vmware by Broadcom

Tanzu Greenplum Streaming Server

Tanzu Greenplum Streaming Server 1.10

You can find the most up-to-date technical documentation on the VMware by Broadcom website at:

https://techdocs.broadcom.com/

VMware by Broadcom
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Contents

VMware Greenplum Streaming Server 1.10 Documentation

This documentation describes how to install, configure, manage, and perform ETL operations with the VMware Greenplum Streaming Server.

Key topics in the VMware Greenplum Streaming Server documentation include:

VMware Greenplum Streaming Server 1.x Release Notes

This document contains pertinent release information about the VMware Greenplum Streaming Server version 1.x releases. The Greenplum Streaming Server (GPSS) is included in certain VMware Greenplum 5.x, 6.x, and 7.x distributions. GPSS is also updated and distributed independently of VMware Greenplum. You may need to download and install the GPSS distribution from Broadcom Support Portal to obtain the most recent version of this component.

Supported Platforms

VMware Greenplum Streaming Server 1.x is compatible with these Operating System and VMware Greenplum versions:

GPSS Version	OS Version	VMware Greenplum Version
all	RHEL 6.x, RHEL 7.x, CentOS 6.x, CentOS 7.x	5.17.0+, 6.x
1.6.0+	Ubuntu 18.04 LTS	6.x
1.7.0+	OEL 7.x, Photon 3, RHEL 8.x	6.x
1.10.3+	RHEL 8.7+, Rocky Linux 8.7+, OEL 8.7+ using Red Hat Compatible Kernel (RHCK)	7.x
1.10.4+	RHEL 9, Rocky Linux 9, OEL 9 using Red Hat Compatible Kernel (RHCK)	6.x, 7.x

Release 1.10

VMware Greenplum Streaming Server version 1.10.x is the last version that supports VMware Greenplum 5.x.

Release 1.10.4

Release Date: November 1, 2023

Greenplum Streaming Server 1.10.4 includes changes and resolves issues.

You are required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.10.4.

Changed Features

Greenplum Streaming Server 1.10.4 includes these changes:

- Version 1.10.4 adds support for Red Hat Enterprise Linux 64-bit 9, Oracle Linux 64-bit 9 using the Red Hat Compatible Kernel (RHCK), and Rocky Linux 9 *for VMware Greenplum version 6.x and 7.x*.
- To alleviate possible data skew, GPSS changes the distribution key that it uses for its Kafka history tables.

Resolved Issues

Greenplum Streaming Server 1.10.4 resolves this issue:

33098
: Resolves an issue where GPSS lost retry information for jobs that were manually stopped and then restarted. This resulted in GPSS returning the warning `retry job <jobname> is disabled, stop schedule` and exiting a job when a primary Greenplum Database segment went down. GPSS now explicitly retains the retry configuration for manually stopped jobs, enabling it to better tolerate a segment failure and mirror switch over.

Release 1.10.3

Release Date: September 21, 2023

Greenplum Streaming Server 1.10.3 includes changes and resolves issues.

You are required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.10.3.

Changed Features

Greenplum Streaming Server 1.10.3 includes these changes:

- The per-run, job-specific server log file now includes the YAML configuration used to submit the job and the job status at completion. You can find the YAML configuration in a log message prefaced with `start job`. The job status log message is prefaced with `job finished`.
- Version 1.10.3 adds support for Red Hat Enterprise Linux 64-bit 8.7+, Oracle Linux 64-bit 8.7+ using the Red Hat Compatible Kernel (RHCK), and Rocky Linux 8.7+ *for VMware Greenplum version 7.0.0*.
- Shadowed passwords are now supported for LDAP user accounts.

Resolved Issues

Greenplum Streaming Server 1.10.3 resolves these issues:

33015

Resolves an issue where GPSS returned a `Resource temporarily unavailable` error due to a resource leak that occurred when it repeatedly retried a Kafka job that consumed illegal JSON. GPSS now ensures that it releases all connections to Kafka when it detects an offset gap.

32935
: The per-run, job-specific server log file did not include enough information about the job. GPSS version 1.10.3 adds the job YAML configuration and the job status at completion to the log file.

Release 1.10.2

Release Date: July 27, 2023

Greenplum Streaming Server 1.10.2 resolves an issue.

You are required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.10.2.

Resolved Issues

Greenplum Streaming Server 1.10.2 resolves this issue:

32960
: Resolves an issue where GPSS returned a `value out of range` error when the object identifier of the target Greenplum Database table was larger than 2^32. GPSS now checks for the existence of the target table rather than attempting to access the table's object identifier.

Release 1.10.1

Release Date: June 9, 2023

Greenplum Streaming Server 1.10.1 includes changes and resolves issues.

You are required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.10.1.

Changed Features

Greenplum Streaming Server 1.10.1 includes these changes:

- The per-run, job-specific server log file name is changed from `gpss-<jobname>_<timestamp>.log` to `gpss_<jobname>_<timestamp>.log`.
- The user name and password are now optional components of the RabbitMQ `SERVER` and `server` (version 3 (Beta)) load configuration file properties.
- GPSS supports TLS encryption only when loading from a RabbitMQ queue. GPSS does not support TLS encryption when loading from a RabbitMQ stream.
- Version 1.10.1 adds support for Red Hat Enterprise Linux 64-bit 8.7+, Oracle Linux 64-bit 8.7+ using the Red Hat Compatible Kernel (RHCK), and Rocky Linux 8.7+ *for VMware Greenplum*

version 7 Beta 4+.

Resolved Issues

Greenplum Streaming Server 1.10.1 resolves these issues:

N/A
: Resolves an issue where, when loading from RabbitMQ, GPSS returned a vague error when the Greenplum Database table specified in the load configuration file did not exist. The message now more accurately reflects the error condition.

N/A
: Resolves an issue where GPSS did not direct certain log messages to the appropriate per-run, job-specific server log file. GPSS now correctly routes these messages.

N/A
: Resolves an issue where certain messages in the per-run, job-specific server log file were missing the job identifier. These log messages now include the job id.

Release 1.10.0

Release Date: May 15, 2023

Greenplum Streaming Server 1.10.0 adds new features and includes changes.

This version of the VMware Greenplum Streaming Server documentation replaces the term *master* with the term *coordinator*.

You are required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.10.0.

New and Changed Features

Greenplum Streaming Server 1.10.0 includes these new and changed features:

- GPSS updates the `go` library dependency to version 1.19.1.
- GPSS introduces support for TLS encryption to RabbitMQ. Refer to Configuring gpss for TLS-Encrypted Communications with RabbitMQ for more information.
- GPSS v1.10.0 includes these logging-related changes and new features:
    - Version 1.10.0 changes the naming format of GPSS server log files. Previous versions of the server log file name included a date. The new naming format replaces the date with a timestamp that specifies the day and time including milliseconds. Refer to Managing GPSS Log Files for more information. (Upgrade actions may be required as described in Upgrading the Streaming Server.)
    - Log messages that GPSS writes to server log files now include the job identifier (truncated to 8 characters).

    - You can direct GPSS to automatically rotate the server log file on an hourly or daily basis by setting the new `Logging:Rotate` property in the gpss.json server configuration file. See Configuring Automatic Server Log File Rotation for more information about this new feature.
    - You can direct GPSS to create per-run server log files for each job by setting the new `Logging:SplitByJob` property in the gpss.json server configuration file. Refer to Configuring Per-Run Server Log Files for more information.
- The GPSS gRPC Batch Data API exposes a new `ConnectionRequest` message field named `SessionTimeout` that allows the developer to specify the maximum amount of idle time before GPSS releases a connection to Greenplum Database. If you choose to make use of this feature in your GPSS client application, upgrade actions are required as described in Upgrading the Streaming Server.

Release 1.9

Release Date: March 10, 2023

Greenplum Streaming Server 1.9.0 adds new features, includes changes, and resolves issues.

You are required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.9.0.

New and Changed Features

Greenplum Streaming Server 1.9.0 includes these new and changed features:

- GPSS now invokes the user-defined functions or SQL commands that you provide in `TEARDOWN_SQL` (version 2) or `teardown_statement` (version 3 (Beta)) on both job success and failure. The functions/commands were previously invoked only when the job was successful.
- The gpsscli dryrun command now supports the `--property <template_var>=<value>` option. This allows you to use property template variables in the load configuration file that you provide to the command.
- You can optionally provide the `--name <jobname>` option to the `gpsscli dryrun` command to name the dry run job.
- GPSS introduces a new `ENCODING` (version 2) / `encoding` (version 3 (Beta) property to the load configuration file that allows you to specify the character set encoding for source data that is of the `csv`, `custom`, `delimited`, or `json` formats.
- GPSS introduces a new `FILTER` (version 2) / `filter` (version 3 (Beta)) property to the load configuration file that allows you to specify an *output filter* for a job. An output filter may be useful when you want to write different data to multiple Greenplum Database output tables.
- GPSS introduces a new `ALERT` (version 2) / `alert` (version 3 (Beta)) property block to the load configuration file that allows you to register for a job stopped notification, specifying a command that GPSS will run when a job is stopped.

- GPSS introduces a new `TRANSFORMER` (version 2) / `transformer` (version 3 (Beta)) property block to the load configuration file that allows you to specify input and/or output transform functions for the data. An input transformer is a `go` plugin, an output transformer is a user-defined SQL function (UDF). GPSS supports specifying transforms only when loading from Kafka or RabbitMQ data sources.
- GPSS now supports reading Kafka and RabbitMQ messages that contain multiple lines when included in only one of the key or value input data (not both).
- The GPSS RabbitMQ data source is no longer Beta, it is promoted to a supported feature.
- The GPSS RabbitMQ data source now supports `strong` consistency for streams. Refer to Understanding RabbitMQ Message Offset Management for more information about how GPSS manages RabbitMQ offsets and message consistency.

Resolved Issues

Greenplum Streaming Server 1.9.0 resolves these issues:

32640
: Resolves an issue where idle `SELECT VERSION()` queries consumed connection resources.

Release 1.8

Release 1.8.1

Release Date: December 21, 2022

Greenplum Streaming Server 1.8.1 includes changes and resolves issues.

You are required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.8.1.

Changed Features

Greenplum Streaming Server 1.8.1 includes these changes:

- GPSS now names the external table that it creates for an `S3` load job with the `s3ext` prefix. The prefix was previously `S3ext`.

Resolved Issues

Greenplum Streaming Server 1.8.1 resolves these issues:

32584
: Resolves an issue where the Greenplum Streaming Server returned the error `pq: password authentication failed for user` when a load job specified no password because it did not clear the configuration of the previous job.

32522

Resolves an issue where the Greenplum Streaming Server exposed the shadow password string in the logs. GPSS now obscures the password in the log file.

32498

Resolves a resource leak issue where, when a Kafka job failed, the Greenplum Streaming Server did not close the Kafka metadata consumer.

