pubsubplus-connector-spark

User Guide

Solace Corporation

Version 3.1.7

solace.

Table of Contents

Preface . . . 1
Getting Started . . . 2
- Prerequisites . . . 2
- Supported Platforms . . . 2
- Quick Start common steps . . . 2
- Databricks Considerations . . . 9
- Handling Idle Connections . . . 9
- Checkpointing & Acknowledgement . . . 9
- Checkpoint Handling . . . 9
  - Prerequisites for LVQ creation . . . 10
- Checkpoint Storage . . . 10
- User Authentication . . . 11
  - Using Databricks Secret Management . . . 12
- Message Replay . . . 13
- Parallel Processing . . . 13
- Solace Spark Streaming Source Schema Structure . . . 13
- Solace Spark Streaming Sink Schema Structure . . . 14

Configuration . . . 15
- Solace Spark Connector Source Configuration Options . . . 15
- Solace Spark Connector Sink Configuration Options . . . 26

License . . . 33
- Support . . . 33

Preface

Solace PubSub+ Connector for Spark streams data from Solace PubSub+ broker to Spark Data Sources.

Getting Started

This guide assumes you are familiar with Spark set up and Spark Structured Streaming concepts. In the following sections we will show how to set up Solace Spark Connector to stream data from Solace to Spark and publish events from Spark to Solace.

Prerequisites

- Solace PubSub+ Event Broker
- Apache Spark 3.5.2 and Scala 2.12

Supported Platforms

The connector is built on the Spark Structured Streaming API and has been tested on Azure Databricks(15.4 LTS (includes Apache Spark 3.5.0, Scala 2.12) with photon acceleration disabled and 16.4 LTS(includes Apache Spark 3.5.2, Scala 2.12)). Since the Databricks runtime is consistent across all supported cloud platforms(AWS & Google Cloud), it is expected to behave similarly in other Databricks environments. Additionally, the connector has been validated on vanilla Apache Spark, ensuring compatibility with any platform that supports standard Spark deployments.

Quick Start common steps

This quick start shows how to deploy connector in Databricks Cluster.

Solace Spark Connector is available in maven central with below coordinates.

```
<dependency>
    <groupId>com.solacecoe.connectors</groupId>
    <artifactId>pubsubplus-connector-spark</artifactId>
    <version>3.1.7</version>
</dependency>
```

Steps to deploy the connector

1. Create a Databricks Cluster with Spark Version 3.5.2 and Scala 2.12
2. Once the cluster is started choose libraries and select maven as connector source. Enter above maven coordinates to download connector from maven central.

Before installing latest version of connector make sure earlier versions of solace spark connector are completely deleted from cluster. This is to ensure there are no version conflicts to start the connector.

1. Create new scala notebook and copy below code snippets.
2. Creating Solace Source Streaming Queries

a. First, let's create a Solace consumer to read messages and write to parquet file. Update configuration options as per your environment.

   i. Scala Code Snippet

```
val spark = SparkSession.builder.appName("SolaceSparkStreaming").
getOrCreate()
val streamName = "solace-spark-connector-sample-test"
val df = spark.readStream.format("solace")
    .option("host", "tcp://localhost:55555")
    .option("vpn", "default")
    .option("username", "default")
    .option("password", "default")
    .option("queue", "<queue-name>")
    .option("connectRetries", 2)
    .option("reconnectRetries", 2)
    .option("batchSize", 100) ⑤
    .option("includeHeaders", true)
    .option("partitions", 1)
    .load()

// write to parquet file
val query = df.writeStream
    .format("parquet")
    .outputMode("append")
    .queryName(streamName)
    .option("checkpointLocation",
s"/spark_checkpoints/solace_spark_connector_checkpoint/$streamName/")
    .option("path", s"/solace_spark_connector_parquet_output/$streamName/")
    .start()

query.awaitTermination()
```

   ii. Python Code Snippet

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("SolaceSparkStreaming") \
    .getOrCreate()

stream_name = "solace-spark-connector-sample-test"

df = spark.readStream \
    .format("solace") \
    .option("host", "tcp://localhost:55555") \
    .option("vpn", "default") \
    .option("username", "default") \
    .option("password", "default") \
    .option("queue", "<queue-name>") \
```

