" error.
- **`histLatest` `*stream*` `*index*` –** a mixed input-and-output data kind. See **`hist`**. Select an individual transaction pair from the stream's history. The index is a number, 0 selecting the latest transaction saved in the history, increasing index numbers indicating earlier transactions. If there is no transaction with such an index, returns the "No such object" error.
- **`aggrGroup` `*aggregationStream*` –** the internal state of an aggregation stream, from its group index. Works only for aggregation streams that are not optimized to use additive aggregations. The value fields have names from the input stream row definition. The key fields have the same name as in this stream's row definition but with

ESP_Key_ prefixed to them. The index of the record in the aggregation bucket is in the **ESP_Index**field.

- **`states` *`patternStream` [`patternNum`]* –** states of the automatons in a pattern stream. Initially, a pattern stream has one automaton per defined pattern. As data is received and matched by patterns, a new automaton is cloned for each sequence of events that may match the pattern. As complete patterns are found, or the sequences of events are found to not match the patterns, automatons are destroyed.

  If the optional parameter *patternNum* is present, **states** shows only the automatons for that pattern. The fields are:

  - **pnum –** number, starting with 0, of the pattern being parsed by this automaton.
  - **instance –** instance number of the automaton. As new automatons are cloned, each receives a unique instance number. The instance number is never repeated (unless the Event Stream Processor is restarted). The pair (pnum, instance) identifies an automaton during its execution.
  - **state –** a number identifying the current state of the automaton. Automatons are linear; they have no loops in their logic, and may visit a state only once. If the state has not changed since last examination, it indicates that the automaton has not matched any new data. The numbers used for states are not sequential.
  - **timed –** if set to 1, an expiration timer attaches to this automaton. If the timer expires, its pattern match is considered failed and the automaton is destroyed. If set to 0, the automaton does not expire. The untimed automaton is the very initial automaton of a pattern, used to clone all others.
  - **time_left –** the time, in seconds, until expiration for a timed automaton. For an untimed automaton, this is always 0. By default, when the Event Stream Processor is paused, the platform logical clock stops. However, if the clock is set to not stop on pause, the timers do not stop. A value of 0 or negative means that the automaton expires when the Event Stream Processor execution resumes.
- **`bindings` *`patternStream` [`patternNum`]* –** the pattern variable bindings that are caused by the data parsed so far, for each automaton in *patternStream*. If the optional parameter *patternNum* is present, shows only the data for that pattern. The fields are:

  - **pnum –** number, starting from 0, of the pattern being parsed by this automaton.
  - **instance –** instance number of the automaton. As new automatons are cloned, each receives a unique instance number. The instance number is never repeated, unless the Event Stream Processor is restarted. **pnum**, **instance** identifies an automaton during its execution.
  - **var –** name of the bound variable. Besides these variables, bound events and constants are listed as well. The constants appear with unique compiler-generated names.
  - **value –** value of the bound variable, in string format. The values of bound events are reported here as NULL. Examines the events data kind to see the contents of the event rows.

- **`events` *`patternStream` [`patternNum`]* –** the events that have been parsed by the automaton so far, for each automaton in a pattern stream. This data kind returns a mix of records of different types. Record types are named after the input streams where they flow from.

  If the optional parameter *patternNum* is present, shows only the data for that pattern. The fields are:

  - **ESP_Pnum –** number, starting from 0, of the pattern being parsed by this automaton.
  - **ESP_Instance –** instance number of the automaton. As new automatons are cloned, each receives a unique instance number. The instance number is not repeated, unless the Event Stream Processor is restarted. **pnum**, **instance** identifies an automaton during its execution.
  - **ESP_Var –** name of the event variable.
  - **as in input stream –** the rest of the fields keep the names as in the row type of the input stream that they are from.
- **`expect` *`patternStream` [`patternNum`]* –** the expected records that would advance the automaton to the next state, for each automaton in a pattern stream. This data kind returns a mix of records of different types. Record types are named after the input streams that they flow from.

  If the optional parameter *patternNum* is present, shows only the data for that pattern. The fields are:

  - **ESP_Pnum –** number, starting from 0, of the pattern being parsed by this automaton.
  - **ESP_Instance –** instance number of the automaton. As new automatons are cloned, each receives a unique instance number. The instance number is not repeated, unless the Event Stream Processor is restarted. **pnum**, **instance** identifies an automaton during its execution.
  - **ESP_Var –** name of the event variable. If preceded by a "!", receiving such a record causes a pattern mismatch. Otherwise, it advances the automaton to the next state.
  - **as in input stream –** the rest of the fields keep the names as in the row type of the input stream that they are from. Only the fields that are bound to values are shown, the rest are shown as NULL.
- **exf *`kind` [`stream` [`object`]] `filter`* –** similar to **ex**, but specifies a filter SPLASH expression to be evaluated on the Event Stream Processor. Only records for which the filter evaluates to a true (non-zero, non- NULL) value are returned. With filters, any transaction and update pair boundaries are lost; each record returns by itself.

  The filter may refer to predefined variables with names matching the XML tags of the rows when printed. For most data kinds, the variable row contains the current record to be filtered. For the input data kinds multiple variables are defined, each is named after an input stream of the target stream. In this case when evaluating a record, the variable matching the stream of its origin contains the record, and all other variables are set to

NULL. The condition may refer to the fields in the records as usual, for example **"currow.field"**.

- **eval `*stream*` `*block*` –** evaluate a SPLASH statement (not expression) on a stream, to change the contents of the local variables (those defined in the local **DECLARE** block or global **DECLARE** block) of the stream. The variables that are defined inside the SPLASH blocks of a stream exist only when the appropriate methods run, and cannot be modified. Evaluation in context of any computational stream is used to modify the global variables.

  The SPLASH statement must be either a simple statement terminated by ";" or a block enclosed in braces "{}". Multiple statements must always be enclosed in a block. If you use braces to quote the block argument, the outside braces do not count as the block delimiters (they are just **esp_client** quotes).

Correct Example:

```
`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); }}
```

Incorrect Example:

```
`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); }
```

All the usual SPLASH syntax applies, including that for defining the temporary variables in the block. All of the stream's variables and global variables are visible and may be read or changed in the statement. No streams or stream iterators are visible in the statement.

You cannot use back quotes and curly braces when entering multiline statements. In the previous examples, splitting the lines represents the wrapping of the line on the terminal. In many cases, the multiline quoting format would be more convenient:

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

Operations on eventCaches require special preparation. Normally, the key of the eventCache is determined by the current input record. In the current example, there is no input record, so the key is not set and operations on eventCaches have no effect. For operations to work, you must manually set the key using the operator **keyCache(ec-variable, record)**, before performing any aggregation operations on the eventCache.

Output Control Flags

- **hdr *[on|off]* –** without an argument, displays the state of the "include column name header line" flag, otherwise enables or disables the "include column name header line" flag. When **hdr** is enabled, the column name heading prints before the tabular data. For snapshots, the field position, name, and field type appear. If the field is a key field, the field name is prefixed with an asterisk ("*").
- **sphdr *[on|off]* –** without an argument, displays the state of the "include header/data prefix" flag, otherwise enables or disables "include header/data prefix" flag. When **sphdr** is enabled, the StreamName and opcode values prefix each line of the tabular snapshot data. In addition, if the **hdr** flag is enabled, the header line includes the Event Stream Processor **StreamName** and **opcode** field names. The Event Stream Processor header/prefix are the **putd** and **putx esp_client** commands.
- **txb *[on|off]* –** without an argument, displays the state of the "include TRANSACTION BLOCK content" flag, otherwise enables or disables the "include TRANSACTION BLOCK content" flag. When **txb** is enabled, the output produced by the snapshot command includes all messages contained within the Gateway I/O TRANSACTION blocks. If **txb** is disabled, the tabular snapshot output is generated using only the INSERT messages from the TRANSACTION blocks.
- **verbose *[on|off]* –** without an argument, displays the state of the "verbose" output flag, otherwise enables or disables the "verbose" output flag. When **verbose** is enabled, this flag produces additional output when processing the snapshot commands. In particular, start_sync, end_sync, TRANSACTION block indicators, final snapshot, and record/row count are produced.
- **xml *[on|off]* –** without an argument, displays the state of the "XML" output flag, otherwise enables or disables the "XML" output flag. When **xml** is enabled, the output that is produced by the snapshot command is in the ESP XML record format. The XML format is used by the **putx** command.

Put Command Notes

The **putd** and **putx** commands use the **sphdr StreamName** and **OpCode** prefix. The date strings are in the format:

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

The TZ environment variable is set to "UTC" before the record is uploaded to the Gateway interface.

Usage Notes

In console mode, you can issue commands from the console, which allows for command-line editing and command history retrieval. Enter this mode using the following command:

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

In command-string mode, feed in a double-quoted string containing one or more commands. If you specify multiple commands in the double-quoted string, terminate each command with a semicolon character. When setting the field separator from the command line, enclose the new field separator character within single quotes, and place a space character between the ending single quote and the semicolon.