N/A

Resolves an issue where the Greenplum Streaming Server calculated the job identifier hash incorrectly when the RabbitMQ load configuration file specified a queue source.

Release 1.8.0

Release Date: September 9, 2022

Greenplum Streaming Server 1.8.0 adds new features, includes changes, and resolves issues.

You are required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.8.0.

New and Changed Features

Greenplum Streaming Server 1.8.0 includes these new and changed features:

GPSS Configuration

The `gpss.json` server configuration file now includes a `Gpfdist:Certificate:DBClientShared` property. Use this boolean property to instruct GPSS to reuse the Gpfdist SSL certificate for the control channel (client) connection to Greenplum Database. Configuring SSL for the Control Channel provides the relevant configuration information.

General

- When `ReuseTables` is set to `false`, GPSS now creates each job's external table using the job name rather than a hash. This enables you to more easily track external tables per-job. About External Table Naming and Lifecycle describes how GPSS names external tables, and also provides information about their lifecycle.
- GPSS introduces new scheduling options that allow you to configure automatic stop and restart conditions for jobs. You specify the `RUNNING_DURATION`, `AUTO_STOP_RESTART_INTERVAL`, `MAX_RESTART_TIMES`, and `QUIT_AT_EOF_AFTER` (version 2) or `running_duration`, `auto_stop_restart_interval`, `max_restart_times`, and `quit_at_eof_after` (version 3 (Beta)) options in the `SCHEDULE/schedule` block of the load configuration file.
- GPSS enhances the `delimited` data format to support setting quote and escape characters and an end-of-line prefix string when you use the format to load data into Greenplum Database.

Kafka Data Source

- GPSS changes the name of the version 3 (Beta) load configuration file `window` property to `task`.

- GPSS records in the progress log file the total number of rows that it processes in a Kafka message. Now, when loading `jsonl`, `delimited`, and `csv` format data where a Kafka message can include multiple rows, the `total_rows_read` identifies the Kafka message and the new `total_rows` field identifies the total number of rows inserted and rejected.
- The Kafka data source exposes a new metadata field named `timestamp`. This `int64`-type field identifies the time that a message was written to the Kafka log.
- When `SAVE_FAILING_BATCH` is `true`, GPSS records the time that a record was inserted into the backup table. The name of the new column is `gpss_save_timestamp`. Refer to Redirecting Data to a Backup Table when GPSS Encounters Expression Evaluation Errors for a discussion of the backup table schema.
- When `RECOVER_FAILING_BATCH (Beta)` is `true`, GPSS reports more information about the result of the operation, including the batch size and number of records recovered.

File Data Source

- The file data source now supports the `delimited` data format.
- The file data source can now load the `stdout` of a command into a Greenplum Database table. You specify command specifics via the new `EXEC` (version 2) or `exec` (version 3 (Beta)) block in the load configuration file.
- GPSS now supports initiating a dry run of a file job.

New RabbitMQ Data Source (Beta)

GPSS introduces Beta support for loading from a RabbitMQ data source. You can load messages from a RabbitMQ queue or stream into Greenplum Database. Refer to Loading from RabbitMQ into Greenplum (Beta) for more information about using this new Beta feature, and rabbitmq-v3.yaml (Beta) and rabbitmq-v2.yaml (Beta) for more information about the supported load configuration file properties.

Resolved Issues

Greenplum Streaming Server 1.8.0 resolves these issues:

32278, 32180
: Resolves an issue where a Greenplum Database cluster using `pgbouncer` to manage connections did not receive a client SSL certificate as expected. GPSS now exposes a `DBClientShared` GPSS server configuration property that you can use to instruct GPSS to present the Gpfdist certificate as the client SSL cert to Greenplum Database.

32096, 31802
: Resolves an issue where GPSS was unable to automatically stop a job based on run time by exposing new job scheduling properties.

32044
: Resolves an issue where the recovery of a failed batch (Beta) could not be adequately monitored. GPSS now records the time that a record is inserted into the backup table in a new column named `gpss_save_timestamp`. GPSS also reports more information during bad batch recovery operations.

32144

Resolves an issue where external tables used by GPSS were difficult to locate. Now, when `ReuseTables` is `false`, GPSS names the external table using the job name instead of a hash of configuration properties.

182386619

GPSS would incorrectly fall back (to earliest or latest offset) all Kafka partitions, even those without offset gaps. This issue is resolved; GPSS now falls back only those partitions that have experienced an offset gap and writes this information to the GPSS log.

N/A

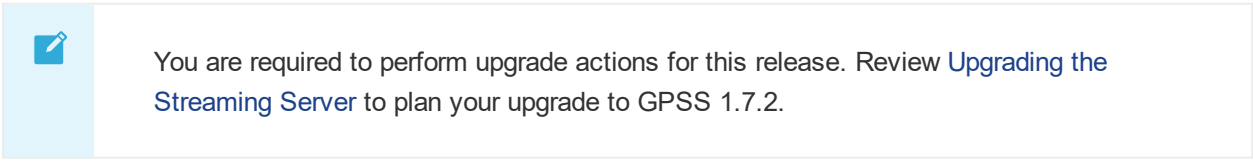
Resolves an issue where GPSS did not reset `MAX_RETRIES` after a job was successfully submitted and running.

Release 1.7

Release 1.7.2

Release Date: April 21, 2022

Greenplum Streaming Server 1.7.2 includes changes and resolves issues.



You are required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.7.2.

Changed Features

Greenplum Streaming Server 1.7.2 includes these changes:

- GPSS adds support for specifying backslash escape sequences when you set the following CSV options: delimiter, quote, and escape. GPSS supports the standard backslash escape sequences for backspace, form feed, newline, carriage return, and tab, as well as escape sequences that you specify in hexadecimal format (prefaced with `\x`). Refer to Backslash Escape Sequences in the PostgreSQL documentation for more information.
- To resolve issue `32168`, GPSS version 1.7.2 introduces support for loading files or messages that contain one JSON record per line into Greenplum Database. To use this new feature, you must specify `FORMAT: jsonl` in version 2 format load configuration files, or specify `json` format with `is_jsonl: true` in version 3 (Beta) format load configuration files.

Resolved Issues

Greenplum Streaming Server 1.7.2 resolves these issues:

32168

Resolves an issue where GPSS did not support loading multi-line JSON files into Greenplum Database. GPSS 1.7.2 introduces support for loading JSON message or file data that contains a single JSON record per line.

N/A

Resolves an issue where GPSS did not support escape sequences that were specified in the CSV delimiter, quote, and escape options. GPSS now supports standard and hexadecimal-format

backslash escape sequences.

Release 1.7.1

Release Date: March 31, 2022

Greenplum Streaming Server 1.7.1 resolves issues.

You are required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.7.1.

Resolved Issues

Greenplum Streaming Server 1.7.1 resolves these issues:

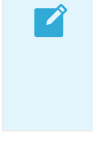
32105
: Resolves an issue where GPSS incorrectly added an offset based on the Greenplum Database local time zone to `timestamp` (without timezone) types that it loaded into a Greenplum Database table.

181293923
: In some cases, GPSS returned the error `pq: missing data for column *name*` when loading a file containing CSV-format data. This issue is resolved; GPSS no longer automatically adds a newline when one already exists at the end of the file.

Release 1.7.0

Release Date: March 18, 2022

Greenplum Streaming Server 1.7.0 adds new features, includes changes, and resolves issues.



You are required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.7.0.

New and Changed Features

Greenplum Streaming Server 1.7.0 includes these new and changed features:

OS and Platforms

- GPSS introduces support for Red Hat Enterprise Linux 8 and Photon 3 for VMware Greenplum 6, and now provides download packages for these operating system versions on Broadcom Support Portal.
- GPSS updates the version of `go` that it uses to build the CLI tools to version 1.17.6 to mitigate CVE-2021-44716.

GPSS Configuration

GPSS introduces a default timeout of 10 seconds for a `gpss` service instance to connect to Greenplum Database and a related environment variable named `GPDB_CONNECT_TIMEOUT`. You can set this environment variable to change the amount of time that GPSS waits to establish a connection to Greenplum Database as described in Running the Greenplum Streaming Server.

Authentication

- After it encounters an SSL connection failure on the control channel, GPSS will attempt to initiate a non-SSL connection on the channel.
- The `gpss.json` server configuration file now includes an `Authentication` property block. Use the configuration properties in this block to specify a user name and password for client authentication to the GPSS server. Refer to Configuring the Streaming Server for Client-to-Server Authentication for additional information about this new feature.
- GPSS adds the `-U/--username` and `-P/--password` options to the gpsscli subcommands to specify the user name and password for client authentication to the GPSS server.

Kafka Data Source

- GPSS now saves the `topic:partition:offset` for each badly-formatted Kafka message written to the error log; you can view this information when you run the `SELECT * FROM gp_read_error_log('<exttbl>')` command.
- GPSS adds the `--skip-explain` flag to the gpsscli start subcommand to skip the explain SQL check step of its internal processing.
- GPSS now supports loading from a single kafka topic into multiple Greenplum Database tables. Provide an `OUTPUTS:TABLE` (version 2) or `targets:gpdb:tables:table` (version 3 (Beta)) block for each table, and specify the properties that identify the data targeted to each.
- GPSS introduces a new datatype named `gp_json` (Beta) to the `dataflow` extension. For additional information about using the `gp_json` data type, refer to About the JSON Format and Column Type documentation.

File and Kafka Data Sources

- GPSS adds support for new CSV options for file and Kafka jobs. You can now specify the delimiter, quote, and null string values in the load configuration file. You can identify a list of columns whose values GPSS forces to be not null. You can also specify GPSS's behaviour when it encounters missing trailing fields in a row of data. New version 2 property names include `DELIMITER`, `QUOTE`, `NULL_STRING`, `ESCAPE`, `FORCE_NOT_NULL`, and `FILL_MISSING_FIELDS`. New version 3 property names include `delimiter`, `quote`, `null_string`, `escape`, `force_not_null`, and `fill_missing_fields`.
- GPSS exposes new `PREPARE_SQL` and `TEARDOWN_SQL` (version 2) and `prepare_statement` and `teardown_statement` (version 3) load configuration file properties for Kafka and file data sources. You can use the properties to specify user-defined function or SQL commands for GPSS to run before executing a job, and/or at job completion.

version 3 (Beta) Configuration

GPSS 1.7.0 adds, changes, and relocates property keywords in the version 3 (Beta) configuration file format. Refer to the gpsscli-v3.yaml (Beta), gpkafka-v3.yaml (Beta), and filesource-v3.yaml (Beta) reference pages for the new keywords and locations.