```
    .option("connectRetries", 2) \
    .option("reconnectRetries", 2) \
    .option("batchSize", 100) \
    .option("includeHeaders", "true") \
    .option("partitions", 1) \
    .load()

query = df.writeStream \
    .format("parquet") \
    .outputMode("append") \
    .queryName(stream_name) \
    .option("checkpointLocation",
f"/spark_checkpoints/solace_spark_connector_checkpoint/{stream_name}/") \
    .option("path", f"/solace_spark_connector_parquet_output/{stream_name}
/") \
    .start()

query.awaitTermination()
```

iii. Java Code Snippet

```
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import org.apache.spark.sql.streaming.StreamingQuery;
import org.apache.spark.sql.streaming.StreamingQueryException;

public class SolaceSparkStreamingJava {
    public static void main(String[] args) throws StreamingQueryException {
        SparkSession spark = SparkSession.builder()
                .master("local[*]")
                .appName("SolaceSparkStreaming")
                .getOrCreate();

        String streamName = "solace-spark-connector-sample-test";

        Dataset<Row> df = spark.readStream()
                .format("solace")
                .option("host", "tcp://localhost:55555")
                .option("vpn", "default")
                .option("username", "default")
                .option("password", "default")
                .option("queue", "<queue-name>")
                .option("connectRetries", 2)
                .option("reconnectRetries", 2)
                .option("batchSize", 100)
                .option("includeHeaders", true)
                .option("partitions", 1)
                .load();
```

```
        StreamingQuery query = df.writeStream()
                .format("parquet")
                .outputMode("append")
                .queryName(streamName)
                .option("checkpointLocation",
"/spark_checkpoints/solace_spark_connector_checkpoint/" + streamName + "/")
                .option("path", "/solace_spark_connector_parquet_output/" +
streamName + "/")
                .start();

        query.awaitTermination();
    }
}
```

For optimal throughput, configure the Solace queue's 'Maximum Delivered Unacknowledged Messages per Flow' property to a value equal to twice the batch size.

a. Finally, let's read data from the parquet file from the location configured above

i. Scala Code Snippet

```
val streamName = "solace-spark-connector-sample-test" // this should be equal to
stream name variable provided as above
val df = spark.read.format("parquet").load(
s"/solace_spark_connector_parquet_output/$streamName/") // this should be same
as value of "path" property configured in write stream as above
display(df)
// Ex: Parse payload as string
df.select($"payload".cast("STRING"))
```

ii. Python Code Snippet

```
from pyspark.sql.functions import col

stream_name = "solace-spark-connector-sample-test"

df = spark.read \
    .format("parquet") \
    .load(f"/solace_spark_connector_parquet_output/{stream_name}/")

# If using Databricks or notebook environment that supports display()
display(df)

# Parse 'payload' column as STRING
df_parsed = df.select(col("payload").cast("string"))

# Show the parsed payload
```

```
df_parsed.show()
```

iii. Java Code Snippet

```
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import static org.apache.spark.sql.functions.col;

public class ReadParquetAndParsePayload {
    public static void main(String[] args) {
        SparkSession spark = SparkSession.builder()
                .master("local[*]")
                .appName("SolaceReadParquet")
                .getOrCreate();

        String streamName = "solace-spark-connector-sample-test";

        Dataset<Row> df = spark.read()
                .format("parquet")
                .load("/solace_spark_connector_parquet_output/" + streamName +
"/");

        df.show();  // Equivalent of display()

        Dataset<Row> dfParsed = df.select(col("payload").cast("string"));

        dfParsed.show();  // Show the casted payload
    }
}
```

1. Creating Solace Sink Streaming Queries

b. First, let's create a parquet consumer to read messages and publish to Solace. Update configuration options as per your environment.

c. Next, create a queue on Solace Broker and subscribe to the topic published by Solace Spark Connector. In this case it is `solace/spark/publish`

i. Scala Code Snippet

```
val streamName = "solace-spark-connector-sample-test"
val parquetData = spark.read.parquet("<path-to-parquet-file>")
val struct_stream = spark.readStream
.schema(parquetData.schema)
.parquet("<path-to-parquet-file>")

// write to parquet file
val query = df.writeStream
      .format("solace")
```

```
      .option("host", "tcp://localhost:55555")
      .option("vpn", "default")
      .option("username", "default")
      .option("password", "default")
      .option("topic", "solace/spark/stream/processing/result") // This can be
commented if topic column is present in output dataframe.
      .option("id", "<application-message-id>")
      .outputMode("append")
      .option("checkpointLocation",
"/spark_checkpoints/solace_spark_connector_checkpoint/" + streamName + "/")
      .start()

query.awaitTermination()
```

ii. Python Code Snippet

```
from pyspark.sql import SparkSession

stream_name = "solace-spark-connector-sample-test"
path_to_parquet = "<path-to-parquet-file>"

spark = SparkSession.builder.appName("SolaceSparkStreaming").getOrCreate()