CHAPTER 4 Publish and Subscribe Executables

Use the command-line utilities to subscribe to, convert, and publish data from Event Stream Processor.

esp_convert

Converts XML and delimited records into 32-bit binary records compatible with Event Stream Processor. The metadata describing the streams is obtained either from a connection to a running instance of the Event Stream Processor (via the Command and Control interface) or via an Event Stream Processor compatible configuration file.

Syntax

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

Required Arguments

- `-c` ***user[:password]*** **–** (required) authenticates with a *user* ID and, optionally, a *password*. If you do not provide either the *password* or the **-k** or **-G** option, you are prompted for the password. If the Event Stream Processor successfully authenticates with these credentials, the connection is maintained, otherwise the Event Stream Processor immediately closes the connection.
- `-f` ***configFile*** **–** (dependent required) specifies the XML style configuration file that describes the data to be converted to Event Stream Processor format.
- `-p` ***[<host>:]<port>/workspace-name/project-name*** **–** (required) together, the *host:<port>/<workspace name>/<project name>* arguments specify the URI to connect to the ESP Server (cluster manager). For example, if you have started your ESP cluster server in the port **19011**, the host name is set as **localhost**, and you are running a project called **prj1** in the default workspace, specify `-p` as: `-p localhost:19011/default/prj1`

Options

- `-b` **–** (optional) indicates that the machine architecture on which the server (that is consuming data) is running has the reverse byte order of the machine architecture on which **esp_convert** is running.

- **`-d` *separator* –** (optional) reads and converts delimited records from standard input instead of the default XML format.
- **`-e` –** (optional) encrypts traffic with the Event Stream Processor via openSSL sockets.

  Note: Ensure that the Event Stream Processor is started in encrypted mode to use this option.
- **`-f` *configfile* –** (dependent required) specifies the CCX configuration file that describes the location of data that is read via standard input.
- **`-F` *path* –** (optional) specifies the full pathname of the XML Schema file (default is `$ESP_HOME/etc/Platform.xsd`).
- **`-G` –** (optional) authenticates access to the Event Stream Processor with credentials held within a Kerberos authentication ticket. Environment variables determine where the system will look to find authentication tickets (See *Administrators Guide > Security in the Event Stream Processor > Authentication > Setting Environment Variables for Kerberos* for more information). If the user name differs from the default principal name in the ticket cache, specify an alternate user name with the **`–c`** option to use the corresponding authentication ticket.
- **`-h` –** (optional) prints a list of possible options on the screen along with a brief explanation for each option.
- **`-k` *privateRsaKeyFile* –** (optional) performs authentication using the RSA private key file mechanism instead of password authentication. The *privateRsaKeyFile* must specify the pathname of the private RSA key file.

  Note: Ensure that the ESP Server has been started with the `-k` option specifying the directory in which to store the RSA keys.
- **`-m` *datetimemask* –** (optional) specifies the format string for date values (in strptime format). Default value is "%Y-%m-%dT%H:%M:%S".
- **`-v` –** (optional) prints the **esp_convert** utility version.

Examples

- **Converting Records to Binary Format –** Convert records from XML or a comma separated value (CSV) file.

  To convert all XML records in file `foo.xml` to native binary format, and post them to a running instance of the Event Stream Processor, use:

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

  To convert all comma-separated records in file `foo.csv` to native binary format, and post them to a running instance of the Event Stream Processor, use:

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

```
esp_upload -p localhost:19011/default/prj1
```

To convert all XML records in file `foo.xml` to native binary format, and post them to a running instance of the Event Stream Processor on a target machine HOST that has a differing byte order than the machine on which **esp_upload** is running, use:

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

To convert all XML records in file `foo.xml` to native binary format and post them to standard output without a running instance of ESP, use the `-f` option and provide the CCX configuration file used to start the server.

```
cat foo.xml | $ESP_HOME/bin/esp_convert -f test.ccx
```

Input Formats

Records in a delimited or XML format.

This is a record in a delimited format.

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

All columns must be present in the delimited form and the row must end with a line feed character. Operation is a single character, {i|u|d|s|p} for insert, update, delete, safe delete (delete only if the record exists), and upsert respectively.

This is a record in an XML format.

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

If ESP_OPS is not present, operation is taken as an upsert. There is no requirement on the number of columns present. Those that are missing are taken to have null values. If ESP_FLAGS is present, it can only have the value "s" indicating that the SHINE flag should be set for the record.

esp_kdbin

Reads data from a KDB database table into an Event Stream Processor stream.

The **esp_kdbin** adapter reads data from a KDB database into a stream in the Event Stream Processor. You can configure the adapter to read either queried or streaming data, based on a configuration parameter.

By default, the adapter matches the field names (in a case-insensitive manner) to decide the mapping between the source KDB table and the target stream. You also have the option of explicitly specifying the mapping.

Syntax

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

Required Arguments

- **`-c`** ***user[:password]* –** Passes authentication credentials to the Event Stream Processor. If you do not provide either the *password* or the **`-k`** option, you are prompted for the password. If the Event Stream Processor successfully authenticates with these credentials, the connection is maintained, otherwise the Event Stream Processor will immediately close the connection.
- **`-H`** ***kdbhost:kdbport* –** Specifies the port number, or the host name and port number, on which KDB is listening. The default value is localhost:5001.
- **`-p`** ***[<host>:]<port>/workspace-name/project-name* –** Together, the *host:<port>/ <workspace name>/ <project name>* arguments specify the URI to connect to the ESP Server (cluster manager). For example, if you have started your ESP cluster server in the port **19011**, the host name is set as **localhost**, and you are running a project called **prj1** in the default workspace, specify `-p` as: `-p localhost:19011/default/prj1`
- **`-q`** ***source* –** Specifies the KDB table when running in streaming mode. When running in non-streaming mode, specifies a valid query string.
- **`-s`** ***stream* –** Specifies the target stream: the stream to which the data being read is published.

Options

- **`-a` –** Uses asynchronous mode transmission; where the adapter does not wait for acknowledgment from the Event Stream Processor that it received the data. This option is necessary when using a hot spare configuration to ensure that both the primary and the hot spare receive the data. The default value is 'non async'.

- **`-b` *blocksize* –** Specifies how many records to put in a block of data for transmission. A higher value may increase throughput but it will also increase latency. A block may contain fewer records than specified if there is not enough data available. The default value is 64.
- **`-d` –** Outputs debug messages.
- **`-e` –** Uses encrypted OpenSSL sockets for all communications between the adapter and the Event Stream Processor (which requires that it be started in encrypted mode). When this option is not present, no encryption occurs. The default is not encrypted.
- **`-g` *gatewayhost* –** Uses the specified gateway host. Ignore the host name returned by the Event Stream Processor.
- **`-h` –** Prints detailed help.
- **`-k` *privateRsaKeyFile* –** Performs authentication using the RSA private key file mechanism instead of password authentication. The *privateRsaKeyFile* must specify the pathname of the private RSA key file.

  Note: Ensure that the ESP Server has been started with the `-k` option specifying the directory in which to store the RSA keys.
- **`-I` *interval* –** Specifies the number of seconds to wait before running the supplied query again when running in nonstreaming mode. A value of `0` indicates that the query should only be run once; no polling is performed. The default value is 0.
- **`-M` *mapping* –** Specifies a mapping between the column name in the target stream and the column name in the KDB database table. The mapping is a colon-separated series of `SPColumn=KDBColumn` statements. If this parameter is not provided, the connector absorbs data only for those columns where the target stream column name matches the source table column name (in case-insensitive manner).
- **`-m` –** If this option is not specified, the adapter connects to a KDB database and reads in streaming data. If it is specified, the adapter executes the supplied database query and feeds the result to the Event Stream Processor. The default is to use streaming mode.
- **`-T` *attempts* –** Specifies the number of times to attempt to reconnect to the KDB database if the connection breaks during operation. The default value is 1.
- **`-t` –** Uses transaction blocks. This improves performance, but causes all records in a block to be rejected when one record fails. The default is to use envelopes.
- **`-u` *user:password* –** Passes authentication credentials to the KDB database.

Examples

Execute a basic streaming mode query that reads data from a KdbTrades table in a KDB database on the server myServer, where KDB is listening on port 9200, and writes it to the SpTrades stream in the Event Stream Processor on the local server where the Command and Control interface is on port 1190, use:

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

To execute the same query, explicitly mapping fields in the KDB database to columns in the Event Stream Processor stream, use:

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

To execute a pull mode operation that issues the specified query every 5 seconds to a KDB database on the server myServer, where KDB is listening on port 9200, and writes data to the Event Stream Processor, on the server outputServer, where the Command and Control interface is on port 1221, use:

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

esp_kdbout

Feeds streaming data from the Event Stream Processor to a KDB database table.