New S3 Data Source (Beta)

GPSS 1.7.0 introduces Beta support for a new data source, S3. This data source does not read directly from S3, but rather uses the Greenplum Database s3 protocol and external tables to read from s3 and write to Greenplum in parallel. Refer to Loading from S3 into Greenplum (Beta) for more information about using this new feature, and s3source-v3.yaml (Beta) for the supported load configuration file properties.

New Commands and Options

- GPSS adds the new gpsscli dryrun subcommand. When you invoke this command, GPSS performs a trial run of a Kafka or S3 job without actually writing to Greenplum Database. You can use the command to help diagnose load job errors as described in Diagnosing an Error with a Trial Load.
- GPSS adds the `-f/--force` flag to the gpsscli remove subcommand to forcibly stop and remove a GPSS job(s).

Other Changes

- GPSS adds new *Submitted* and *Success* statuses for batch (file, s3) jobs. GPSS 1.7.0 also changes the *Stopped* status to signify that a job was stopped by the user. Refer to the gpsscli status reference page for a description of GPSS job statuses.
- GPSS 1.7.0 removes the Streaming Job API (Beta) documentation.

Resolved Issues

Greenplum Streaming Server 1.7.0 resolves these issues:

CVE-2021-44716
: Updates the `go` library to version 1.17.6.

N/A
: You can now specify an Avro schema file path for both the key and the value when you load Kafka data into Greenplum Database.

N/A
: Resolves an issue where GPSS erroneously inserted a `\n` after parsing 76 characters of Avro data when the load configuration file specified `bytes_to_base64: true`.

32022
: Resolves an issue where GPSS did not provide any way to run SQL commands before GPSS initiates a job or after a GPSS job completes by exposing new properties in version 2 and version 3 (Beta) load configuration files (`PREPARE_SQL/TEARDOWN_SQL` and `prepare_statement/teardown_statement`).

31886
: Resolves an issue where GPSS returned an authentication error when SSL was deactivated for the user (i.e. there was a `hostnossl` connection type entry configured for the user in the `pg_hba.conf`

file). GPSS now attempts to initiate a non-SSL connection when it encounters an SSL connection failure on the control channel.

Release 1.6

Release 1.6.0

Release Date: May 28, 2021

Greenplum Streaming Server 1.6.0 adds new features, includes changes, and resolves issues.

You are required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.6.0.

New and Changed Features

Greenplum Streaming Server 1.6.0 includes these new and changed features:

- GPSS adds the `-c | --config` flag/option to the `gpss` command to specify the JSON-formatted configuration file.
- The `gpsscli --version` command now displays the version of the GPSS server in addition to displaying that of the client.
- The `gpss.json` server configuration file now includes a `KeepAlive` property block. Use the configuration properties in this block to specify timeout options for the gRPC connection between the GPSS client and the GPSS server.
- GPSS changes the format of front-end logs (messages written by commands to `stdout`) from CSV format to a more human-readable format. Related, GPSS adds a `--csv-log` option to the commands to write the front-end logs in CSV format. GPSS also adds a `--color` option to commands to enable the use of color in message display.
- GPSS exposes a new load configuration property for Kafka data sources named `IDLE_DURATION` (version 2 configuration) and `idle_duration_ms` (version 3 configuration). Use this property to specify that GPSS use lazy load mode, waiting until data arrives before locking the target Greenplum Database table.
- GPSS exposes a new load configuration property for Kafka data sources named `SCHEMA_PATH_ON_GPDB` (version 2 configuration) and `schema_path_on_gpdb` (version 3 configuration). Use this property to specify the path to the Avro `.avsc` file that contains the schema of the Kafka key or value data (but not both). This file must reside in the same location on all Greenplum Database segment hosts.
- GPSS exposes a new load configuration property for Kafka data sources named `FALLBACK_OFFSET` (version 2 configuration) and `fallback_offset` (version 3 configuration). Use this property to specify that GPSS automatically handle Kafka message offset mismatches, and how.
- GPSS exposes new load configuration properties for Kafka data sources to support access to an SSL-secured schema registry. Refer to Accessing an SSL-Secured Schema Registry for more information.

- GPSS now supports acting as a high-level Kafka consumer when the Kafka client properties include a `group.id` setting.
- GPSS exposes a new load configuration property for Kafka data sources named `CONSISTENCY` (version 2 configuration) and `consistency` (version 3 configuration). Use this property to specify how GPSS manages Kafka message offsets when it acts as a high-level consumer. Refer to Understanding Kafka Message Offset Management for more information.
- GPSS 1.6.0 provides additional documentation about developing and using custom formatters with GPSS.

Beta Features

Greenplum Streaming Server 1.6.0 includes these new *Beta* features:

- GPSS exposes a new load configuration property for Kafka data sources named `RECOVER_FAILING_BATCH` (version 2 configuration) and `recover_failing_batch` (version 3 configuration). Use this property in conjunction with `SAVE_FAILING_BATCH` to instruct GPSS to automatically reload the good data in the batch, and retain only the error data in the backup table.

  Note: Enabling this feature may have severe performance implications when any data in the Kafka topic generates an expression error.

  Note: This feature requires that GPSS has the Greenplum Database privileges to create a function.
- GPSS adds a new extension named `dataflow`. This extension includes a new data type, `gp_jsonb` (available for Greenplum Database version 6.x only), and a new formatter, `text_in`. You must `CREATE EXTENSION dataflow;` in each database in which you choose to use these types and formatters. For additional information about the `gp_jsonb` data type, see About the JSON Format and Column Type.

Resolved Issues

Greenplum Streaming Server 1.6.0 resolves this issue:

31458
: Resolves an issue where job progress information was available only via `stdout`. GPSS now supports consumer groups, which saves message offsets to the Kafka topic.

31396
: Resolves an issue where the GPSS Ubuntu download package was missing certain dependent libraries. These libraries are now marked as required.

31359
: Resolves an issue where GPSS could not restart a job that had been stopped for a long period of time. GPSS now supports a `FALLBACK_OPTION` load configuration property that instructs GPSS to automatically handle offset mismatches, and how to handle them.

31315
: Resolves an issue where GPSS was unable to load data from Kafka when TLS-secured communication was required between the Kafka broker and the schema registry. GPSS now supports load configuration properties to specify the certificates and keys required for this communication.

31278

Resolves an issue where GPSS was unable to load Avro data when the schema was not embedded in the `.avro` file. GPSS now supports the `SCHEMA_PATH_ON_GPDB` load configuration property to specify the `.avsc` schema file.

31277

Resolves a request for a job timeout by supporting a new `IDLE_DURATION` load configuration property.

30723, 30711

Resolves an issue where GPSS failed to load JSON-format data that included `\u0000` by creating a new Greenplum Database data type named `gp_jsonb` (Beta).

Release 1.5

Release 1.5.3

Release Date: April 15, 2021

Greenplum Streaming Server 1.5.3 resolves an issue.

You are required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.5.3.

Resolved Issues

Greenplum Streaming Server 1.5.3 resolves this issue:

31357

Resolves an issue where GPSS did not correctly handle `CUSTOM_OPTION` properties specified in a load configuration file. GPSS now supports using the `NAME` and `PARAMSTR` properties to specify a custom formatter user-defined function.

Release 1.5.2

Release Date: March 5, 2021

Greenplum Streaming Server 1.5.2 resolves several issues.

You are required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.5.2.

Changed Features

Greenplum Streaming Server 1.5.2 includes this change:

- GPSS omits the end time in its output error hints. Resolved issue 31287 provides more information.

Resolved Issues

Greenplum Streaming Server 1.5.2 resolves these issues:

N/A
: Resolves an issue where GPSS logged the message `execInsert and err: nil` because it did not check for an error before logging.

31287
: Resolves an issue where GPSS did not always display the correct end time in the output error hint by removing the end time condition.

177153850
: Resolves an issue where a GPSS query returned a syntax error from Greenplum Database because `MATCH COLUMNS` was empty. GPSS now requires and checks that this field includes at least one column when you submit a load job that specifies `UPDATE` or `MERGE` mode.

177133400
: Resolves an issue where GPSS stopped a Kafka job unexpectedly and did not return an error when it encountered a batch that contained only a control message.

177077055
: Resolves an issue where the `--all` option was incorrectly displayed in the help output of the `gpsscli load` command.

177077007
: GPSS consumed a large mount of memory caching Kafka messages when it ran many concurrent jobs that read from multiple partitions. This issue is resolved; GPSS now specifies a less aggressive default value for the `librdkafka queued.max.messages.kbytes` property when the user does not explicitly configure it.

177014072
: Resolves an issue where GPSS incorrectly returned the error `gpkafka load show job progress fail, err: job progress is nil` when it failed to start a Kafka job. GPSS now returns the more meaningful error `gpkafka load start job failed` in this situation.

176842005
: Resolves an issue where GPSS submitted a job with the wrong name when a `gpsscli load *.yaml` command operated on more than one load job.

Release 1.5.1

Release Date: February 5, 2021

Greenplum Streaming Server 1.5.1 includes changes and resolves issues.

You are required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.5.1.

Changed Features

Greenplum Streaming Server 1.5.1 includes these changes:

- Version 1.5.1 is the first standalone GPSS release that includes a `.deb` installation package for Ubuntu 18.04 LTS systems.
- The `gpsscli` subcommands now consistently return zero (`0`) on success and non-zero when GPSS encounters an error.

- GPSS improves the error message that it returns when it encounters a mismatched extension or formatter version.
- GPSS bundles a patched version of the `libserdes` library to fix an issue that can arise when the `SCHEMA_REGISTRY_ADDRS` property value includes a trailing slash. See resolved issue 31137.
- GPSS now registers the `gp_read_persistent_error_log()` function when you register the GPSS extension in a database. Resolved issue 31201 provides more information.
- The progress log file name format has changed; the new format retains the complete job name rather than truncating it to 8 characters.

Resolved Issues

Greenplum Streaming Server 1.5.1 resolves these issues:

31201
: Resolves an issue where GPSS returned a `permission denied for language c` error when it attempted, at runtime, to register an internal function as the Greenplum Database user that started GPSS, and this user did not have the privileges required to create such functions. GPSS now registers this internal function when you create the GPSS extension in a database.

31137
: Due to a bug in the dependent library `libserdes`, GPSS did not correctly handle a trailing slash when specified in the first address in a list of `SCHEMA_REGISTRY_ADDR`s. This issue is resolved; GPSS 1.5.1 bundles a patched version of the `libserdes` library that can handle such addresses.

176136800
: Resolves an issue where GPSS returned an error when it interpreted and parsed the `SAVE_FAILING_BATCH` property and value in a (deprecated) version 1 load configuration file, when version 1 of the file does not support this property. GPSS now displays a warning message when it encounters a property that is not supported in a version 1 configuration file.

176068963
: GPSS reported an offset gap when it read Kafka messages using the `read_committed` isolation level, the job was restarted, and the topic retention period had expired. This issue is resolved; GPSS now records control message offsets.