# Read static data to infer schema
parquet_data = spark.read.parquet(path_to_parquet)

# Read the streaming source
struct_stream = spark.readStream.schema(parquet_data.schema).parquet
(path_to_parquet)

# Write to Solace
query = struct_stream.writeStream \
    .format("solace") \
    .option("host", "tcp://localhost:55555") \
    .option("vpn", "default") \
    .option("username", "default") \
    .option("password", "default") \
    .option("topic", "solace/spark/stream/processing/result") \ # This can be
commented if topic column is present in output dataframe.
    .option("id", "<application-message-id>") \ # This can be commented if Id
column is present in output dataframe.
    .outputMode("append") \
    .option("checkpointLocation",
f"/spark_checkpoints/solace_spark_connector_checkpoint/{stream_name}/") \
    .start()

query.awaitTermination()
```

iii. Java Code Snippet

```
import org.apache.spark.sql.SparkSession;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.streaming.StreamingQuery;

public class SolaceSparkStreaming {
    public static void main(String[] args) throws Exception {
        String streamName = "solace-spark-connector-sample-test";
        String parquetPath = "<path-to-parquet-file>";

        SparkSession spark = SparkSession.builder()
                .master("local[*]")
                .appName("SolaceSparkStreaming")
                .getOrCreate();

        // Read static schema
        Dataset<Row> parquetData = spark.read().parquet(parquetPath);

        // Read stream using inferred schema
        Dataset<Row> structStream = spark.readStream()
                .schema(parquetData.schema())
                .parquet(parquetPath);

        // Write to Solace
        StreamingQuery query = structStream.writeStream()
                .format("solace")
                .option("host", "tcp://localhost:55555")
                .option("vpn", "default")
                .option("username", "default")
                .option("password", "default")
                .option("topic", "solace/spark/stream/processing/result") //
This can be commented if topic column is present in output dataframe.
                .option("id", "<application-message-id>") // This can be
commented if Id column is present in output dataframe.
                .outputMode("append")
                .option("checkpointLocation",
"/spark_checkpoints/solace_spark_connector_checkpoint/" + streamName + "/")
                .start();

        query.awaitTermination();
    }
}
```

Above sample code used parquet as example data source. You can configure your required data source to write data.

In case of databricks deployment, it is recommended to store and retrieve sensitive credentials from Databricks secrets. Please refer to Using Databricks

Secret Management on how to configure secrets and use them in notebook.

Solace Partitioned Queue's is not supported.

Databricks Considerations

In case if you are using Shared compute cluster, make sure your cluster has appropriate permissions to install connector from maven central and access the jars. Please contact your Databricks administrator for required permissions.

Handling Idle Connections

The connector provides the ability to terminate idle connections when there is no data available in queue for processing or if spark session is not gracefully terminated.

To manage this, it is recommended to configure the following parameters:

1. Idle Connection Timeout in Milliseconds: Specifies the maximum duration a connection can remain idle before being considered for closure.
2. Idle Connection Timeout Check in Milliseconds: Defines the interval at which idle connections are checked. The Solace Spark Connector monitors the time since the last processed message. If this duration exceeds the configured idle timeout, the connector closes the idle connection.

Both configurations are disabled by default and must be explicitly set to enable this feature. Any pending messages to be acknowledged will be redelivered and the downstream should be idempotent to handle duplicates. The connection will be re-established in case of restart(job or notebook) or if new messages arrive in the queue(job or notebook should be up and running).

Checkpointing & Acknowledgement

Solace Spark connector relies on Spark Checkpointing mechanism to resume from last successful message. Connector acknowledges & writes processed message id's to spark checkpoint whenever Spark sends commit signal to connector. Commit signal is an indication by Spark that data has been processed successfully.

Checkpoint Handling

Starting from version 3.1.0 connector, solace connection is now executed on worker node instead of driver node. This gives us the ability to utilize cluster resource efficiently and also improves processing performance. The connector uses Solace LVQ to store checkpoint along with Spark Checkpoint.

In case of recovery, connector uses offset state from LVQ to identify last successfully processed messages. Hence, it is recommended not to delete or modify offset state in LVQ. In some cases, there might be checkpoint failures as spark may

fail to write to checkpoint during instance crash or unavailability or other reasons. Though the connector will handle duplicates in most cases, we recommend to keep your downstream systems idempotent.

Prerequisites for LVQ creation

The following prerequisites are applicable for LVQ that are provisioned by Solace Administrator.

1. The Queue should be of type Exclusive
2. Spool Quota should be set to 0
3. Owner of the Queue should be the client username used by the micro integration
4. Non-Owner access should be set to No Access to prevent unauthorized access
5. Add a topic subscription
6. Ensure the ACL applied to the user has publish and subscribe access to the topic subscribed by LVQ

The following pre-requisites are applicable if the micro integration need to create LVQ if it doesn't exist.

1. Ensure the Client Profile applied to the user has "Allow Client to Create Endpoints" enabled.
2. Ensure the ACL applied to the user has publish and subscribe access to the topic subscribed by LVQ. The Subscribe access is required to programmatically apply the topic subscription to LVQ.
3. The Micro Integration will create a LVQ with pre-requisites mentioned in above section.

Checkpoint Storage

The connector supports multiple checkpoint storage options depending on the deployment platform. When running on Databricks, Unity Catalog Volumes can be used as the checkpoint location. Since Databricks executor nodes do not have direct access to Unity Catalog Volumes, the connector uses unified authentication via a `Databricks workspace level service principal` to establish secure access. Ensure the Databricks workspace level service principal has access to Unity Catalog Volume configured as checkpoint storage and the secret scope.

To enable this, the following must be configured:

- Spark Runtime Platform — Set to DATABRICKS to enable Unity Catalog Volume support
- Secret Scope — A Databricks secret scope containing the workspace level service principal credentials (databricks_host, databricks_client_id, databricks_client_secret)
- Client Secret Refresh Interval — Controls how frequently the connector fetches updated client secret from the secret scope to handle client secret rotation without requiring a restart

Both client secret rotation and the connector's client secret refresh must complete before the secret expires. The refresh interval must be configured to trigger after rotation completes and before the existing secret expires. See the configuration section for detailed guidance on configuring the refresh interval correctly.

Refer to the Source Configuration section for a complete list of properties, accepted values for Databricks unified authentication.

User Authentication

Solace Spark Connector supports Basic, Client Certificate and OAuth authentication to Solace. Client Credentials flow is supported when connecting using OAuth.

If OAuth server is available use below options to fetch access token from endpoint. For property description please refer to Configuration section.