By default, the adapter matches the field names (in a case-insensitive manner) to determine the mapping between and the Event Stream Processor stream and the KDB table. You can also explicitly specify the mapping.

Syntax

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

Required Arguments

- **`-c`** ***<user:password>* –** Passes authentication credentials to the Event Stream Processor. If you do not provide either the *password* or the **`-k`** option, you are prompted for the password. If the Event Stream Processor successfully authenticates with these credentials, the connection is maintained, otherwise, the Event Stream Processor immediately closes the connection.
- **`-p`** ***[<host>:]<port>/workspace-name/project-name* –** Together, the *host:<port>/ <workspace name>/ <project name>* arguments specify the URI to connect to the ESP server (cluster manager). For example, if you have started your ESP cluster server in the port **19011**, the host name is set as **localhost**, and you are running a project called **prj1** in the default workspace, specify `-p` as: `-p localhost:19011/default/prj1`
- **`-H`** ***kdbhost:kdbport* –** Specifies the port number, or the host name and port number, on which KDB is listening. The default host name is localhost.
- **`-q`** ***<query>* –** Specifies either the name of a stream on the Event Stream Processor or a valid SQL query to retrieve the data to write to the KDB table.
- **`-s`** ***<stream>* –** Specifies the name of the KDB table to which the data will be written.

Options

- **`-a` –** Uses asynchronous mode transmission; where the adapter does not wait for acknowledgment from Event Stream Processor that it received the data.
- **`-B` –** Uses droppable subscriptions. Event Stream Processor drops the subscription if the adapter cannot keep up with the data.
- **`-b` <*blocksize*> –** Allows you to set the maximum number of records to include in a single batch write to KDB table. The default is 5000.
- **`-d` –** Logs debug messages.
- **`-e` –** Uses encrypted OpenSSL sockets for all communications between the adapter and the Event Stream Processor (which requires that it be started in encrypted mode). When this option is not present, no encryption occurs.
- **`-h` –** Prints a list of possible options on the screen along with a brief explanation for each option.
- **`-I` –** Specifies a comma-separated list of KDB field names whose values will be ignored. These fields are included in the message, but are always populated with NULL.
- **`-k` <*privateRsaKeyFile*> –** Authenticates using the RSA private key file mechanism instead of a password. The *privateRsaKeyFile* must specify the pathname of the private RSA key file. With this option enabled, the user name must be specified with the `-c` option, but the password is not required. In addition, the ESP Server must have been started with the `-k` option specifying the directory in which to store the RSA keys.
- **`-L` <*interval*> –** Uses pulsed subscribe when connecting to the Event Stream Processor. The pulse *interval* is specified in seconds.
- **`-l` –** Uses “lossy” subscribe.
- **`-M` <*permutation*> –** Specifies a mapping between the column name in the target stream and the column name in the KDB database table. The mapping is a colon-separated series of SPColumn=KDBColumn statements. If this parameter is not provided, the connector absorbs data only for those columns where the target stream column name matches the source table column name (in a case-insensitive manner).
- **`-m` –** Writes the data to the table using the “upsert” operation. If this option is not used, the adapter works in streaming mode and uses the **u.upd** operation to write data to the KDB database.
- **`-n` –** Transactions only, no base data is received. Default value is false.
- **`-O` –** Specifies a comma-separated list of KDB field names to omit from the message. Unlike ignored fields which are part of the message but always NULL, these fields are not included in the message.
- **`-Q` –** Runs in quiet mode, no messages will be displayed.
- **`-R` –** Subscribes with shine through (if possible) so that previously received information is retained for any fields for which the update contains no new data.

- **`-T` <*numtries*> –** Specifies the number of times to attempt to reconnect to the KDB database if the connection breaks during operation. The default is 1.
- **`-t` –** Targets KDB table.
- **`-u` <*user:password*> –** Passes authentication credentials to the KDB database.
- **`-v` –** Prints the **esp_kdbout** utility version.

Examples

To subscribe to the `SpTrades` stream of project `default/prj1` in the Event Stream Processor on the server `myserver`, where the cluster manager is on port 1221, and stream the data out to KdbTrades in the KDB database on the server `outputserver`, where KDB is listening on port 9200, use:

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

To populate fields XXX and YYY in a KDB table may with NULL (because they have no corresponding data in the Event Stream Processor), use:

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

Sometimes a table in KDB may compute fields (for example, XXX and YYY) which should not be specified in the data update message. Doing so usually results in a length error in the database. To omit these fields from the update message altogether, use:

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

esp_rapexport

Functions as an adapter to modify data output by the Event Stream Processor so that it can be accepted as input by RAP - The Trading Edition. Since RAP can only handle inserts, deletes are dropped and updates are converted to inserts.

Syntax

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

Required Arguments

- **`-f` *config_file* –** Specifies the full pathname of the file containing information about which streams on the Event Stream Processor are providing the data to RAP.

- **`-p` *Dir* –** Specifies the full pathname of the directory containing the publisher file, which defines the multicast address used by the RAP subscriber.
- **`-t` *Dir* –** Specifies the full pathname of the directory containing RDS templates which map the columns in the streams on the Event Stream Processor to the appropriate database table and columns.

Examples

Shows how to run an adapter with a configuration file named `config.xml` and `templates` and `publish` subdirectories all under a home directory whose full path is contained in the RAP_HOME environment variable:

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

esp_subscribe

Connects to an instance of the Event Stream Processor via the Command and Control and Gateway interfaces and subscribes to transaction streaming data. The received records are converted to XML (or optionally delimited format) and written to the standard output.

Syntax

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

Required Arguments

- **`-c` *user[:password]* –** Authenticates with a *user* ID and, optionally, a *password*. If you do not provide either the *password* or the **-k** or **-G** option, you are prompted for the password. If the Event Stream Processor successfully authenticates with these credentials, the connection is maintained, otherwise, the Event Stream Processor immediately closes the connection.
- **`-p` *host:port/workspace/project* –** Together, the *host:port/workspace/project* arguments specify the URI to connect to the ESP Server (cluster manager). For example, if you have started your ESP cluster server on port **19011** of your local machine, and you are running a project called **prj1** in the default workspace, specify `-p` as: `-p localhost:19011/default/prj1`

Options

- **`-A` –** Adds subscriptions to this *conn_handle* connection handle. **esp_subscribe** exits after processing the request.

- **`-b`** – Connects to the server in reverse byte order of the machine architecture on which **esp_subscribe** is running.
- **`-B`** – Drops the connection if it supplies data faster than **esp_subscribe** can absorb it.
- **`-d`** ***separator*** **–** Puts the subscribe client into delimited output mode and uses the specified *separator* as the delimiting character.
- **`-D`** ***conn_handle*** **–** Deletes subscriptions to this *conn_handle* connection handle. This **esp_subscribe** exits after processing the request.
- **`-e`** – Encrypts traffic with the Event Stream Processor via openSSL sockets.

  Note: Ensure that the Event Stream Processor is started in encrypted mode to use this option.
- **`-f`** ***configfile*** **–** Specifies the CCX configuration file that describes the location data that will be read via standard input.
- **`-F`** ***schemafile*** **–** Sets the XML schema file. The default schema file is `$ESP_HOME/etc/Platform.xsd`.
- **`-g`** ***gateway_host*** **–** Uses the specified gateway host rather than the host returned from the `get_gateway()` call.
- **`-G`** – (optional) authenticates access to the Event Stream Processor with credentials held within a Kerberos authentication ticket. Environment variables determine where the system will look to find authentication tickets (See *Administrators Guide > Security in the Event Stream Processor > Authentication > Setting Environment Variables for Kerberos* for more information). If the user name differs from the default principal name in the ticket cache, specify an alternate user name with the **`–c`** option to use the corresponding authentication ticket.
- **`-h`** – Prints a list of possible options on the screen along with a brief explanation for each option.
- **`-H`** – Subscribes to the Event Stream Processor heartbeat messages, (sID,Qd), ... (sID,qD).
- **`-i`** ***streamId[,...]*** **–** Specifies one or more streams to subscribe to, using each stream's integer handle.
- **`-k`** ***privateRsaKeyFile*** **–** Performs authentication by using the RSA private key file mechanism instead of password authentication. The *privateRsaKeyFile* must specify the pathname of the private RSA key file.

  Note: Ensure that the ESP Server has been started with the `-k` option specifying the directory in which to store the RSA keys.
- **`-l`** – Puts the subscribe client into lossy mode, where data is shed (lost) if the subscribe client cannot keep up with the streaming data produced by the Event Stream Processor.
- **`-L`** ***N*** **–** Specifies a pulsed subscription, where records are collected, and delivered every *N* seconds.
- **`-m`** ***conn_name*** **–** Sets a symbolic tag name for the connection. This allows **esp_subscribe** to look up the connection easily in the _ESP_Clients metadata stream. The tag is stored in

the *conn_tag* field. To kill the connections by tag name, use the **esp_client** to input the `kill every` command.