175867685
: Resolves an issue where the `-i | --edit-in-place` option was displayed in the help output of subcommands that did not support the option. GPSS now correctly displays the option only for the `gpsscli convert` command.

175867670
: Resolves an issue where the `gpsscli` subcommands did not return consistent values. `gpsscli` now returns zero (`0`) on success and non-zero on failure.

n/a
: Resolves an issue where GPSS did not correctly validate a `filesource.yaml` load configuration file before submitting the job.

Release 1.5.0

Release Date: December 2, 2020

Greenplum Streaming Server 1.5.0 adds new features, includes changes, and resolves issues.

You are required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.5.0.

New and Changed Features

Greenplum Streaming Server 1.5.0 includes these new and changed features:

- The load configuration file `ERROR_LIMIT` property, previously mandatory, is now optional. The default value for the property is zero (`0`); GPSS deactivates error logging and stops a load operation upon encountering the first error.
- GPSS includes out-of-the-box Prometheus integration, enabling you to use the tool to monitor your `gpss` server instances. Refer to Monitoring GPSS Service Instances for more information on enabling and using this integration.
- New configuration properties in the `gpss.json` server configuration file include:
    - The `DebugPort` configuration property. You can use this property to identify the port number on which GPSS starts a debug server for the `gpss` server instance. Refer to Pulling Information from the Debug Server for more information.
    - The `MinTLSVersion` configuration property. You use this property to specify the minimum TLS version that GPSS requests on encrypted connections.
    - The `Logging` configuration property block. You can use these configuration properties to set the front-end and back-end logging levels for GPSS commands. See About GPSS Logging.
    - The `JobStore` configuration property block. Use the configuration property in this block to specify a local directory in which GPSS maintains job status information. This allows a GPSS server instance to (re)start any in-progress jobs when the instance first starts up. See About GPSS Job Management.
    - The `Monitor` configuration property block. You use this property to enable GPSS Prometheus integration.
- GPSS no longer generates and assigns a unique identifier as the job name when you invoke the `gpsscli submit` or `gpsscli load` commands without specifying the `--name` option. GPSS now assigns the base name of the load configuration file as the default job name.
- GPSS exposes a new load configuration property for Kafka data sources named `PARTITIONS`. Use this property to specify the specific partition numbers from which you want GPSS to load Kafka messages from the topic. (This property is not supported for the Kafka version 1 configuration file format.)
- GPSS supports specifying template parameters for load configuration file properties. When you specify the `{{template\_var}}` value syntax in the file, GPSS substitutes `template\_var` with a `value` that you specify via the `-p | --property template\_var=value` option when you submit or load the job.
- GPSS supports SSL encryption on the control channel between GPSS and the Greenplum Database coordinator, and ships with an updated `pq` library to support this feature. See Configuring

SSL for the Control Channel for configuration information.

- The `gpsscli start`, `stop`, and `remove` subcommands now support a `--all` flag. When you specify this flag, GPSS: starts all submitted jobs, stops all running jobs, or removes all stopped jobs.
- The `gpsscli submit` and `gpsscli load` commands can now operate on one or more YAML load configuration files.
- GPSS exposes the new `SAVE_FAILING_BATCH` load configuration property. When you set this property to `true`, GPSS also writes loading data to a backup table. When GPSS encounters expression evaluation errors, this backup table aids in the recovery of the load operation. See Redirecting Data to a Backup Table when GPSS Encounters Expression Evaluation Errors for additional information. (This property is not supported for the Kafka version 1 configuration file format.)
- GPSS 1.5.0 introduces a new Beta feature, the version 3 load configuration file format. This format introduces a new YAML organization and keywords, and more closely aligns with the GPSS gRPC Streaming Job API. Refer to gpsscli-v3.yaml (Beta) for the version 3 syntax.
- GPSS 1.5.0 supports the persisent error log feature of Greenplum Database when you are running against Greenplum version 5.26+ or 6.6+. For more details about the persisent error log, refer to the CREATE EXTERNAL TABLE SQL reference page in the Greenplum Database documentation.

Resolved Issues

Greenplum Streaming Server 1.5.0 resolves these issues:

30332
: In some cases when GPSS reused external tables for jobs, it did not update the external table that it uses internally for load operations when the target Greenplum table definition was modified.

171299427
: Resolves an issue where GPSS was unable to cancel a batch write operation when it encountered an error, and left a lingering session.

Release 1.4

Release 1.4.3

Release Date: December 17, 2021

Greenplum Streaming Server 1.4.3 resolves an issues and includes related changes.

You may be required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.4.3.

Changes

Greenplum Streaming Server 1.4.3 includes this change:

- After it encounters an SSL connection failure on the control channel, GPSS will attempt to initiate a non-SSL connection on the channel.

Resolved Issues

Greenplum Streaming Server 1.4.3 resolves this issue:

31886
: Resolves an issue where, after upgrade to version 1.4.2, GPSS returned an authentication error when SSL was deactivated for the user (i.e. there was a `hostnossl` connection type entry configured for the user in the `pg_hba.conf` file). GPSS now attempts to initiate a non-SSL connection when it encounters an SSL connection failure on the control channel.

Release 1.4.2

Release Date: November 2, 2020

Greenplum Streaming Server 1.4.2 resolves issues and includes related changes.

You may be required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.4.2.

Changes

Greenplum Streaming Server 1.4.2 includes these changes:

- GPSS now specifies the SSL `prefer` mode on the control channel to the Greenplum Database coordinator host. GPSS previously explicitly deactivated SSL on the channel.

Resolved Issues

Greenplum Streaming Server 1.4.2 resolves these issues:

n/a
: Resolves an issue where GPSS recorded an incorrect count in the progress log file when the messages it received included offset gaps, such as with transaction control messages.

30776, 174685715
: Resolves an issue where `gpsscli stop` would not respond (hang).

174685711
: Resolves an issue where GPSS failed to load a large (>2GB) file. GPSS now transfers a file in multiple, smaller chunks when loading to Greenplum.

174984151
: GPSS sent an HTTP request to the Avro schema registry service on every segment on every commit; in some cases, this created and destroyed a large number of TCP connections in the process. GPSS resolves this issue by reading the schema a single time per session (as long as the schema remains unchanged).

Release 1.4.1

Release Date: August 7, 2020

Greenplum Streaming Server 1.4.1 resolves issues and includes related changes.

You may be required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.4.1.

Changes

Greenplum Streaming Server 1.4.1 includes these changes:

- GPSS bundles a patched version of the `librdkafka` library to fix an issue that can arise when the Kafka topic that GPSS loads includes messages with discontinuous offsets. See resolved issue 30797, 30776.
- GPSS now always tracks Kafka job progress in a separate, CSV-format log file. See resolved issue 173603095 and Checking the Progress of a Load Operation.
- GPSS 1.4.1 changes the format and content of the server and client log file messsages. The old log file format was delimited text, which could not be parsed when the text contained a newline. The log files are now CSV-format and include a header row. See resolved issue 173603029 and Examining GPSS Log Files.

Resolved Issues

Greenplum Streaming Server 1.4.1 resolves these issues:

n/a
: When the schema registry service was down, GPSS appeared to hang during a Kafka load operation because it tried to access the registry multiple times for each Kafka message. This issue is resolved; GPSS now reports an error and stops retrying immediately when it detects that the schema registry is down.

30797, 30776
: Due to a bug in the dependent library `librdkafka`, a load job from Kafka would hang when there were aborted Kafka transactions in the topic, or when the messages were deleted before GPSS was able to consume them. This issue is resolved. GPSS 1.4.1 bundles a patched version of the `librdkafka` library and can now handle message offsets that are not continuous.

30760
: Certain merge/update operations failed with the error `Cannot parallelize an UPDATE statement that updates the distribution columns` because GPSS versions 1.3.5 through 1.4.0 used the Greenplum Postgres Planner by default, which does not support updating columns that are specified as the distribution key. GPSS 1.4.1 resolves this issue by not explicitly specifying a query planner/optimizer, but rather using the default that is configured in the Greenplum cluster.

173653147
: In some cases, `gpsscli stop` would hang when you invoked it to stop a Kafka load job that GPSS had previously retried. This issue is resolved.

173637940
: The GPSS utilities distributed in the Greenplum Database 6.8.x and 6.9.0 *Client and Loader Tools* packages were missing the dependent library `libserdes.so`. This issue is resolved, the package now includes this library.

173637900

The GPSS 1.4.1 Batch Data gRPC API fixes a parallel loading regression that manifested itself when the `gpss.json` server configuration file included the (default) `ReuseTables: true` property setting.

173603095

Because GPSS tracked job progress only during `gpsscli progress` command execution, the progress information for jobs for which you did not run the command was lost. This issue is resolved. GPSS now always tracks job progress in a separate, CSV-format log file (with header row) named `progress_*jobname*_*jobid*_*date*.log`.

173603029

GPSS log file messages with embedded newlines could not be parsed. This issue is resolved; GPSS changes the client and server log file format to CSV (with header row).

Release 1.4.0

Release Date: June 26, 2020

Greenplum Streaming Server 1.4.0 adds new features, includes changes, and resolves issues.

You may be required to perform upgrade actions for this release. Review Upgrading the Streaming Server to plan your upgrade to GPSS 1.4.0.

New and Changed Features

Greenplum Streaming Server 1.4.0 includes these new and changed features:

- GPSS supports loading from a file data source. You can now load data in Avro, binary, CSV, and JSON files into Greenplum Database. See Loading File Data into Greenplum for more information.
- GPSS defines a new `META` load configuration property block. You can load the properties in this single JSON-format column into the target table, or use the properties in update or merge criteria for a load operation. The available `META` properties are data-source specific:
    - The Kafka data source exposes the following `META` properties: `topic` (`text`), `partition` (`int`), and `offset` (`bigint`).
    - The file data source exposes a single `META` property named `filename` (`text`).
- GPSS supports Avro data containing binary fields.
- GPSS implements a faster update in merge mode for large datasets when the load configuration specifies no `UPDATE_COLUMNS`. In this scenario, GPSS updates all `MAPPING` columns in each row.
- You can use GPSS to load data into a Greenplum Database cluster that utilizes the PgBouncer connection pooler.
- The CentOS 7.x GPSS packages for Greenplum 6 support Oracle Enterprise Linux 7.
- GPSS uses a single thread and socket per partition by sharing a Kafka consumer between workers.
- GPSS bundles `librdkafka` version 1.4.2. This version provides support for controlling how GPSS reads Kafka messages written transactionally via the `isolation.level` property.
- GPSS 1.4 introduces the new Streaming Job API (Beta), a gRPC API that allows you to manage and submit streaming jobs to the server.

Resolved Issues

Greenplum Streaming Server 1.4.0 resolves these issues:

172142789
: The GPSS Batch Data gRPC API fixes inaccurate `TransferStats` success and error counts for data load operations initiated in update mode.