```
    spark.readStream.format("solace").option("host", "")
    .option("vpn", "default")
    .option("solace.apiProperties.AUTHENTICATION_SCHEME",
"AUTHENTICATION_SCHEME_OAUTH2")
    .option("solace.oauth.client.auth-server-url", "")
    .option("solace.oauth.client.client-id", "")
    .option("solace.oauth.client.credentials.client-secret", "")
    .option("solace.oauth.client.auth-server.client-certificate.file", "")
    .option("solace.oauth.client.auth-server.truststore.file", "")
    .option("solace.oauth.client.auth-server.truststore.password", "")
    .option("solace.oauth.client.auth-server.ssl.validate-certificate", false)
    .option("solace.oauth.client.token.refresh.interval", 110)
```

If rotating access token is present in file accessible by connector use below options to enable OAuth authentication to read access token from file. For property description please refer to Solace OAuth Client Refresh Interval property. In case when access token file is not updated, connector retries the connection based on Reconnect Retries and stops if authentication is not successful.

```
    spark.readStream.format("solace").option("host", "")
    .option("vpn", "")
    .option("solace.apiProperties.AUTHENTICATION_SCHEME",
"AUTHENTICATION_SCHEME_OAUTH2")
    .option("solace.oauth.client.access-token", "<absolute-path-to-token-file>")
    .option("solace.oauth.client.token.refresh.interval", 110)
```

When access token is read from file, it may lose some of it's expiry time by the time it is accessed by connector. It is recommended to have minimal time difference between writing to file and access by the connector so that a valid new token is updated in solace session before expiry of old token.

Below is an example on how to use client certificate authentication when connecting to Solace.

```
    sparkSession.readStream().format("solace")
    .option("host", "")
    .option("vpn", "default")
```

```
    .option("username", "")
    .option("solace.apiProperties.AUTHENTICATION_SCHEME",
"AUTHENTICATION_SCHEME_CLIENT_CERTIFICATE")
    .option("solace.apiProperties.SSL_TRUST_STORE", "<path-to-jks-file>")
    .option("solace.apiProperties.SSL_TRUST_STORE_FORMAT", "jks")
    .option("solace.apiProperties.SSL_TRUST_STORE_PASSWORD", "")
    .option("solace.apiProperties.SSL_KEY_STORE", "<path-to-jks-file>")
    .option("solace.apiProperties.SSL_KEY_STORE_FORMAT", "jks")
    .option("solace.apiProperties.SSL_KEY_STORE_PASSWORD", "")
```

For more properties please refer to Solace Java API documentation for com.solacesystems.jcsmp.JCSMPProperties

Using Databricks Secret Management

If Solace Spark Connector is deployed in Databricks, it is recommended to use Databricks secrets to store sensitive credentials.

To configure secrets refer to the Databricks documentation.

You can reference those secrets in your Spark cluster using the same Spark config options:

Below is an example on how to retrieve username and password from Databricks secrets and connect to Solace.