- **`-M` *number* –** Specifies the number of multiple connections to establish (all for the same streams).
- **`-P` *precision* –** Sets the number of decimal places in output for `FLOAT`. Default value is 6 decimal places.
- **`-Q` *SQL statement* –** Specifies an SQL select statement to apply to outbound records.

  The outbound records are marked with insert, update, and delete, even when the SQL statement might not include the key columns of the stream. To give insert/update/delete meaning in these cases, you can specify a special column ESP_SEQNO. For example, the statement `select *, ESP_SEQNO from Vwap order by Price` automatically fills in values for the ESP_SEQNO column. With `order by`, the ESP_SEQNO reflects the ordering of the rows in the output set.
- **`-R` –** Subscribes with shine through (if possible) so that when an update contains no new data for previously received fields, information for those fields is retained.
- **`-s` *stream_1 [,...,stream_N]* –** Specifies one or more streams using the logical name of each stream, to subscribe to snapshot the table, (exit after the SYNC records are received).
- **`-S` –** Takes a snapshot of the table: receive the initial state of the stream and exit immediately afterward.
- **`-t` –** Puts the subscribe client into transaction-only mode, so it receives transactional updates to streams upon connection to the Event Stream Processor, rather than the initial state of the streams.
- **`-T` –** Prints out transaction block begin `<block>`, and transaction block end `</block>` markers.
- **`-v` –** Reports special events (start synchronization, end synchronization, or stream exit).
- **`-W` *baseDrainTimeout* –** Sets the time limit, in milliseconds, that this client has to read all base data from the subscribe stream before being dropped. When this parameter is not specified, the default value is 8,000 milliseconds or 8 seconds.
- **`-X` –** Exits the process after all the subscriptions exit, even if the Event Stream Processor does not drop the connection.
- **`-z` *queueSize* –** Sets the subscriber queue size to *queueSize*, of the output gateway connection. This queue front ends the connected socket, buffering data to be delivered to the client. The minimum value for the *queueSize* is 1000 records. The default value is 8192 records.
- **`-V` –** Prints the **esp_subscribe** utility version.

Examples

to subscribe to two streams named PreprocessorTransactions and DebitMovements, of default/prj1 project running on a cluster manager on localhost:11180, printing all stream data in XML format on standard output:

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

```
prj1
```

To subscribe to two streams named PreprocessorTransactions and DebitMovements, printing all stream data in pipe-separated format on standard output:

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

To subscribe to one stream named PreprocessorTransactions that has data generated by a server running on a machine named HOST (which has differing byte order than the machine that subscribe is running on) and print all stream data in pipe-separated format on standard output:

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

To subscribe to one stream named baseInput and apply a SQL statement:

```
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
```

To subscribe to an error stream, named ErrorStream:

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

Extended Opcodes for esp_subscribe

Extended operations codes (opcodes) appear in the output when using esp_subscribe in verbose mode.

All extended opcode letters are preceded by ESP_OPS. Below is an example of the **ESP_START_SYNC** extended opcode:

```
ESP_OPS="s"
```

ESP_START_SYNC

- An extended opcode visible when using the subscribe tool with the '-v' option.
- Indicates that the base data (data in the window available when the subscription starts) is about to be delivered.
- Represented as the letter 's' when subscribing to Streams, Windows, and Delta Streams.

ESP_END_SYNC

- An extended opcode visible when using the subscribe tool with the '-v' option.
- Indicates that the base data (data in the window available when the subscription starts) has been delivered.
- Represented as the letter 'e' when subscribing to Streams, Windows, and Delta Streams.

ESP_WIPEOUT

- An extended opcode visible when using the subscribe tool with the '-v' option.
- This opcode is received when running in High Availability mode and the primary server goes down; and you are going to receive a fresh set of base data from the secondary server.
- Represented as the letter 'w' when subscribing to Streams, Windows, and Delta Streams.

ESP_STREAM_EXIT

- An extended opcode visible when using the subscribe tool with the '-v' option.
- This opcode is received when the stream that is being subscribed to has exited. You will no longer receive more data from this Stream/Window/Delta Stream.
- Represented as the letter 'X' when subscribing to Streams, Windows, and Delta Streams.

ESP_DATA_LOST

- An extended opcode visible when using the subscribe tool with the '-v' option.
- This opcode is received when subscribing to data in lossy mode. The server purges some data because the subscribe tool is not able to keep up with the server.
- Represented as the letter 'l' (lowercase L) when subscribing to Streams, Windows, and Delta Streams.

esp_upload

Records binary records from the standard input and forwards them to a running instance of the Event Stream Processor via the Gateway interface.

The format of the data is zero or more occurrences of `<Stream Handle><Raw Binary Record>`. `<Stream Handle>` is an `uint32_t` indicating the destination stream for the record. This tool is typically used at the end of a pipeline with the **esp_convert** tool.

Syntax

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

Required Arguments

- **`-c`** ***user[:password]*** **–** Authenticates with a *user* ID and, optionally, a *password*. If you do not provide either the *password* or the **-k** or **-G** option, you are prompted for the password. If the Event Stream Processor successfully authenticates with these credentials, the connection is maintained, otherwise the Event Stream Processor immediately closes the connection.

Options

- **`-b`** – Sets byteswap mode. The raw records fed into **esp_upload** (via **esp_convert** `-b`) and the server to which **esp_upload** is sending data have different byte orders than the architecture on which the **esp_upload** client is running (the byte order of the data must always match the byte order of the server).

- **`-d` *N* –** Inserts a delay of *N* microseconds between records or transaction blocks.
- **`-e` –** Encrypts traffic with the Event Stream Processor via openSSL sockets.

Note: Ensure that the Event Stream Processor is started in encrypted mode to use this option.

- **`-f` *timeout:finalizer* –** Sets a finalizer to be run. The ESP Server runs the SQL finalizer statement (a combination of insert, update, or delete statements, separated by semicolons) if no message is received from **esp_upload** within *timeout* milliseconds. The SQL statement is also run when **esp_upload** stops.
- **`-G` –** (optional) authenticates access to the Event Stream Processor with credentials held within a Kerberos authentication ticket. Environment variables determine where the system will look to find authentication tickets (See *Administrators Guide > Security in the Event Stream Processor > Authentication > Setting Environment Variables for Kerberos* for more information). If the user name differs from the default principal name in the ticket cache, specify an alternate user name with the **`–c`** option to use the corresponding authentication ticket.
- **`-h` –** Prints a list of possible options on the screen along with a brief explanation for each option.
- **`-k` *privateRsaKeyFile* –** Performs authentication by using the RSA private key file mechanism instead of password authentication. The *privateRsaKeyFile* must specify the pathname of the private RSA key file. Ensure that the ESP Server has been started with the `-k` option specifying the directory in which to store the RSA keys.

Note: With this option enabled, the user name must be specified with the `-c` option, but the password is not required.

- **`-m` *conn_name* –** Sets a symbolic tag name for the connection. This allows the **esp_upload** to look up the connection easily in the _ESP_Clients metadata stream. To kill the connections by tag name, use the **esp_client** command.
- **`-p` *[<host>:]<port>/workspace-name/project-name* –** (required) Together, the *host:<port>/<workspace name>/<project name>* arguments specify the URI to connect to the ESP Server (cluster manager). For example, if you have started your ESP cluster server in the port **19011**, the host name is set as **localhost**, and you are running a project called **prj1** in the default workspace, specify `-p` as: `-p localhost:19011/default/prj1`
- **`-r` *N* –** Uploads records or transaction blocks at a rate of *N* per second. The default is to upload as fast as the server can absorb the data.
- **`-s` *N* –** Synchronizes the source streams every *N* records or transaction blocks. This guarantees that the records have been absorbed by the source streams. By default, source streams are not synchronized.
- **`-t` *size* –** Runs in transaction mode. Each record that **esp_upload** reads is buffered on a per-stream basis. When a buffer reaches the indicated number of records, it is wrapped as a

single transaction and sent to the Event Stream Processor. If all records read are from one stream, this effectively buffers the stream into *size* record chunks and commits them as transactions. Any buffered records are sent as a single transaction per stream when an EOF is read.
- **`-w` *size* –** Runs in envelope mode. Each record that **esp_upload** reads is buffered on a per-stream basis. When a buffer reaches the indicated number of records, it is wrapped in a single envelope and sent to the Event Stream Processor. If all records read are from one stream, this effectively buffers the stream into *size* record chunks. Any buffered records are sent as a single envelope per stream when an EOF is read.
- **`-x` –** Upon receiving an EOF on the standard input, sends an <END OF STREAM> marker to each stream for which data has been uploaded. If all source streams of the Event Stream Processor receive an <END OF STREAM> marker, the Event Stream Processor shuts down and exits.
- **`-X` –** Forces the Event Stream Processor to exit when upload completes.
- **`-Y` *<beat>* –** Forces the Event Stream Processor to exit if no data is received in *<beat>* microseconds.
- **`-v` –** Prints the **esp_upload** utility version.