Deprecated Features

Deprecated features may be removed in a future minor release of the Greenplum Streaming Server. GPSS 1.4.x deprecates:

- The `gpkafka` Version 1 configuration file format (deprecated since 1.4.0).
- The `gpkafka.yaml` (versions 1 and 2) `POLL` block, including the `POLL:BATCHSIZE` and `POLL:TIMEOUT` properties (deprecated since 1.3.5).

Removed Features

Deprecated features may be removed in a future minor release of the Greenplum Streaming Server. GPSS 1.4.x removes:

- The `gpsscli history` and `gpkafka history` commands (deprecated in 1.3.5).

Release 1.3

Release 1.3.1

Release Date: December 19, 2019

Greenplum Streaming Server version 1.3.1 is the first standalone release of GPSS. GPSS 1.3.1 is also included in the Greenplum Database version 5.24 and 6.2 distributions.

Greenplum Streaming Server 1.3.1 is a maintenance release that resolves several issues.

Resolved Issues

Greenplum Streaming Server 1.3.1 resolves these issues:

169806983
: In some cases, reading from Kafka using the default `MINIMAL_INTERVAL` (0 seconds) caused GPSS to consume a large amount of CPU resources, even when no new messages existed in the Kafka topic. This issue is resolved.

169807372, 169831558
: GPSS 1.3.0 did not recognize internal history tables that were created with GPSS 1.2.6 and earlier. In some cases, this caused GPSS to load duplicate messages into Greenplum Database. This issue is resolved.

Release 1.3.0

Release Date: November 1, 2019

Greenplum Streaming Server version 1.3.0 is included in the Greenplum Database version 5.23 and 6.1 distributions.

Greenplum Streaming Server 1.3.0 is a minor release that includes new and changed features and resolves several issues.

New and Changed Features

Greenplum Streaming Server 1.3.0 includes these new and changed features:

- GPSS now supports log rotation, utilizing a mechanism that you can easily integrate with the Linux `logrotate` system. See Managing GPSS Log Files for more information.
- GPSS has added the new `INPUT:FILTER` load configuration property. This property enables you to specify a filter that GPSS applies to Kafka input data before loading it into Greenplum Database.
- GPSS displays job progress by partition when you provide the `--partition` flag to the `gpsscli progress` command.
- GPSS enables you to load Kafka data that was emitted since a specific timestamp into Greenplum Database. To use this feature, you provide the `--force-reset-timestamp` flag when you run `gpsscli load`, `gpsscli start`, or `gpkafka load`.
- GPSS now supports update and merge operations on data stored in a Greenplum Database table. The load configuration file accepts `MODE`, `MATCH_COLUMNS`, `UPDATE_COLUMNS`, and `UPDATE_CONDITION` property values to direct these operations. Example: Merging Data from Kafka into Greenplum Using the Streaming Server provides an example merge scenario.
- GPSS supports Kerberos authentication to both Kafka and Greenplum Database.
- GPSS supports SSL encryption between GPSS and Kafka.
- GPSS supports SSL encryption on the data channel between GPSS and Greenplum Database.

Resolved Issues

Greenplum Streaming Server 1.3.0 is a minor release that resolves these issues:

168130147
: In some situations, specifying the `--force-reset-earliest` flag when loading data failed to read from the correct offset. This problem has been fixed. (Using the `--force-reset-*xxx*` flags outside of an offset mismatch scenario is discouraged.)

167997441
: GPSS did not save error data to the external table error log when it encountered an incorrectly-formatted JSON or Avro message. This issue has been fixed; invoking `gp_read_error_log()` on the external table now displays the offending data.

164823612
: GPSS incorrectly treated Kafka jobs that specified the same Kafka topic and Greenplum output schema name and output table name, but different database names, as the same job. This issue has been resolved. GPSS now includes the Greenplum database name when constructing a job definition.

Beta Features

Greenplum Streaming Server 1.x includes these *Beta* features:

- GPSS adds support for a RabbitMQ data source (introduced in 1.8.0, promoted to supported in 1.9.0).
- GPSS adds support for an `s3` data source (introduced in 1.7.0).
- GPSS adds a new datatype named `gp_json` to the `dataflow` extension (introduced in 1.7.0).
- GPSS exposes a new load configuration property for Kafka data sources named `RECOVER_FAILING_BATCH` (version 2 configuration) and `recover_failing_batch` (version 3 configuration). Use this property in conjunction with `SAVE_FAILING_BATCH` to instruct GPSS to automatically reload the good data in the batch, and retain only the error data in the backup table.

  Note: Enabling this feature may have severe performance implications when any data in the Kafka topic generates an expression error.

  Note: This feature requires that GPSS has the Greenplum Database privileges to create a function.

  (Introduced in 1.6.0.)
- GPSS adds a new extension named `dataflow`. This extension includes a new data type, `gp_jsonb` (available for Greenplum Database version 6.x only), and a new formatter, `text_in`. (Introduced in 1.6.0).
- GPSS specifies a new version 3 load configuration file format. This format introduces a new YAML organization and keywords. (Introduced in 1.5.0.)

Deprecated Features

Deprecated features may be removed in a future release of the Greenplum Streaming Server. GPSS 1.x deprecates:

- Specifying the `gpss.json` configuration file to the `gpss` command standalone (deprecated since 1.6.0). Use the `-c | --config` option when you specify the file.
- The `gpkafka` Version 1 configuration file format (deprecated since 1.4.0).
- The `gpkafka.yaml` (versions 1 and 2) `POLL` block, including the `POLL:BATCHSIZE` and `POLL:TIMEOUT` properties (deprecated since 1.3.5).

Known Issues and Limitations

Greenplum Streaming Server 1.x has these known issues:

31998
: In some cases, an `EXPLAIN INSERT` command internally launched by GPSS on a Kafka job may take a long time to complete. You can work around this issue by specifying the `--skip-explain` flag to the gpsscli start command when you start the job.

N/A
: The `SAVE_FAILING_BATCH` and `PARTITIONS` configuration properties are not supported when you use the version 1 configuration file format to load data.

N/A

The Greenplum Streaming Server may consume a very large amount of system memory when you use it to load a huge (hundreds of GBs) file, in some cases causing the Linux kernel to kill the GPSS server process. Do not use GPSS to load very large files; instead, use `gpfdist`.

30503

Due to limitations in the Greenplum Database external table framework, GPSS cannot log a data type conversion error that it encounters while evaluating a mapping expression. For example, if you use the expression `EXPRESSION: (jdata->>'id')::int` in your load configuration file, and the content of `jdata->>'id'` is a string that includes non-integer characters, the evaluation fails and GPSS terminates the load job. GPSS cannot log and propagate the error back to the user via `gp_read_error_log()`.

Workarounds for Kafka:

- Set the `SAVE_FAILING_BATCH` load configuration property to `true`, and then manually load any data batch that included expression errors.
- Skip the bad Kafka message by specifying a `--force--reset-*xxx*` flag on the job start or load command.
- Correct the message and publish it to another Kafka topic before loading it into VMware Greenplum.

Overview of the Greenplum Streaming Server

The Greenplum Streaming Server (GPSS) is an ETL (extract, transform, load) tool. An instance of the GPSS server ingests streaming data from one or more clients, using Greenplum Database readable external tables to transform and insert the data into a target Greenplum table. The data source and the format of the data are specific to the client.

The Greenplum Streaming Server includes the gpss command-line utility. When you run `gpss`, you start an instance of GPSS; this instance waits indefinitely for client data.

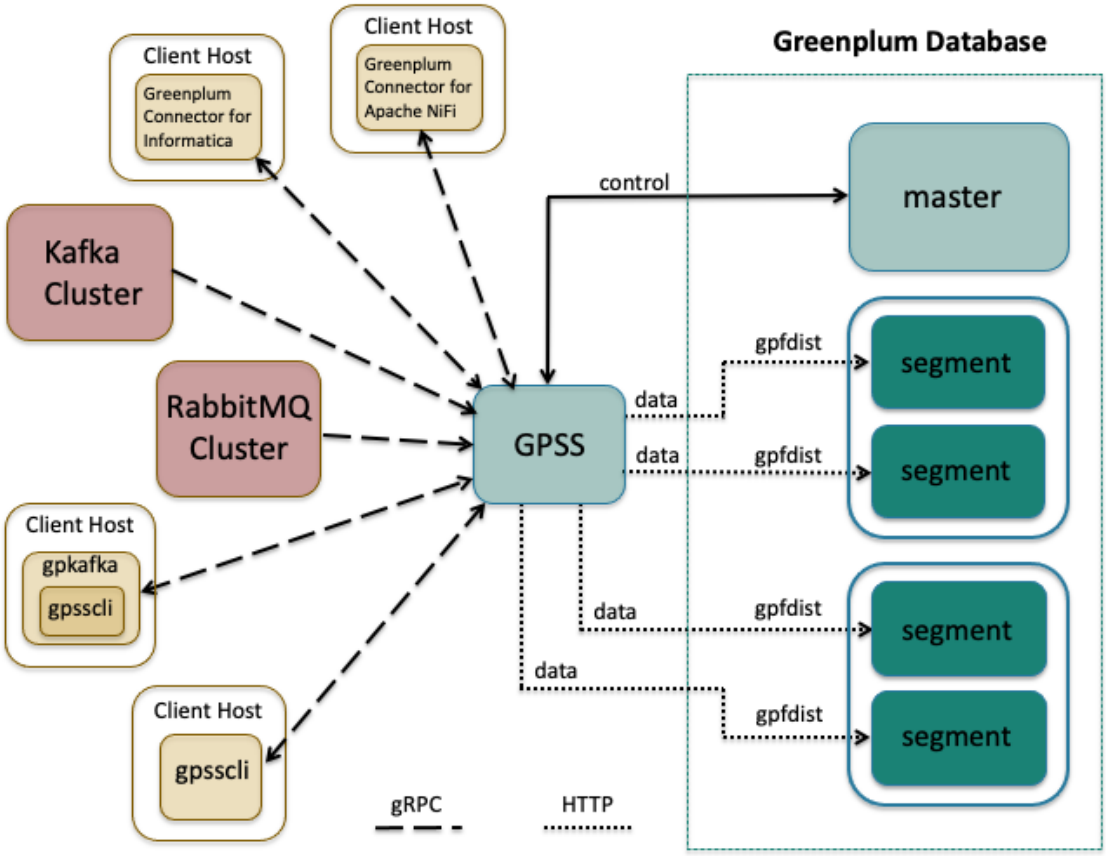
The Greenplum Streaming Server also includes the gpsscli command-line utility, a client tool for submitting data load jobs to a GPSS instance and managing those jobs.

The Greenplum Streaming Server `gpsscli` client utility currently supports Kafka, file, RabbitMQ, and S3 (Beta) data sources.