```
    spark.readStream.format("solace").option("host", dbutils.secrets.get(scope =
"solace-dev-credentials", key = "host"))
    .option("vpn", "default")
    .option("username", dbutils.secrets.get(scope = "solace-dev-credentials", key =
"username"))
    .option("password", dbutils.secrets.get(scope = "solace-dev-credentials", key =
"password"))
```

OAuth based authentication to Solace using Databricks secrets. The certificates can be stored in cloud object storage, and you can restrict access to the certificates only to cluster that can access Solace. See Data governance with Unity Catalog.

```
    spark.readStream.format("solace").option("host", dbutils.secrets.get(scope =
"solace-dev-credentials", key = "host"))
    .option("vpn", "default")
    .option("solace.apiProperties.AUTHENTICATION_SCHEME",
"AUTHENTICATION_SCHEME_OAUTH2")
    .option("solace.oauth.client.auth-server-url", "")
    .option("solace.oauth.client.client-id", dbutils.secrets.get(scope = "solace-dev-
credentials", key = "client-id"))
    .option("solace.oauth.client.credentials.client-secret", dbutils.secrets.get(scope
= "solace-dev-credentials", key = "client-secret"))
    .option("solace.oauth.client.auth-server.client-certificate.file", "")
```

```
    .option("solace.oauth.client.auth-server.truststore.file", "")
    .option("solace.oauth.client.auth-server.truststore.password", dbutils.secrets.
get(scope = "solace-dev-credentials", key = "truststore-password"))
    .option("solace.oauth.client.auth-server.ssl.validate-certificate", false)
    .option("solace.oauth.client.token.refresh.interval", 110)
```

Message Replay

Solace Spark Connector can replay messages using Solace Replay Log. Connector can replay all messages or after specific replication group message id or after specific timestamp. Please refer to Message Replay Configuration to enable replay log in Solace PubSub+ broker.

Parallel Processing

The Solace Spark Connector supports automatic scaling of consumers based on the number of worker nodes or can be configured to use a fixed number of consumers. To control this behavior, use the partition property in the Solace Spark Connector Source configuration. Setting this property to 0 enables automatic scaling, where the number of consumers matches the number of worker nodes.

Solace Spark Streaming Source Schema Structure

Solace Spark Connector transforms the incoming message to Spark row with below schema definition.

Column Name	Column Type	Description
Id	String	Represents replication group message id of incoming message.
Payload	Binary	Represents payload in binary format.
PartitionKey	String	Represents Partition Key if present in message.
Topic	String	Represents the topic on which message is published.
TimeStamp	Timestamp	Represents sender timestamp if present in message. By, default it returns the timestamp when message is received by connector.
Headers	Map<string, binary>	Represent message headers if present in message. This column is created only when includeHeaders option is set to true.

Solace Spark Streaming Sink Schema Structure

Solace Spark Connector transforms the incoming message to Spark row with below schema definition.

Column Name	Column Type	Description
Id	String	Set the message id for the published message. This will be overwritten if message id is set using the id option. If no message id is set connector will throw an exception as message id is required to track the state of published messages. In case of publish failure the message id along with exception is logged.
Payload	Binary	Payload to be added to the published message. If no payload is set connector will throw an exception.
PartitionKey(Optional)	String	Partition Key for the published message. Useful when published message topic is subscribed by partitioned queues.
Topic(Optional)	String	Set□s the topic for the published message. This will be overwritten if topic is set using the topic option. If no topic is set connector will throw an exception as topic is required to publish a message.
TimeStamp(Optional)	Timestamp	Set the timestamp for published message. This column is mapped to Sender Timestamp field in Solace Message.
Headers(Optional)	Map<string, binary>	Set the headers to be added in published message. This column is mapped only when includeHeaders option is set to true.

Configuration

Solace Spark Connector Source Configuration Options

Config Option	Type	Valid Values	Default Value	Description
host	string	any		Fully Qualified Solace Hostname with protocol and port number.
vpn	string	any		Solace VPN name.
username	string	any		Solace Client Username.
password	string	any		Solace Client Password.
connectRetries	int	(-1) or greater	0	Number of times to retry connection during initial setup. 0 means no retries. -1 means retry forever.
reconnectRetries	int	(-1) or greater	3	Number of times to attempt reconnection after a disconnect. 0 means no retries. -1 means retry forever.
connectRetriesPerHost	int	(-1) or greater	0	When using a host list, defines how many times to try connecting to a single host before moving to the next. 0 means one attempt. -1 means infinite retries on the first host only.