Examples

For a description of the format of the CSV and XML input files, see **esp_convert**.

To convert all XML records in file `foo.xml` to native binary format and post them to a running instance of the Event Stream Processor:

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

To convert all comma-separated records in the `foo.csv` file to native binary format and post them to a running instance of the Event Stream Processor:

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

To convert all XML records in the `foo.xml` file to native binary format and post them to a running instance of the Event Stream Processor on a target machine HOST which has a differing byte order than the machine on which **esp_upload** is running:

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

CHAPTER 5 Authoring Executables

esp_compiler

Translates a given set of CCL statements to the corresponding CCX representation to be consumed by the Event Stream Processor. Also, verifies the correctness of the CCL statements, checks for datatype consistencies, and performs limited optimization.

Syntax

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

Required Arguments

- **`-i`** ***CCL File* –** Uses the specified file as the input, which translates to CCX.

Options

- **`-o`** ***CCX File* –** Writes the CCX output to the named file; this overwrites the file if it already exists. If not specified, by default, the CCX is written to standard output.
- **`-v` –** Prints the **esp_compiler** utility version.

esp_studio

Starts the ESP Studio. Use Studio to visually author projects (in other words, continuous queries) and to kick off and monitor ESP Server execution of those services.

Note: To find version information on `esp_studio`, launch the ESP Studio and access the About dialog. The `-v` option, used with other utilities, is not available with `esp_studio`.

Syntax

```
esp_studio
```

CHAPTER 6 Advanced Debugging

Introduction

The Sybase Event Stream Processor includes debugging features for locating and fixing problems in your projects.

These features are available through two interfaces: the Event Stream Processor Studio and the **esp_client** command-line utility. Use the debugging tools during project development, not while the Event Stream Processor is in production mode. The debugging tools are normally disabled since they place a substantial overhead on the Event Stream Processor.

To enable the debugging tools, run the Event Stream Processor in trace mode.

Trace Mode

In trace mode, the Event Stream Processor performs extra checks for possible debugging operations and breakpoints, and collects extra information about execution history.

Use the **esp_client** utility to enable, disable, and the check the status of trace mode.

For example, in an instance of **esp_client**, check the status of trace mode, turn it on, check the status again, then turn it off:

For example, in an instance of **esp_client**, check the status of Trace Mode, turn it on, check the status again, then turn it off:

```
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
```

Trace mode is not associated with any instance of **esp_client**: you can turn on trace mode and exit **esp_client**; the Event Stream Processor remains in trace mode until you turn it off in another instance of **esp_client**.

Step the Event Stream Processor

Stepping lets you advance the state of the Event Stream Processor when the system is paused.

You can advance a stream by a single step or multiple steps. To produce substantial changes, a stream may require multiple steps, so users may allow the system to run an entire transaction or run until the streams are quiesced. Users may also choose to run the system again, which runs normally until another breakpoint is triggered.

Automatic Single-Stepping

Automatic stepping does not function with breakpoints or bad row exceptions; any encountered breakpoints or bad row exceptions will be reported to Studio, but this will not stop the stepping.

The first auto step command, **step trans**, steps to the end of a transaction. This command does at least one common step, and then continues stepping as long as the stream stays in the COMPUTE location. The stream stops when it moves into the PUT or BAD_ROW location and allows users to examine the transaction's effect before committing or discarding it. Call **step trans** repeatedly to step past transactions.

If the execution blocks the INPUT or OUTPUT location for longer than 0.3 second, **step trans** stops.

The other auto-stepping commands are related to the concept of quiescence (running the streams until they have processed all the available input). These commands are:

Command	Function
step quiesce stream *{streamName}*	Automatically steps the stream and all of its direct and indirect descendants until they are quiesced (until all their input queues are empty).
step quiesce downstream *{streamName}*	Only the stream's descendants are stepped: the stream itself is not. Use this command to clear out the descendant streams' input queues. When the argument stream produces its output, the progression of the data through the descendant streams can be easily traced.
step quiesce from base	Automatically steps all derived (nonsource) streams until their input queues are empty. Use this command to clean out the queues of derived streams before processing an inconsistent record through the source stream.

Note: If you exit **esp_client** or Studio while the Event Stream Processor is paused, it remains paused. When you connect with another instance of **esp_client**, it still remains paused. If you leave the Event Stream Processor in trace mode, it remains in that mode: if it encounters a breakpoint or exception, it pauses, and stops all processing until unpaused.

You can stop the Event Stream Processor from **esp_client** even when it is paused. The esp_client utility will unpause the Event Stream Processor and disable trace mode before stopping it.

Disabling trace mode also unpauses the Event Stream Processor.

Pause the Event Stream Processor

While in trace mode, you can issue a pause command to pause the stream processing loop. Pausing the loop allows you to examine the stream in a static state.

While the Event Stream Processor is in trace mode, the stream processing mechanism checks for a pause request as it enters each processing group location. Once you issue a pause command, the stream pauses and the does not resume the loop until you allow it to continue. When the Event Stream Processor is paused, no calculations happen. Any transactions in the streams' output buffers are still consumable by subscribers. Publishing to the paused stream continues until the stream's input buffers are full.

If the stream is engaged in actual processing when the pause is requested, processing continues until it enters the next location.

Streams are not affected by a pause request when in an I/O location. For example, the stream pauses automatically in the INPUT location if there are no transactions in the input queue and in the OUTPUT location if the output queue is full.

The stream may move between the I/O locations to a processing location even when it is paused. If there is a slow subscriber on a stream, the stream's output fills up, and the stream remains in the OUTPUT location, until the buffer space becomes available. If the stream in this location receives a pause request, the request is ignored. If the stream's subscriber takes a transaction off the output buffer after this request, the stream deposits its current output transaction on the buffer and goes to the INPUT location. At this point, after entering the INPUT location, the stream recognizes the pause request. No new data is processed after the pause request, but the stream does change its location.

Pause metadata streams like any other stream. No updates should come from these streams while the Event Stream Processor is paused, except for _ESP_RunUpdates.

Use the **esp_client** command-line utility to check the pause status of the Event Stream Processor, to pause, and to unpause:

```
esp_client> check_pause
Sybase Event Stream Processor is not paused
esp_client> pause
esp_client> check_pause
PAUSED
esp_client> run
esp_client> check_pause
Sybase Event Stream Processor is not paused
```

The Stream Processing Loop

The internal logic of a stream in the Event Stream Processor can be represented as a loop with states that correspond to ways the Event Stream Processor handles data.

A normal processing sequence proceeds as follows:

1. INPUT

The stream waits for the input queue to become non-empty, then picks a transaction from the head of the input queue. The transaction is visible as inTrans, the current input transaction. The transaction is processed row-by-row.

The current output transaction is set to be empty, prepared to collect the results of processing.

2. COMPUTE

The next record is selected from the current input transaction. It becomes visible as inRow, the current input row. In some cases, the current input record may actually be two records, combined into an UPDATE_BLOCK.

If this is not the first iteration of the loop, the records produced from processing the previous input record are still visible as outRow.

If there are any input breakpoints defined on the stream, they are evaluated against the current input record, which may trigger an Event Stream Processor pause.

The check as to whether the Event Stream Processor is paused is performed. If paused, the stream pauses here and waits for permission to continue.

Finally, the actual computation is performed on the current input record. It produces zero or more output records. These records become visible as outRow, and are also appended to the end of outTrans. These records follow certain internal rules, and are not exactly the same as when they are published externally. For example, the update records at this point usually have the operation type UPSERT, and the delete records are SAFEDELETE.

Any output breakpoints defined on the stream are evaluated against the current input record, which may trigger an Event Stream Processor pause.

If there are more records left in the input transaction, the compute loop continues; otherwise, the stream proceeds to put the calculated data into the store, unless an exception such as division by zero has happened, in which case it proceeds to the BAD_ROW processing.

3. PUT

The Event Stream Processor is checked to see whether paused. If so, the stream pauses here and waits for permission to continue.

The new result is placed into the stream's store. This is not a simple process, as the result transaction gets cleaned and transformed according to the information already in the store. Because of this, the current output transaction is invisible after this point. There is no current output row either. Some of these transformations are:

- SAFEDELETEs: are either thrown away (if there was no such record in the store) or converted to DELETEs (filled with all the data that they had in the store before being deleted).
- UPSERTs: They are transformed into either INSERTS or UPDATE_BLOCKs. Any remaining UPDATEs are converted to UPDATE_BLOCKs , or may be discarded if no data is changed in the record from its previous state. An UPDATE_BLOCK is a pair of records; the first one has the operation type UPDATE_BLOCK and contains the new values, while the second one has the operation type DELETE and contains the old values. When an UPDATE_BLOCK is published to outside the Event Stream

Processor, the second record is discarded and the first one is converted to an UPDATE. Inside the Event Stream Processor, the entire update block is visible.