Architecture

The Greenplum Streaming Server is a gRPC server. The GPSS gRPC service definition includes the operations and messages necessary to connect to Greenplum Database and examine Greenplum metadata. The service definition also includes the operations and messages necessary to write data from a client into a Greenplum Database table. For more information about gRPC, refer to the gRPC documentation.

The `gpsscli` utility is a Greenplum Streaming Server gRPC client, as is the VMware Greenplum Connector for Apache NiFi. You can develop your own GPSS gRPC client using the GPSS Batch Data API.



A typical sequence of events for performing an ETL task using the Greenplum Streaming Server follows:

1. A user initiates one or more ETL load jobs via a client application.
2. The client application uses the gRPC protocol to submit and start data load job(s) to a running GPSS service instance.
3. The GPSS service instance submits each load request transaction to the Greenplum Database cluster coordinator instance. GPSS uses the `gpfdist` protocol to store data in external tables that it creates or reuses.
4. The GPSS service instance writes the data delivered from the client directly into the segments of the Greenplum Database cluster.

Installing the Streaming Server

The Greenplum Streaming Server (GPSS) components are included in the VMware Greenplum 5 and 6 server distributions. If you want to install the newest version of these components, you may be required to download a package from the *VMware Greenplum Streaming Server* tile on Broadcom Support Portal.

The GPSS packages available for Greenplum 5/6/7 for download include:

- *GPSS gppkg Installer* - A `.gppkg` file that you install to upgrade GPSS on all hosts in your Greenplum Database cluster.
- *GPSS ETL Installer* - An `.rpm` or `.deb` file that you use to install or upgrade GPSS on a dedicated ETL server host with no Greenplum Database bits installed.
- *GPSS tarball* - A `.tar.gz` file that you install to upgrade GPSS on a single dedicated ETL server host that includes a Greenplum Database server installation.

Refer to the Supported Platforms section in the *Release Notes* to determine specific Greenplum Database and operating system version support for GPSS.

About the Download Packages

The *GPSS gppkg Installer* and the *GPSS tarball* package files install the libraries, executables, and script files required to register and use the Greenplum Streaming Server client and server utilities directly into your Greenplum Database installation.

The *GPSS ETL Installer* package file installs the client side executables and dependent libraries, and a script to set up the ETL runtime environment.

Name	Description
gpkafka	Load Kafka data into Greenplum Database using a single command.
gpss	Start a VMware Greenplum Streaming Server instance.
gpsscli	Manage (submit, start, stop, and so forth) a VMware Greenplum Streaming Server data load job; currently supports Kafka, file, RabbitMQ, and S3 (Beta) data sources.
kafkacat	https://github.com/edenhill/kafkacat Kafka test and debug utility.

Downloading a GPSS Installer

Download the appropriate GPSS installer package for your VMware Greenplum version and operating system platform from Broadcom Support Portal. For example, to download the Greenplum 6 `.gppkg` package for Red Hat/CentOS 7, click to select the **RHEL7->GPSS gppkg Installer for GPDB6 RHEL7** file.

For more information about download prerequisites, troubleshooting, and instructions, see Download Broadcom products and software.

The naming format of the GPSS installer files is:

```
gpss-gpdb<major-version>-<gpss-version>-<platform>-x86_64.<filetype>
```

For example, the GPSS installer files for Greenplum Database 5 for Red Hat/CentOS 6 are named:

```
gpss-gpdb5-1.10.4-rhel6-x86_64.gppkg
gpss-gpdb5-1.10.4-rhel6-x86_64.tar.gz
gpss-gpdb5-1.10.4-rhel6-x86_64.rpm
```

The GPSS installer files for Greenplum Database 6 for Red Hat 7 are named:

```
gpss-gpdb6-1.10.4-rhel7-x86_64.gppkg
gpss-gpdb6-1.10.4-rhel7-x86_64.tar.gz
gpss-gpdb6-1.10.4-rhel7-x86_64.rpm
```

The GPSS installer files for Greenplum Database 6 for Red Hat 8 are named:

```
gpss-gpdb6-1.10.4-rhel8-x86_64.gppkg
gpss-gpdb6-1.10.4-rhel8-x86_64.tar.gz
gpss-gpdb6-1.10.4-rhel8-x86_64.rpm
```

The GPSS installer files for Greenplum Database 6 for Photon 3 are named:

```
gpss-gpdb6-1.10.4-photon3-x86_64.gppkg
gpss-gpdb6-1.10.4-photon3-x86_64.tar.gz
gpss-gpdb6-1.10.4-photon3-x86_64.rpm
```

The GPSS installer files for Greenplum Database 6 for Ubuntu 18.04 are named:

```
gpss-gpdb6-1.10.4-ubuntu18.04-x86_64.gppkg
gpss-gpdb6-1.10.4-ubuntu18.04-x86_64.tar.gz
gpss-gpdb6-1.10.4-ubuntu18.04-x86_64.deb
```

The GPSS installer files for Greenplum Database 6 for Rocky Linux 9 are named:

```
gpss-gpdb6-1.10.4-rocky9-x86_64.gppkg
gpss-gpdb6-1.10.4-rocky9-x86_64.tar.gz
gpss-gpdb6-1.10.4-rocky9-x86_64.rpm
```

The GPSS installer files for Greenplum Database 7 for Red Hat 9 are named:

```
gpss-gpdb7-1.10.4-rhel9-x86_64.gppkg
gpss-gpdb7-1.10.4-rhel9-x86_64.tar.gz
gpss-gpdb7-1.10.4-rhel9-x86_64.rpm
```

The GPSS installer files for Greenplum Database 7 for OEL 8 are named:

```
gpss-gpdb7-1.10.4-oel8-x86_64.gppkg
gpss-gpdb7-1.10.4-oel8-x86_64.tar.gz
gpss-gpdb7-1.10.4-oel8-x86_64.rpm
```

The GPSS installer files for Greenplum Database 7 for OEL 9 are named:

```
gpss-gpdb7-1.10.4-oel9-x86_64.gppkg
gpss-gpdb7-1.10.4-oel9-x86_64.tar.gz
gpss-gpdb7-1.10.4-oel9-x86_64.rpm
```

The GPSS installer files for Greenplum Database 7 for Rocky Linux 8 are named:

```
gpss-gpdb7-1.10.4-rocky8-x86_64.gppkg
gpss-gpdb7-1.10.4-rocky8-x86_64.tar.gz
gpss-gpdb7-1.10.4-rocky8-x86_64.rpm
```

Note the name and the file system location of the downloaded file.

Follow the instructions in Verifying the VMware Greenplum Software Download to verify the integrity of the *Greenplum Streaming Server* software.

Prerequisites

Before you install a GPSS package, ensure that you have stopped all Greenplum Streaming Server load jobs and `gpss` server instances running in the Greenplum Database cluster and on the ETL host system.

Installing the GPSS gppkg

The *GPSS gppkg Installer* updates the GPSS components on all hosts in the Greenplum Database cluster.

The GPSS executables, libraries, and supporting files are installed directly into `$GPHOME`, overwriting the previous versions of the files.

Perform the following procedure to install the GPSS `.gppkg`:

1. Locate the installer file that you downloaded from Broadcom Support Portal and copy the file to the VMware Greenplum coordinator host.
2. Log in to the VMware Greenplum coordinator host as the `gpadmin` administrative user and set up your environment. For example:

   ```
   $ ssh gpadmin@<gpcoord>
   gpadmin@gpcoord$ . /usr/local/greenplum-db/greenplum_path.sh
   ```

3. Ensure that Greenplum Database is running.
4. Run the `gppkg` command to install the GPSS `.gppkg` on all hosts in the Greenplum Database cluster.

   For example, *to install the package on a Greenplum 6 cluster running on Red Hat/CentOS 7*:

```
$ gppkg -i gpss-gpdb6-1.10.4-rhel7-x86_64.gppkg
```

To install the package on a Greenplum 7 cluster running on Red Hat/CentOS 8:

```
$ gppkg install gpss-gpdb7-1.10.4-rhel8-x86_64.gppkg
```

Installing the GPSS Tarball

The *GPSS tarball* `.tar.gz` installer updates the GPSS components on a single Greenplum Database host.

The GPSS executables, libraries, and supporting files are installed directly into `$GPHOME`, overwriting the previous versions of the files.

Perform the following procedure to install the GPSS `.tar.gz`:

1. Locate the installer file that you downloaded from Broadcom Support Portal and copy the file to the VMware Greenplum host.
2. Log in to the VMware Greenplum host as the `gpadmin` administrative user and set up your environment. For example:

   ```
   $ ssh gpadmin@<gphost>
   gpadmin@gphost$ . /usr/local/greenplum-db/greenplum_path.sh
   ```

3. Unpack the `.tar.gz` file. For example, to unpack the file for Greenplum 5 on Red Hat/CentOS 6:

   ```
   gpadmin@gphost$ tar xzvf gpss-gpdb5-1.10.4-rhel6-x86_64.tar.gz
   ```

   Unpacking the file creates a directory named `gpss-gpdb5-1.10.4-rhel6_x86_64/` in the current working directory. Its contents include `bin/`, `lib/`, and `share/` directories, as well as an install script named `install_gpdb_component`.

4. Navigate to the unpacked directory. For example:

   ```
   gpadmin@gphost$ cd gpss-gpdb5-1.10.4-rhel6_x86_64
   ```

5. Run the install script to install the new GPSS components into `$GPHOME`. For example:

   ```
   gpadmin@gphost$ ./install_gpdb_component
   ```

Installing the GPSS ETL Package

The *GPSS ETL Installer* installs the GPSS executables, libraries, and supporting files on a single ETL host.

Perform the following procedure to install the GPSS ETL package:

1. Locate the installer file that you downloaded from Broadcom Support Portal and copy the file to the ETL host.
2. Log in to the ETL host. For example:

```
$ ssh <etluser>@<etlhost>
```

3. Install the package using your package management utility. You must be the superuser or have `sudo` access to install packages.

   For example, to install the ETL package for Greenplum 6 on Red Hat/CentOS 7:

   ```
   etluser@etlhost$ sudo yum install gpss-gpdb6-1.10.4-rhel7-x86_64.rpm
   ```

   To install the ETL package for Greenplum 6 on Ubuntu 18.04:

   ```
   etluser@etlhost$ sudo dpkg --install gpss-gpdb6-1.10.4-ubuntu18.04-x86_64.deb
   ```

   The GPSS ETL tools are installed into the `/usr/local/gpss-<version>` directory. The installation process creates a symbolic link from `/usr/local/gpss` to this install directory.

4. Before using the GPSS ETL tools, you must first source the `gpss_path.sh` environment file:

   ```
   etluser@etlhost$ . /usr/local/gpss/gpss_path.sh
   ```

Upgrading the Streaming Server

If you are using the Greenplum Streaming Server (GPSS) in your current Greenplum Database installation, you must perform the GPSS upgrade procedure when:

- You upgrade to a newer version of Greenplum Database, or
- You install a new standalone GPSS package on your ETL host or in your Greenplum Database installation.