reconnectRetryWaitInMillis	int	0 - 60000	3000	Time in milliseconds to wait between each connection or reconnection attempt.
connectIdleTimeoutInMillis	int	Timeout in Milliseconds	0	Time in milliseconds before closing an idle connection to Solace. Useful in long-running clusters where Spark sessions do not terminate gracefully.

Config Option	Type	Valid Values	Default Value	Description
connectIdleTimeoutCheckInMillis	int	Timeout in Milliseconds	0	Interval in milliseconds at which to check for idle connections. Connections idle beyond the configured timeout are closed automatically.
solace.apiProperties.<Property>	any	any		Additional Solace Java API properties can be set using solace.apiProperties.<Property> where <Property> is defined in the Solace Java API documentation. Example: solace.apiProperties.reapply_subscriptions=false solace.apiProperties.client_channel_properties.keepAliveIntervalInMillis=3000
solace.oauth.client.access-token	string	absolute file path	empty	Absolute path to a file containing a rotating access token. The connector reads the token directly from the file. Requires solace.oauth.client.token.refresh.interval to be configured.
solace.oauth.client.auth-server-url	string	any	empty	Full URL of the token endpoint used to fetch the access token.
solace.oauth.client.client-id	string	any	empty	OAuth Client ID.
solace.oauth.client.credentials.client-secret	string	any	empty	OAuth Client Secret.
solace.oauth.client.auth-server.client-certificate.file	string	any	empty	Absolute path to the X.509 client certificate file for TLS connections.

Config Option	Type	Valid Values	Default Value	Description
`solace.oauth.client.auth-server.truststore.file`	`string`	`any`	`empty`	Absolute path to the trust store file for TLS connections. • If a JKS file is available, provide the absolute path to load it directly. • If `solace.oauth.client.auth-server.client-certificate.file` is configured, provide a path including the file name. The connector will load the client certificate into a keystore and save it as a JKS file.
`solace.oauth.client.auth-server.truststore.password`	`string`	`any`	`empty`	Password for the JKS trust store file. • If a JKS file is available, provide its password. • If `solace.oauth.client.auth-server.client-certificate.file` is configured, provide a password to protect the generated JKS file.
`solace.oauth.client.auth-server.ssl.validate-certificate`	`boolean`	`true or false`	`true`	Enables or disables SSL certificate validation. If set to `false`, TLS requests are sent without validation.
`solace.oauth.client.auth-server.tls.version`	`string`	`SSL, TLS, TLSv1, TLSv1.1, TLSv1.2, TLSv1.3`	`TLSv1.2`	TLS protocol version for the SSL connection.

Config Option	Type	Valid Values	Default Value	Description
solace.oauth.client.token.refresh.interval	int	positive integer	60	Interval in seconds at which the connector fetches a new access token. Configure this value to be less than your token expiry time to avoid disconnections.
solace.oauth.client.token.fetch.timeout	int	positive integer	100	Connection timeout in milliseconds for access token requests.
queue	string	any		Solace Queue name.
queue.receiveWaitTimeout	int	Timeout in milliseconds	10000	Maximum time to wait before submitting available messages to a Spark micro-batch.
batchSize	int	any	1	Number of messages to process per batch. For optimal throughput, configure the Solace queue's Maximum Delivered Unacknowledged Messages per Flow to twice the batch size.
replayStrategy	string	BEGINNING | TIMEBASED | REPLICATION-GROUP-MESSAGE-ID	empty	Replay strategy for replaying messages from the broker. See Solace Replay Configuration.
replayReplicationGroupMessageId	string	valid replication group message id	empty	Required when replayStrategy is REPLICATION-GROUP-MESSAGE-ID. Message playback starts after this replication group message ID.
replayStartTime	string	yyyy-MM-dd□T□HH:mm:ss	empty	Required when replayStrategy is TIMEBASED. Messages equal to or newer than this date and time are replayed. Cannot be earlier than the replay log creation date.

Config Option	Type	Valid Values	Default Value	Description
`replayStartTimeTimezone`	`string`	`valid timezone`	`UTC`	Timezone for `replayStartTime`. Required when `replayStrategy` is `TIMEBASED`.
`ackLastProcessedMessages`	`boolean`	`true or false`	`false`	When set to `true`, the connector identifies and acknowledges messages processed in the last run during restart. Depends on the Spark checkpoint. Enable only when the downstream system has successfully processed data in the previous run. This property has no effect when a replay strategy is configured.
`includeHeaders`	`boolean`	`true or false`	`false`	When set to `true`, message headers are included in the output.

Config Option	Type	Valid Values	Default Value	Description
`partitions`	`int`	`any`	`1`	Number of consumers for the configured queue. Consumers are distributed across worker nodes. Set to `0` to create one consumer per worker node, scaling automatically as nodes are added.
`lvq.name`	`string`	`valid lvq name`	`empty`	Name of the LVQ configured on the Solace broker. Used to communicate checkpoint information from worker nodes to the driver node and ensures checkpoint data is present in both the LVQ and Spark checkpoint directory.