The PUT may trigger an exception too, for example, when trying to insert a record with a key that is already in the store. In this case, the entire transaction is aborted and the stream moves to the BAD_ROW location.

The current input transaction and current output transaction (already transformed) are inserted into the stream's history. The input transaction is added to the end of inHist, the output transaction appended to the end of outHist. Since the processing is done now, inTrans and inRow become invisible, outTrans and outRow are already invisible by this time.

4. OUTPUT

   The result transaction is queued to be published to the clients. If some clients are too slow and the output buffer fills, the stream waits for buffer space to become available.

   The result transaction is delivered to any streams that have this stream as their input. Again, if any of their input queues become full, this stream waits for them to become available.

   The stream then goes to INPUT the next transaction.

Besides this main loop, there are side branches. For the streams with expiry, the following side branch occurs every second:

- EXPIRY

Breakpoints and Exceptions

The breakpoint function pauses Event Stream Processor if there is a problem with the data model. A breakpoint stops Event Stream Processor and allows the user to troubleshoot the problem.

There are two options for pausing the system: sending an explicit pause command or setting a breakpoint on a stream or window (this includes local streams). You can set breakpoints on a stream's input or output; there is no limit to the number of breakpoints you can add. When an event hits a breakpoint, the execution is paused.

You can set specific conditions on a stream or window breakpoint. For example, you can place a filter on a breakpoint so only certain records trigger the pause. You can also set a time condition, which allows the user to trigger the breakpoint after a specified number of records.

There are two kinds of breakpoints for streams:

- On Input – these breakpoints are checked when a row is taken from the input transaction for processing, before any computation is performed. Event Stream Processor is paused in the COMPUTE location.
  - On any input – checked for input from any stream.
  - On a particular stream – checked only when the input transaction is coming from a particular stream.

- On Output – these breakpoints are checked after a row has been processed. Event Stream Processor is paused in the next location (COMPUTE for the next input row, PUT or BAD_ROW).

Unconditional Breakpoints and Exceptions

A simple breakpoint is unconditional: once the stream is in the right location, the breakpoint is triggered and the Event Stream Processor is paused. You can trigger more than one breakpoint simultaneously. Multiple breakpoints can be defined in the exact same location. The Event Stream Processor can tell one breakpoint from another by the unique ID assigned when the breakpoint is created. If you create two breakpoints that are the same, Event Stream Processor gives each one a separate ID, allowing both breakpoints to be triggered at the same time.

Once a breakpoint is created, you can enable it, disable it, or alter some of its information. The breakpoint cannot be moved to another location or have its conditional expression changed. To make major changes, you must delete the breakpoint and create a new one.

You may pause the Event Stream Processor at any record. For example, if a bug surfaces on the 1000th record passing though a stream, you can let 999 records pass, then pause and single-step from that point. To do this, configure a breakpoint to trigger on every nth row. If you restart the Event Stream Processor, the breakpoint triggers on the 2000th row next. In `bp list`, the field `enabledEvery` shows the breakpoint number as `N`, and `leftToTrigger` shows how many records remain to be seen before the breakpoint is triggered. Every time the breakpoint is triggered, `leftToTrigger` is reset to the original value of `enabledEvery`. By default, when you create a breakpoint, `enabledEvery` is set to 1, to trigger on each row. It can be changed with the **esp_client** command **bp every**.

Users can disable a breakpoint temporarily by configuring the breakpoint to trigger on every 0th record or by using the **bp on** command, which triggers the breakpoint on each record.

Conditional Breakpoints and Exceptions

Use conditional breakpoints to see a record with certain contents pass through a stream.

You can apply conditional breakpoints on Event Stream Processor by specifying a filter expression for a breakpoint. The filter expression is evaluated first and if it results in a false (0 or NULL) value, the breakpoint is skipped. The breakpoint is triggered only if the expression results in a true value (or its `leftToTrigger` count is reduced).

The filter expression is standard in SPLASH. It uses the data from two pre-defined record variables: `currow` and `oldrow`. The variable `currow` contains the current record; `oldrow` is defined only for the breakpoints on input. For INSERT and a plain UPDATE the value of `oldrow` is NULL. For UPDATE_ BLOCK `oldrow` contains the second record of the block, the old data that is being replaced. For DELETE and SAFEDELETE `oldrow` contains the same data as `currow`. A particular field can be accessed using the usual `currow.field` syntax. You can obtain the row operation code using **getOpcode(currow)** .

The row definition that provides these predefined variables changes with different types of breakpoints.

For a breakpoint on output, the breakpoint is defined on the Row Definition. The expression is evaluated on the output rows produced during the preceding COMPUTE. Since multiple rows can be produced, the expression is evaluated on each. If no rows are produced, the expression is evaluated once with `currow` set to NULL. In this case, `oldrow` is not available because the UPDATE_ BLOCK is not produced on output of COMPUTE.

For a breakpoint on input from a specific stream, the breakpoint is defined on the Row Definition of that input stream. The expression is evaluated on the record or update block that is about to be computed. Filter expressions are not permitted for this kind of breakpoint.

A source stream receives data from outside of the Event Stream Processor instead of other streams. To add a conditional breakpoint on the input of a source stream, use the source stream's own name for the input stream with **bp add** *{filterInput} {filterInput}*.

Notification of Debugger Events

You can receive notifications as the Event Stream Processor changes between running and pausing, single-steps, and when it hits breakpoints and exceptions.

To receive these updates, you can subscribe to the _ESP_RunUpdates stream. This stream does not retain any content; notifications bypass the stream's store, and always have the operations type UPDATE. See *CCL Reference Guide* for more details.

Sample Debugging: Pausing the Event Stream Processor

Use the **esp_client** command-line utility to check the pause status of the Event Stream Processor, to pause, and to unpause.

1. To pause the Event Stream Processor, use:

   ```
   esp_client> pause
   ```
2. To run the Event Stream Processor and continue execution, use:

   ```
   esp_client> run
   ```
3. To check if the Event Stream Processor is paused, use:

   ```
   esp_client> check_pause
   ```

   The Event Stream Processor notifies the user of whether it is paused or not. If it is not paused, you see `Sybase Event Stream Processor is not paused`. If it is paused, you see `PAUSED`.

Sample Debugging: Stepping the Event Stream Processor

Use the **esp_client** command-line utility to advance a stream by a single step.
Advance a stream by a single step using:

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

Let the Event Stream Processor pick a stream that requires processing using:

```
esp_client> step
```

Automatic Stepping

The **esp_client** command line utility offers many options for automatic stepping. These commands eliminate the need for continuous single-stepping.

1. Advance a stream to the end of a transaction using:
   ```
   esp_client> step trans [`stream`]
   ```
2. Step a stream and all of its direct and indirect descendants until all of their input queues are empty using:
   ```
   esp_client> step quiesce stream {streamName}
   ```
3. Step only the stream's descendants until their input queues are empty using:
   ```
   esp_client> step quiesce downstream {streamName}
   ```
4. Step all derived streams until their input queues are empty using:
   ```
   esp_client> step quiesce from base
   ```

Sample Debugging: Adding Breakpoints

Use the **esp_client** command-line utility to add or delete breakpoints from a stream or window. Pause the Event Stream Processor before adding a breakpoint.

1. Add a breakpoint on a stream, before it starts processing an input record from any stream using:
   ```
   bp add `stream` any
   ```
2. Delete the breakpoint with specified ID using:
   ```
   bp del `id`
   ```
   The ID of a breakpoint is given by using either the **bp add** or **bp list** commands.
3. Delete all breakpoints using:
   ```
   bp del all
   ```
4. Enable or disable a breakpoint with specified ID using:

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

5. Enable or disable all breakpoints using:

```
bp on|off all
```

6. Make the breakpoint with specified ID trigger on every nth occasion.

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

7. Make all the breakpoints trigger on every nth occasion using:

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

8. List all created breakpoints using:

```
bp list
```

Sample Debugging: Adding Conditional Breakpoints

The **esp_client** command-line utility provides options for adding conditions to a breakpoint. These breakpoints trigger only when the condition evaluates at true.

1. Add a breakpoint on a stream, before it begins processing an input record from another stream using:

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

2. Add a breakpoint on a stream, after it has processed an input record and produced an output using:

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

Data Examination

The **examine** command lets you view different data types produced by various streams.

The examination commands, which work only when Event Stream Processor is paused, lets you view different datatypes produced by various streams.

The examination commands return the records in the same format used to send updates to common subscribers. The **esp_client** command generates records in XML format. The operation types that occur in examined data include the standard types seen by the normal subscribers, and the types that are used exclusively inside Event Stream Processor.

There are two ways in which records returned by the examination commands may be grouped:

- Two records can be grouped into an update block. These are printed by **esp_client** within an XML element named <pair>.
- Multiple records and update blocks can be grouped into a transaction. These transactions are printed by **esp_client** within an XML element <trans>. If a transaction contains only one record, it is printed as a single record, without the <trans> wrapping.