The GPSS upgrade procedures describe how to upgrade GPSS in your Greenplum Database installation or on your ETL host. This procedure uses *GPSS.from* to refer to your currently-installed GPSS and *GPSS.new* to refer to the GPSS installed when you upgrade to the new version of Greenplum Database or install a new GPSS package.

The GPSS upgrade procedure has two parts. You perform one procedure before, and one procedure after, you upgrade to a new version of Greenplum Database or GPSS:

- Step1: GPSS Pre-Upgrade Actions
- Upgrade to a new Greenplum Database version or install a new GPSS package.
- Step2: Upgrading GPSS

Step1: GPSS Pre-Upgrade Actions

Perform this procedure in your *GPSS.from* installation before you upgrade to a new version of Greenplum Database or GPSS:

1. Log in to the Greenplum Database coordinator host or the ETL host and set up your environment. For example:

   ```
   $ ssh gpadmin@<gpcoord>
   gpadmin@gpcoord$ . /usr/local/greenplum-db/greenplum_path.sh
   ```

   Or:

   ```
   $ ssh etluser@<etlhost>
   etluser@etlhost$ . /usr/local/gpss/gpss_path.sh
   ```

2. Identify and note the current version (*GPSS.from*) of GPSS. For example:

   ```
   $ gpss --version
   ```

3. Stop all `gpss` jobs that are in the *Running* state.
4. Stop all running `gpss` instances.

5. Upgrade to the new version of Greenplum Database or install a new version of GPSS, and then continue your GPSS upgrade with Step2: Upgrading GPSS.

Step2: Upgrading GPSS

After you upgrade to the new version of Greenplum Database or install the new version of GPSS in your Greenplum installation, perform the following procedure to upgrade the *GPSS.new* software:

1. Log in to the Greenplum Database coordinator host or the ETL host and set up your environment. For example, on the coordinator:

   ```
   $ ssh gpadmin@<gpcoord>
   gpadmin@gpcoord$ . /usr/local/greenplum-db/greenplum_path.sh
   ```

2. Identify and note the new version (*GPSS.new*) of GPSS. For example:

   ```
   gpadmin@gpcoord$ gpss --version
   ```

3. **If you are upgrading *from* GPSS version 1.3.0 or older**:

   GPSS 1.3.0 introduced a regression that caused it to no longer recognize history tables (internal tables that GPSS creates for each job) that were created with GPSS 1.2.6. This regression could cause GPSS to load duplicate Kafka messages into Greenplum. This issue is resolved in GPSS 1.3.1.

   You are not required to perform any upgrade steps related to this issue; GPSS will automatically perform the required actions when you resubmit and restart a load job that you initiated with GPSS 1.3.0. GPSS's upgrade actions are dependent upon the GPSS version(s) from which you are upgrading, and are described below:

   - *If you are upgrading directly from GPSS 1.2.6 or older*, GPSS performs no special upgrade actions.
   - *If you are upgrading from GPSS 1.3.0 and you previously submitted load jobs with both GPSS 1.2.6 or older and 1.3.0*, GPSS copies the internal history table for each submitted job to a table with the correct name format, and uses those tables. GPSS also retains and renames the internal history table for each GPSS 1.3.0 job, adding the prefix `deprecated_`.
   - *If you first and only used GPSS 1.3.0 and are upgrading from this version*, GPSS renames the internal history table for each restarted job.

4. **If you are upgrading *from* GPSS version 1.3.1 or older**:

   - GPSS 1.3.2 changes the `gpss.json` configuration file:
     - The new file format allows you to specify unique SSL `Certificate`s for GPSS and `gpfdist`. If you are using SSL to encrypt communication between GPSS and Kafka, Greenplum, or the GPSS client, you must update the `gpss.json` server configuration file to configure the correct `Certificate` block.
     - The `ListenAddress:SSL` property is removed. Ensure that you remove this property from all GPSS server configuration files.
   - GPSS 1.3.2 renames `gpkafka check` to `gpkafka history`. If you have any scripts or programs that reference `gpkafka check`, you must replace these references with `gpkafka`

`history`.

- GPSS 1.3.2 removes the `ENCRYPTION` property from the `gpkafka.yaml` job configuration file. Ensure that you remove this property from all job configuration files, and that you provide Kafka SSL configuration properties via the `PROPERTY` block in the file.
- GPSS 1.3.2 removes the `LOCAL_HOSTNAME` and `LOCAL_PORT` properties from the `gpkafka.yaml` job configuration file. You must remove these properties from all job configurations, and specify the `gpfdist` configuration for each job in one of the following ways:
  - If you are loading data with `gpkafka load`, provide the `--config gpfdistconfig.json` or `--gpfdist-host hostaddr` and `--gpfdist-port portnum` options when you run the command.
  - If you are loading data with the `gpsscli` job management commands, ensure that the `gpss.json` configuration file for the `gpss` server instance servicing the request specifies the desired `Gpfdist:Host` and `Gpfdist:Port` settings.
- GPSS 1.3.2 removes the `--no-reuse` flag from the `gpsscli load` and `gpsscli start` commands. If you have any scripts or programs that reference this flag, you must remove the references.

5. **If you developed a client application with GPSS 1.3.5 or earlier** and you want to use the new `MaxErrorRows` or `Abort` session capabilities added to the `Close` service that were introduced in GPSS 1.3.6, you must:
   1. Edit the `gpss.proto` service definition and add the new `CloseRequest` field(s):

      ```
      message CloseRequest {
        Session session = 1;
        int32 MaxErrorRows = 2;
        bool Abort = 3;
      }
      ```

   2. Re-generate the GPSS client classes.
   3. Add code to utilize the new fields.
   4. Re-compile and re-distribute your GPSS client application. Refer to Developing a Batch Data Client for supporting information.
6. **If you are upgrading *from* GPSS version 1.4.x or older**:
   - GPSS 1.4.0 removes the `gpsscli history` and `gpkafka history` commands. If you have any scripts or programs that reference these commands, you must remove the references.
   - GPSS 1.4.1 changes the client and server log file format to CSV. If you created any scripts that parsed the previous log file format, you must update that script logic.
   - GPSS 1.4.1 adds a new, separate logfile to track Kafka job progress. If you created any scripts that relied on the existence of progress information in the client or server log files, you must update that script logic.

7. **If you are upgrading *from* GPSS version 1.6.x or older** and you have registered the `dataflow` extension in any database, you must drop and re-create the extension:

    ```
    DROP EXTENSION dataflow;
    CREATE EXTENSION dataflow;
    ```

8. **If you are upgrading *from* GPSS version 1.7.x or older**:

    - GPSS 1.8.0 changes the name of the Kafka version 3 (Beta) load configuration file `window` property to `task`. If you have any Kafka load configuration files that specify `window:`, you must change the references to `task:`.

9. **If you are upgrading *from* GPSS version 1.9.x or older**:

    - GPSS 1.10.0 changes the naming format of its server log files as described in the Version 1.10.0 release notes and adds a `job_id` field to the content of the server log file. You must update any scripts that you have written that rely on the log file naming format or the log file content of previous releases.

10. **If you developed a client application with GPSS 1.9.x or earlier** and you want to use the new session timeout capability added to the `Connect` service that was introduced in GPSS 1.10.0, you must:

    1. Edit the `gpss.proto` service definition and add the new `SessionTimeout` field to the `ConnectRequest` message:

        ```
        message ConnectRequest {
          string Host = 1;
          ...
          bool UseSSL = 6;
          int32 SessionTimeout = 7;
        }
        ```

    2. Re-generate the GPSS client classes.
    3. Add code to utilize the new field.
    4. Re-compile and re-distribute your GPSS client application. Refer to Developing a Batch Data Client for supporting information.

11. **If you are upgrading *from* GPSS version 1.10.0**:

    - GPSS 1.10.1 changes the naming format of its per-run server log files as described in the Version 1.10.1 release notes. You must update any scripts that you have written that rely on the per-run server log file naming format introduced in version 1.10.0.

12. If you installed a new version of Greenplum Database, or you installed the GPSS `gppkg` or `.tar.gz` packages in your Greenplum installation, you must drop and re-create the GPSS extension in any Greenplum database in which you are using GPSS to load data. A database superuser or the database owner must run these SQL commands:

    ```
    DROP EXTENSION gpss;
    CREATE EXTENSION gpss;
    ```

(If the extension does not already exist, GPSS automatically creates it in a database the first time a Greenplum superuser or the database owner submits a load job to any table that resides in that database.)

13. Restart your `gpss` instances.
14. Resubmit and restart your GPSS jobs.

    For any Kafka job that you resubmit and restart, GPSS will consume Kafka messages from the offset associated with the latest timestamp recorded in the history table for the job.

Configuring and Managing the Streaming Server

The Greenplum Streaming Server (GPSS) manages communication and data transfer between a client (for example, the VMware Greenplum Connector for Apache NiFi) and Greenplum Database. You must configure and start a GPSS instance before you use the service to load data into Greenplum Database.

Prerequisites

The Greenplum Streaming Server `gpss` and `gpsscli` command line utilities are automatically installed with Greenplum Database version 5.16 and later.

Before you start a GPSS server instance, ensure that you:

- Install and start a compatible Greenplum Database version.
- Can identify the hostname of your coordinator node.
- Can identify the port on which your Greenplum Database coordinator server process is running, if it is not running on the default port (5432).
- Select one or more GPSS host machines that have connectivity to:
    - The GPSS client host systems.
    - The Greenplum Database coordinator and all segment hosts.

If you are using the `gpsscli` client utility, ensure that you run the command on a host that has connectivity to:

- The client data source host systems. For example, for a Kafka data source, you must have connectivity to each broker host in the Kafka cluster.
- The Greenplum Database coordinator and all segment hosts.

Registering the GPSS Extension

The Greenplum Database and the Greenplum Streaming Server download packages install the GPSS extension. This extension must be registered in each database in which Greenplum users use GPSS to write data to Greenplum tables.

GPSS automatically registers its extension in a database the first time a Greenplum superuser or the database owner initiates a load job. You must manually register the extension in a database if non-privileged Greenplum users will be the first or only users of GPSS in that database.