`lvq.topic`	`string`	`valid solace topic`	`empty`	Topic subscribed by the LVQ on the Solace broker. Each worker node publishes checkpoint information to this topic.

Config Option	Type	Valid Values	Default Value	Description
`ignoreCheckpointMessageIdComparisonError`	`boolean`	`true or false`	`false`	Specifies whether to ignore errors when comparing an incoming message's Replication Group Message ID against the checkpointed message ID during restart. • `false` — throws `SolaceReplicationGroupMessageIdNotComparableException` and stops processing. • `true` — logs a warning, skips the comparison, and continues. The message is treated as new and will not be deduplicated. This error typically occurs when messages originate from different replication groups.

Config Option	Type	Valid Values	Default Value	Description
sparkRuntimePlatform	string	DATABRICKS, OTHER	DATABRICKS	Runtime platform on which the Spark application is deployed. Required when a Databricks Unity Catalog Volume is used as the checkpoint location. • DATABRICKS — enables Unity Catalog Volume support for checkpointing. • OTHER — disables the Unity Catalog Volume check.
databricksSecretScope	string	valid secret scope name	empty	Databricks secret scope used to retrieve authentication credentials for accessing Unity Catalog Volumes as the checkpoint location.
databricksHost	string	valid secret name	empty	Secret name within the configured secret scope that holds the Databricks workspace host URL. Required when spark_runtime_platform is DATABRICKS and the checkpoint path contains Volumes. Refer to Databricks OAuth Workspace level Environment Variables for valid host value

Config Option	Type	Valid Values	Default Value	Description
`databricksClientId`	`string`	`valid secret name`	`empty`	Secret name within the configured secret scope that holds the Databricks service principal client ID. Required when `spark_runtime_platform` is `DATABRICKS` and the checkpoint path contains `Volumes`. For relevant value refer to Databricks OAuth Workspace level Environment Variables for valid client id value Ensure the service principal has access to the configured Unity Catalog Volume checkpoint path and Databricks secret scope.

Config Option	Type	Valid Values	Default Value	Description
databricksClientSecret	string	valid secret name	empty	Secret name within the configured secret scope that holds the Databricks service principal client secret. Required when spark_runtime_platform is DATABRICKS and the checkpoint path is Unity Catalog Volume. For relevant values refer to Databricks OAuth Workspace level Environment Variables for valid client secret value See [databricks-client-secret-notes].

Config Option	Type	Valid Values	Default Value	Description
`databricksClientSecretRefreshInterval`	`long`	`positive integer (minutes)`	`1440`	Interval in minutes at which the connector fetches the latest client secret from the configured Databricks secret scope. The user is responsible for ensuring: • Rotation completes before the next refresh interval triggers. • Both rotation and refresh complete before the existing secret expires. If the refresh interval triggers before rotation is complete, the connector fetches the old secret and continues using it until the next refresh cycle.

Databricks Client Secret — Important Notes

Always manage this credential using Databricks secret scopes. Never expose it in plain text.

Secret rotation is not managed by the connector. The platform administrator is responsible for rotating the secret and updating it in the secret scope. The connector fetches the updated secret automatically based on the interval

configured in `databricks_client_secret_refresh_interval`. No restart is required.

Solace Spark Connector Sink Configuration Options

Config Option	Type	Valid Values	Default Value	Description
`host`	`string`	`any`		Fully Qualified Solace Hostname with protocol and port number.
`vpn`	`string`	`any`		Solace VPN name.
`username`	`String`	`any`		Solace Client Username.
`password`	`string`	`any`		Solace Client Username password.
`connectRetries`	`int`	`(-1) or greater`	`0`	The number of times to attempt and retry a connection during initial connection setup. Zero means no automatic connection retries (that is, try once and give up). -1 means "retry forever".
`reconnectRetries`	`int`	`(-1) or greater`	`3`	The number of times to attempt to reconnect. Zero means no automatic reconnection retries (that is, try once and give up). -1 means "retry forever".

Config Option	Type	Valid Values	Default Value	Description
connectRetriesPerHost	int	(-1) or greater	0	When using a host list for the HOST property, this property defines how many times to try to connect or reconnect to a single host before moving to the next host in the list. NOTE: This property works in conjunction with the connect and reconnect retries settings; it does not replace them. Valid values are >= -1. 0 means make a single connection attempt (that is, 0 retries). -1 means attempt an infinite number of reconnect retries (that is, the API only tries to connect or reconnect to first host listed.)