If a stream employs an input window, as this windows fills, it begins generating SAFEDELETEs for the earlier records. To distinguish these records from the DELETEs sent by the input streams, **esp_client** includes the pseudo-field ESP_RETENTION=1 in each record.

These arguments choose the data to be examined:

Argument	Function
kind	Determines the kind of data to be examined.
stream	Specifies the stream from which the data is taken. To receive data from all streams, leave this field empty.
object	Selects the particular data unit. Use this if there are many units of this kind of data (for example, variables).

Either the stream, data, or both can be left empty or omitted if it is not applicable to the kind of data requested in the **examine** command. The kind and stream must match: you cannot request data across the Event Stream Processor from a stream, and you cannot request per-stream data unless you specify a stream.

Some kinds of data are only available from certain streams. For example, the pattern state can be read only from a pattern stream. If the requested kind of data is not available for a certain stream, the error `No such kind of data` is returned, even if that kind of data is supported for other streams.

Some kinds of data can be used both with and without the stream argument. For example, you can use “var” without a stream to examine global variables, and with a stream to examine that stream's variables.

The groups of data related to the input queues are heterogeneous. Each record receives a tag matching the name of the stream that produced it (records produced from source streams receive tags with the name of the source stream itself). These include: `queue`, `inTrans`, `inRow`, `queueHead`, `queueTail`, `inHist`, `lastInTrans`, `inHistEarliest`, and `inHistLatest`.

There are also groupings of historic data that contain a mix of input and output data. Each of these groups contains one or more pairs of transactions: the first record in each pair is an input transaction, and the second record is the matching output transaction. Each input transaction record is tagged with the name of the stream that produced it; each output transaction record gets the tag `currow`. If any input stream records are also tagged with row, you can distinguish by looking at the order in which they appear.

Certain kinds of data deal with history, such as the input transactions processed and outputs produced. You can obtain these data sets separately (as input history and output history) or in a combined data set. When the input and output history are examined separately, the transactions are matched by their index: the first input transaction matches the first output transaction, and so on.

The amount of historical data kept for a stream is determined by the stream's history size setting. This setting can be set globally for all streams, or set for individual streams, using the

esp_client command history. The default history size limit is 100 transactions. Using large history limits increases the memory usage of the Event Stream Processor.

If trace mode is off, all history is discarded, but the limit is kept. The next time trace mode is enabled, the history begins collecting again.

A record returned by the examine command should be stored in an empty placeholder. If there is no ambiguity with transaction boundaries (such as with `outTrans`), **esp_client** returns no data. In other cases, such as **hist**, a placeholder shows that the transaction occurred, but produced no output. A placeholder record includes all fields, including the key fields; the value of each is set to NULL.

Some types of data records use natural field names. Records, such as "pause", return metadata: the field names in this type of record are defined in the Event Stream Processor. Other types contain a mix of fields defined by the user and added by Event Stream Processor. In this type of record, the fields added by Event Stream Processor have the prefix `ESP_`. Do not use this prefix for user-defined fields.

Filters

Use filters to examine specific rows in a large volume of data. The data that is returned by the debugging commands can be refined to select only the relevant data.

Data is filtered in the Event Stream Processor, before it is sent out.

The **esp_client** command **exf** performs filtering as follows:

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

The names of the stream and object are optional, similar to the command **ex**. There is an added argument containing the filter expression. The filter expression references a pre-defined variable, similar to the breakpoint filter expressions containing the current row. This expression compares the row with the expression and decides if the row should be returned. Whenever the filter expression returns `true (non-0, non-NULL)`, the record is returned to the user and displays.

The rules for the defined variables are:

- If all rows in the returned data set are of the same type, they are wrapped in a single variable row.
- If the data kind contains rows of mixed types (the input or history data), multiple variables are defined, with names matching the XML tags printed for these records. Only one variable, matching the type of the current record, contains a value. All others are set to NULL.

Store Data

Place store data into a file using the debugging tools on the Event Stream Processor.

This process is similar to the attribute `ofile` in the stream's element which causes the stream's contents to be dumped to a file when the Event Stream Processor exits.

To dump stored data, use:

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

Data Manipulation

In **esp_client**, the **eval** command allows the debugging interface to change data within the Event Stream Processor. The contents of global and stream local variables (including event-Cache) can be changed using this functionality.

Data manipulation functionality works only when the Event Stream Processor is in trace mode and paused.

The **eval** command changes data by evaluating a SPLASH statement (or block) in the stream. Only the variables are visible, not the streams or stream iterators. Any SPLASH statements are permissible, including branching and loops, but writing an infinite loop indefinitely stops the Event Stream Processor. Temporary variables are also permissible inside the SPLASH block.

In most situations the key of the eventCache is determined by the current input record. There is no input record, so the key is not set and any operations on eventCaches have no effect. This requires the key to be set manually using the operator *keyCache(ec-variable, record)*. Ensure that you set this operator before performing any operations on eventCache. You may change the key more than once, even in a loop, allowing you to perform operations on multiple keys.

You may only modify the stream's local variables (those defined in the local **DECLARE** block) and global variables with the **eval** command. Variables defined inside the SPLASH blocks of a stream exist only when the appropriate methods are run. These variables cannot be modified.

If it is possible to evaluate a unit of code, then it is not an expression but a SPLASH statement. A SPLASH statement must either be a simple statement terminated by ";" or a block enclosed in braces "{}". Multiple statements must always be enclosed in a block. If braces are used to quote the block argument, the outside quotes do not function as the block delimiters: they are just **esp_client** quotation marks.

The following are examples of correct and incorrect blocking:

Correct Blocking

```
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); }}
```

Incorrect Blocking

```
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); }
```

CHAPTER 7 On-Demand Queries

Each Event Stream Processor instance has a single Command and Control server thread that handles requests for information (metadata) or instructions to perform tasks such as **quiesce** or **shutdown**.

The **esp_query** On-Demand SQL Query utility uses this process to send requests for information to Event Stream Processor.

The **esp_cnc** and **esp_client** utilities send directives to Event Stream Processor via this process. See the *Command and Control Executables* topic for more details.

SQL Syntax in Event Stream Processor

The ESP On-Demand Query interface allows you to query active windows in a running ESP project using the esp_query tool.

These SQL queries can be run at any time, are ad hoc (for example, you do not need to define them in advance), and they return a snapshot based on the contents of the window at the time of the query. This is an important distinction: SQL queries submitted via the On-Demand Query interface are snapshot queries, unlike CCL queries which are continuous queries.

The On-Demand Query interface is suited for loading data slowly by running infrequent ad-hoc queries. It is not suited for inserting large amounts of data since it does not sustain high data throughput. As a result, it does not support high performance insertion techniques such as batch insertion. Instead, use one of the Publisher APIs (C, Java, or .NET) for high performance insertion.

Note that you can query input and output windows in a running ESP project. Streams and delta streams cannot be queried in on-demand queries since they do not have state, and local windows cannot be queried since they are internal to a project and not visible outside the project.

Use **esp_query** to query an ESP project as well as the SQL syntax that is supported for on-demand queries.

In order to make SQL queries to a model from your application, you must retrieve and use the ESP JDBC driver. The ESP JDBC driver allows your java client code to make SQL queries through the On-Demand SQL Interface.

ESP JDBC Driver Retrieval

In order to use the ESP JDBC driver, you must connect to a cluster, retrieve a session ID from ESP, and use it as the argument for the JDBC **user** connection parameter. The code sample below demonstrates this procedure.

```
// Create a URI and credentials to connect to the cluster.
Uri serverURI = new Uri.Builder("esp://nstackxp2:9786/default/
csv_poll_test").create();
Credentials creds = new
Credentials.Builder(Credentials.Type.NONE).create();

System.out.println(serverURI);

// Connect to the cluster
Project project = s_sdk.getProject(serverURI, creds);
project.connect(WAIT_TIME_60000_MS);

// Get the session ID after connecting.
String theSessionID = SDK.getSessionId(serverURI, creds);
System.out.println("The Session ID: " + theSessionID);

// Query the cluster for the project host and port.
Deployment dep = project.getCurrentDeployment();
int sqlPort = dep.getSqlPort();
String host = dep.getHost();
System.out.println("SqlPort = " + sqlPort);

//Server server = s_sdk.getServer(serverURI , creds);
//server.connect();

Jdbc3SimpleDataSource source = new Jdbc3SimpleDataSource();
source.setServerName("localhost");
source.setDatabaseName("database");
source.setUser(theSessionID);
source.setPassword("");
source.setPortNumber(sqlPort);

Connection con = null;
try {
    con = source.getConnection();
    // Use the connection.
} catch (SQLException e) {
    // Log any errors.
}
```

Supported SQL Statements

Syntax for SQL statements supported for on-demand queries.