reconnectRetryWaitInMillis	int	0 - 60000	3000	How much time in (MS) to wait between each attempt to connect or reconnect to a host. If a connect or reconnect attempt to host is not successful, the API waits for the amount of time set for reconnectRetryWaitInMillis, and then makes another connect or reconnect attempt.

Config Option	Type	Valid Values	Default Value	Description
solace.apiProperties.<Property>	any	any		Any additional Solace Java API properties can be set through configuring solace.apiProperties.<Property> where <Property> is the name of the property as defined in the Solace Java API documentation for com.solacesystems.jcsmp.JCSMPProperties, for example: `solace.apiProperties.pub_ack_window_size=50`
solace.oauth.client.access-token	string	absolute file path to token file	empty	Set this configuration, if rotating access token is present in file. In this case connector will read access token directly from file instead of sending request to OAuth Server. Please note Solace OAuth Client Refresh Interval should be set to read access token from file at regular intervals.
solace.oauth.client.auth-server-url	string	any	empty	Full representation of token endpoint to fetch access token.
solace.oauth.client.client-id	string	any	empty	OAuth Client ID
solace.oauth.client.credentials.client-secret	string	any	empty	OAuth Client Secret
solace.oauth.client.auth-server.client-certificate.file	string	any	empty	Absolute path to X.509 client certificate file for TLS connections. Make sure file path is accessible by the connector.

Config Option	Type	Valid Values	Default Value	Description
solace.oauth.client.auth-server.truststore.file	string	any	empty	Absolute path to trust store file for TLS connections. This property works in two ways 1. If JKS file is available in cluster configure absolute path so that connector will load the JKS file. 2. If solace.oauth.client.auth-server.client-certificate.file is configured simply provide a path(should include file name as well). The connector will load the client certificate to key store and saves to JKS file .
solace.oauth.client.auth-server.truststore.password	string	any	empty	Password for JKS file. This property works in two ways 1. If JKS file is available in cluster provide the password to JKS file. 2. If solace.oauth.client.auth-server.client-certificate.file is configured simply provide a password which will be used to protect the JKS file created in above configuration option 2.
solace.oauth.client.auth-server.ssl.validate-certificate	boolean	any	true	Boolean value to enable or disable ssl certificate validation. If set to false connector will send TLS request without any validation.

Config Option	Type	Valid Values	Default Value	Description
solace.oauth.client.auth-server.tls.version	string	SSL, TLS, TLSv1, TLSv1.1, TLSv1.2, TLSv1.3	TLSv1.2	Indicate the type of SSL connection.
solace.oauth.client.token.refresh.interval	integer	positive integer value	60	Interval(Seconds) to fetch access token by the connector to avoid disconnection on token expiry. This value should be less than your token expiry time.
solace.oauth.client.token.fetch.timeout	integer	positive integer value	100	Connection timeout(MS) for access token request.
topic	string	any		Sets the topic that all rows will be published to Solace. This option overrides any topic column that may exist in the data.
id	string	any		**Deprecated** - Sets the message id to all the messages published to Solace. This option overrides any Id column that may exist in the data.
batchSize	int	any	1	Set number of messages to be processed in batch. This should be set to dataframe.count().
includeHeaders	boolean	true or false	false	Set this value to true if message headers in the row need to be added to published message.

Config Option	Type	Valid Values	Default Value	Description
`publishAckTimeout`	`long`	`timeout in milliseconds`	`5000`	Specifies the maximum time the connector waits for publish acknowledgements from the Solace broker after sending a batch of messages. If all acknowledgements are received before the timeout expires, the connector proceeds immediately without waiting for the full duration. If the timeout expires before all acknowledgements are received, the connector throws an exception if `publishAckTimeoutFailOnError` is set to true or logs a warning and proceeds to next batch. This value should be tuned based on the network latency between the connector and the Solace broker, the batch size configured via batch-size, and the expected broker response time under load.

Config Option	Type	Valid Values	Default Value	Description
`publishAckTimeoutFailOnError`	`boolean`	`true or false`	`true`	Specifies whether the connector should fail and stop processing when a publish acknowledgement timeout occurs. When set to false, the connector logs a warning and continues processing the next batch. When set to true, the connector throws a `SolacePublishAckTimeoutException` and stops processing, allowing Spark to handle the failure according to its configured retry or error handling policy.

License

This project is licensed under the Solace Community License, Version 1.0. - See the `LICENSE` file for details.

Support

Support is offered best effort via our Solace Developer Community.

Premium support options are available, please Contact Solace.