DELETE Statement

Deletes specified rows from an input window.

Syntax

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

Components

window	An input window.
whereclause	An expression resulting in a 1 for true, or a 0 for false.

Usage

The **DELETE** statement removes specified rows from an INPUT window. It cannot be used with streams. The **WHERE** clause acts a filter which limits which rows are deleted. If a **WHERE** clause is not provided all rows in the input window are deleted.

Note that you cannot use esp_query to run concurrent **DELETE** statements on the same window or stream.

Example

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

INSERT Statement

Creates one or more rows, and inserts them into a specified stream or window.

Syntax

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

Components

StreamWindow	The name of an input stream or window.
column	The name of a column in a specified stream or window.
value	The corresponding value to insert. There must be as many values as there are specified columns.

Usage

The **INSERT** statement produces rows of data and publishes them to a destination stream or window. Rows are published to the specified destination any time the statement executes and produces output. To control when the statement executes, an optional **OUTPUT** clause can be added.

The **INSERT** statement explicitly indicates the columns to which data should be published, from left to right. The number and data types of the columns specified in the **INSERT INTO** clause must correspond to the number of items and data types of the column expressions in the **VALUES** clause.

If the destination stream or window includes more columns than you specify, unlisted columns are set to NULL when rows are published.

Note that you cannot use esp_query to run concurrent **INSERT** statements on the same window or stream.

Example

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

SELECT Statement

Queries the contents of an input or output window. This syntax is for querying the contents of a running project. It is not valid syntax for building a project.

A **SELECT** statement must include at least one **SELECT** clause and at least one **FROM** clause, but may contain other optional clauses, such as **WHERE**, **GROUP BY**, and **ORDER BY** clauses.

Syntax

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

Components

N	The number of rows.
expression	A SQL-92 expression specifying selection, grouping, or filtering conditions, as appropriate. See Usage for more information.
out_window	The name of an input/output window. Identify any submodules in the path to the module containing the window, by name and separated by dots. For example: module1.out_window1.
column	The name of the column the **GROUP BY** clause will use to organize the output.

Usage

The **SELECT** statement queries the current contents of INPUT/OUTPUT windows listed in the **FROM** clause and generates result rows, each of which has a fixed number of columns. Rows from the INPUT/OUTPUT window listed in the **FROM** clause are passed to the **SELECT** clause, after being filtered by the query's **WHERE** clause, if one is specified. These results are processed by any other clauses in the query.

The **SELECT** statement contains the following clauses:

- **SELECT** clause
- **FROM** clause
- (optional) **WHERE** clause
- (optional) **GROUP BY** clause

SELECT Clause

Every **SELECT** statement must contain one **SELECT** clause. The **SELECT** clause specifies a select-list, which is then used to generate results in a query. A select-list has a number of different features, including:

- The number of items in the simple **SELECT** statement's select-lists determines the number of columns in the result.
- A select-list expression can refer to column names of the OUTPUT windows specified in the **SELECT** statement's **FROM** clause.
- The asterisk (*) character can be used to select all columns from the INPUT/OUTPUT window specified in the **FROM** clause.
- The select-list can be prefaced by a DISTINCT keyword, which makes each row included in the result unique. If two or more rows contain the same values in all queried columns, the DISTINCT keyword causes only one of the rows to be included in the result. Otherwise, all rows that fit the criteria will be included in the results.

You can also include a `rowtime` or `rowid` in the **SELECT** clause to display the rowtime and the rowid in the output, if required.

FROM Clause

The **FROM** clause specifies the data source for the **SELECT** statement's query. The data source must be an OUTPUT window. The OUTPUT window can be referred to by its defined name from a **CREATE WINDOW** statement.

WHERE Clause

The optional **WHERE** clause can be used to specify selection conditions. As a selection condition, the **WHERE** clause filters rows from the data source before they are passed to the SELECT clause. Expressions in the **WHERE** clause cannot have aggregate functions.

GROUP BY Clause

The optional **GROUP BY** clause causes one or more rows of the result to be combined into a single row of output. A **GROUP BY** clause is often used when the query result contains aggregate functions. The clause can contain expressions using constants or expressions from the input window or stream, but it cannot contain expressions that have aggregate functions.

Examples

This example selects all trades with the symbol 'SAP' from Trades:

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

This example selects all trades with the symbol 'SAP', groups them by symbol then by price:

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

UPDATE Statement

Updates existing rows in an input window.

Syntax

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

Components

window	An input window.
whereclause	An expression resulting in 1 for true, or 0 for false.
value	An expression that evaluates to a value of the same data type as the specified column.
column	The name of a destination column to which value is published.

Usage

The **UPDATE** statement updates existing rows in an INPUT window. It cannot be used with streams. The **UPDATE** statement updates the rows identified by the **WHERE** clause if one is provided or updates all the rows if one is not provided. Any columns that are not specified in the **UPDATE** statement are automatically set to NULL.

Note that you cannot use esp_query to run concurrent **UPDATE** statements on the same window or stream.

Example

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

Supported SQL92 Expressions

Supported syntax for SQL references to expression.

SQL references to expression refer to the following syntax, unless indicated otherwise:

```
{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}|
```

Binary

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

Unary

```
- | + | NOT
```

esp_query

Executes an SQL statement, supplied by standard input or the `-Q` option, on the server and prints the results to the standard output.

Syntax

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

Required Arguments

- **`-c` *user[:password]* –** authenticates access to an ESP server with a user ID and, optionally, a *password*. If you do not provide either the *password* or the **-k** or **-G** option, you are prompted for the password. If the Event Stream Processor successfully authenticates with these credentials, the connection is maintained, otherwise the Event Stream Processor immediately closes the connection.
- **`-p` *host:port/workspace/project* –** Together, the *host:port/workspace/project* arguments specify the URI to connect to the ESP Server (cluster manager). For example, if you have started your ESP cluster server on port **19011** of your local machine, and you are running a project called **prj1** in the default workspace, specify `-p` as: `-p localhost:19011/default/prj1`

Options

- **`-e` –** Specifies that an encrypted SSL connection to the Event Stream Processor should be used.
- **`-G` –** (optional) authenticates access to the Event Stream Processor with credentials held within a Kerberos authentication ticket. Environment variables determine where the system will look to find authentication tickets (See *Administrators Guide > Security in the Event Stream Processor > Authentication > Setting Environment Variables for Kerberos* for more information). If the user name differs from the default principal name in the ticket cache, specify an alternate user name with the **–c** option to use the corresponding authentication ticket.

- **`-h` –** Prints a list of possible options on the screen along with a brief explanation for each option.
- **`-k` <*path*> –** Performs authentication by using the RSA private key file mechanism instead of password authentication. The *privateRsaKeyFile* must specify the pathname of the private RSA key file. Ensure that the ESP Server has been started with the `-k` option specifying the directory in which to store the RSA keys.

  Note: With this option enabled, the user name must be specified with the `-c` option, but the password is not required.
- **`-m` <*date format*> –** Sets the format for date values using `strptime` format. The default is "%Y-%m-%d %H:%M:%S+00".
- **`-P` <*precision*> –** Specifies the number of decimal places in output (default 2).
- **`-Q` <*query*> –** Specifies the query to execute (if not provided via stdin).
- **`-q` *hostname:port* –** Specifies the port, or both the hostname and port, of the SQL Listener of the target ESP Server. The default host is "localhost" and the default port is "22200".
- **`-t` <*table name*> –** Specifies the name of the table for the XML output. The default name is "Result".
- **`-v` –** Prints the **esp_query** utility version.

Usage

esp_query accepts an SQL query on the standard input and forwards it to a running instance of the Event Stream Processor. It then prints the results of the query on the standard output.

From a UNIX or Linux command line prompt, or the Query panel in the ESP Studio, the SQL query must be enclosed in double quotes. From a DOS command line prompt, double quotes must not enclose the SQL query.

Querying streams doesn't make sense, but the **esp_query** can be used to obtain information from an error stream if a downstream window is defined to retain the state of the error stream.

Examples

To print the contents of stream *Emp*, on a UNIX machine named myhost, using SQL port 11100:

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

If myhost is a Windows machine, the **esp_query** syntax is the same, but the query must not be in quotes:

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

To delete an entry from the *Dept* stream and update the *Emp* stream accordingly:

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

To query a window, named ErrorState, that retains the state of an error stream named ErrorStream:

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

SQL Syntax

The Event Stream Processor accepts a subset of SQL92 statements.

The SQL92 statements that Event Stream Processor accepts are **select**, **insert**, **update**, and **delete** statements.

Queries using **select** are limited to single streams, with no joins or subqueries, but may use **where**, **group by**, and **order by** clauses. The **insert**, **update**, and **delete** statements are restricted to source streams. You can put these modification statements in sequence with a semicolon. You cannot run **insert**, **update**, or **delete** statements concurrently on the same window or stream.

Index