Perform the following procedure as a Greenplum Database superuser or the database owner to manually register the GPSS extension:

1. Open a new terminal window, log in to the Greenplum Database coordinator host as the `gpadmin` administrative user, and set up the Greenplum environment. For example:

   ```
   $ ssh gpadmin@gpcoord
   gpadmin@gpcoord$ . /usr/local/greenplum-db/greenplum_path.sh
   ```

2. Start the `psql` subsystem, connecting to a database in which you want to register the GPSS formatter function. For example:

   ```
   gpcoord$ psql -d testdb
   ```

3. Enter the following command to register the extension:

   ```
   testdb=# CREATE EXTENSION gpss;
   ```

4. Perform steps 2 and 3 for each database in which the Greenplum Streaming Server will write client data.

Configuring the Greenplum Streaming Server

You configure an invocation of the Greenplum Streaming Server via a JSON-formatted configuration file. This configuration file includes properties that identify the listen address of the GPSS service and an optional debug server port number, as well as the `gpfdist` service host, bind address, and port number. You can specify encryption options in the file, can configure a password shadow encode/decode key, and can also configure whether GPSS reuses external tables.

The contents of a sample GPSS JSON configuration file named `gpsscfg1.json` follow:

```
{
    "ListenAddress": {
        "Host": "",
        "Port": 5019,
        "DebugPort": 9998
    },
    "Gpfdist": {
        "Host": "",
        "Port": 8319,
        "ReuseTables": false
    },
    "Shadow": {
        "Key": "a_very_secret_key"
    }
}
```

Refer to the gpss.json reference page for detailed information about the GPSS configuration file format and the configuration properties that the utility supports.

If your Kafka or Greenplum Database clusters are using Kerberos authentication or SSL encryption, see Configuring the Streaming Server for Encryption and Authentication.

Refer to Configuring the Streaming Server for Client-to-Server Authentication for information about configuring client authentication for GPSS.

Running the Greenplum Streaming Server

You use the `gpss` utility to start an instance of the Greenplum Streaming Server on the local host. When you run the command, you provide the name of the configuration file that defines the properties of the GPSS and `gpfdist` service instances. You can also specify the name of a directory to which `gpss` writes server and progress log files. For example, to start a GPSS instance specifying a log directory named `gpsslogs` relative to the current working directory:

```
$ gpss --config gpsscfg1.json --log-dir ./gpsslogs
```

By default, GPSS waits 10 seconds to establish a connection to Greenplum Database. If GPSS does not establish the connection in that time, a time out error is displayed and the `gpss` command returns. You can change the time out value by setting the `GPDB_CONNECT_TIMEOUT` environment variable before or when you start GPSS. For example, to set the timeout to 30 seconds, start the GPSS instance as follows:

```
$ GPDB_CONNECT_TIMEOUT=30 gpss --config gpsscfg1.json --log-dir ./gpsslogs
```

The default mode of operation for `gpss` is to wait for, and then consume, job requests and data from a client. When run in this mode, `gpss` waits indefinitely. You can interrupt and exit the command with Control-c. You may also choose to run `gpss` in the background (`&`). In both cases, `gpss` writes server log and status messages to `stdout`.

`gpss` keeps track of the status of each client job in memory. When you stop a GPSS server instance that did not specify a `JobStore` setting in its server configuration file, you lose all registered jobs. You must re-submit any previously-submitted jobs that you require after you restart the server instance. `gpss` will resume a job from the last load offset.

Refer to the gpss reference page for additional information about this command.

About GPSS Logging

The `gpss`, `gpsscli`, and `gpkafka` commands each write log messages to `stdout` (front-end) and to a log file (back-end). These messages provide useful information about GPSS command processing and any errors that it encounters.

GPSS supports the following log levels, listed in order from most to least severe:

Level	Description
fatal	Logs conditions that prevent GPSS from functioning, such as being unable to listen on the configured port.
error	Logs job failure messages.
warn or warning	Logs messages that contain information that requires the user's attention, such as the use of deprecated features or notification of job retry in progress.

Level	Description
info	Logs messages that contain information about GPSS actions, including job status changes and requests between the GPSS client and server.
debug	Logs more detailed and more verbose messages that may aid in troubleshooting.

The default log level for command front-end messages to `stdout` is `info`. The default log level for back-end messages that the commands write to the log file is `debug`.

You can change the front-end or back-end log level by specifying a `Logging` block in the gpss.json GPSS server configuration file and setting the appropriate property:

```
"Logging": {
    "Backendlevel": "<level>",
    "Frontendlevel": "<level>"
}
```

The default format for command front-end messages that GPSS writes to `stdout` uses spaces between fields. You can provide options to commands to instruct GPSS to write the front-end messages in CSV format, or to use color in the message. The default format for back-end messages that GPSS writes to the log file is CSV format.

Managing GPSS Log Files

If you specify the `-l` or `--log-dir` option when you start `gpss` or run a `gpsscli` subcommand, GPSS writes log messages to a file in the directory that you specify. If you do not provide this option, GPSS writes log messages to a file in the `$HOME/gpAdminLogs` directory.

By default, GPSS writes server log messages to a file with the following naming format:

```
gpss_<timestamp-with-millis>.log
```

Where `<timestamp-with-millis>` identifies the date and time (with milliseconds) that the log file was created. This date reflects the day/time that you started the `gpss` server instance, or the day/time that the log was rotated for that server instance (see Rotating the GPSS Server Log File below):

GPSS writes client log messages to a file with the following naming format, where `<date>` identifies the date that you ran the command:

```
gpsscli_<date>.log
```

GPSS writes progress messages for each Kafka job to a separate file in the server log directory. Progress logs are written to a file with this naming format:

```
progress_<jobname>_<jobid>_<date>.log
```

`<jobname>` and `<jobid>` (max 8 characters each) identify the name and the identifier of the GPSS job, and `<date>` identifies the date that you ran the command.

Example GPSS log file names:

- `gpss_23-04-27_151722.950.log`

- `gpsscli_230427.log`
- `progress_jobk2_d577cf37_20200803.log`

Configuring Per-Run Server Log Files

You can set the `Logging:SplitByJob` property in the gpss.json server configuration file to direct GPSS to generate per-run server log files for each job:

```
"Logging": {
    "SplitByJob": "<level>"
}
```

The only valid `<level>` is `StartTime`.

When you specify `"SplitByJob": "StartTime"` in the server configuration file, GPSS creates, for every job, a new log file in the server log directory each time the job is started (`gpsscli start`) or loaded (`gpsscli load`). GPSS creates the log file regardless of the success or failure of the start or load job operation.

When you set `SplitByJob`, the server log file name will also include the job name: `gpss_<jobname>_<timestamp-with-millis>.log`. For example:

```
gpss_nightly_load-23-05-11_104800.123.log
```

When the job is stopped (`gpsscli stop`), GPSS logs to the most recently created log file for the specified job.

Rotating the GPSS Server Log File

If the log file for a `gpss` server instance grows too large, you may choose to archive the current log and start fresh with an empty log file.

There are two ways to rotate GPSS server logs:

1. Configure GPSS to automatically rotate the server logs.
2. You initiate server log rotation on-demand.

Configuring Automatic Server Log File Rotation

You can set the `Logging:Rotate` property in the gpss.json server configuration file to direct GPSS to automatically rotate the server log files:

```
"Logging": {
    "Rotate": "<policy_period>"
}
```

Valid `<policy_period>`s are `daily` and `hourly`.

When you specify a `Logging:Rotate` property setting in the server configuration file, GPSS automatically rotates the server log file for you at the end of the policy period (hour, day) that you specify. If you stop the server instance, a new invocation restarts the time period.

Rotating the Server Log File On-Demand

To prompt GPSS to rotate the *server* log file on-demand, you must:

1. Rename the existing log file. For example:

    ```
    gpadmin@gpcoord$ mv <logdir>/gpss_<timestampmillis>.log <logdir>/gpss_<timestam
    pmillis>.log.1
    ```

2. Send the `SIGUSR2` signal to the `gpss` server process. You can obtain the process id of a GPSS instance by running the `ps` command. For example:

    ```
    gpadmin@gpcoord$ ps -ef | grep gpss
    gpadmin@gpcoord$ kill -SIGUSR2 <gpss_pid>
    ```

    There may be more than one `gpss` server process running on the system. Be sure to send the signal to the desired process.

    When it receives the signal, GPSS emits a log message that identifies the time at which it reset the log file. For example:

    ```
    ... -[INFO]:- gpss log file rotate at 20230411:20:59:36.093
    ```

Integrating with logrotate

You can configure and manage GPSS server log file rotation with the Linux `logrotate` utility.

This sample `logrotate` configuration rotates and compresses the log file of each `gpss` server instance running on the system weekly or when the file reaches 10MB in size. It operates on all-in-one server log files that are written to the default location:

```
/home/gpadmin/gpAdminLogs/gpss_*.log {
    rotate 5
    weekly
    size 10M
    postrotate
        pkill -SIGUSR2 gpss
    endscript
    compress
}
```

If this configuration is specified in a file named `gpss_rotate.conf` residing in the current working directory, you integrate with the Linux `logrotate` system with the following command:

```
$ logrotate -s status -d gpss_rotate.conf
```

You may choose to create a `cron` job to run this command daily.

Monitoring GPSS Service Instances

GPSS provides out-of-the-box integration with the Prometheus open-source system monitoring and alerting toolkit. Refer to Enabling Prometheus Metrics Collection for information about enabling and using this integration.

About GPSS Job Management

When you submit a GPSS job, you provide a name/identifier for the job. If you do not specify a job name, GPSS assigns and returns the base name of the load configuration file as the job name. You use this name to manage the job throughout its lifecycle.

GPSS uses a data source-specific combination of properties specified in a load configuration file to internally identify a job. For example, when it loads from a Kafka data source, GPSS uses the Kafka topic name, and the target Greenplum database, schema, and table names for internal job identification. GPSS creates internal and external tables for each job that are keyed off of these properties; these tables keep track of the progress of the load operation. GPSS considers any load configuration file submitted with the same value for these job-identifying properties to be the same internal job.

A `gpss` server instance keeps track of the status of each client job in memory. By default, this information is invocation-specific. When you stop the server instance, you must re-submit any job that you want to run when you next restart the instance.

You can configure the `JobStore` property block in the gpss.json server configuration file to instruct GPSS to retain job and job status information across invocations.

```
"JobStore": {
    "File": {
        "Directory": "<jobstore_dir>"
    }
}
```

When you specify a `JobStore:File:Directory` property setting in the server configuration file, the GPSS server instance keeps track of, and writes job information to, the directory that you specify. If you stop the server instance, a new invocation will restore the jobs that were in progress when it last exited, loading the jobs in memory and restoring their last known state.

Shadowing the Greenplum Database Password

When you use GPSS to load data into Greenplum Database, you specify the Greenplum user/role password in the `PASSWORD:` property setting of a YAML-format load configuration file; see gpsscli.yaml.

You specify the Greenplum password in clear text. If your security requirements do not permit this, you can configure GPSS to encode and decode a shadow password string that the GPSS client and server use when communicating the Greenplum password.

GPSS supports shadowing the Greenplum password only on load jobs that you submit and manage with the `gpsscli` subcommands. GPSS does not support shadowed passwords on load jobs that you submit with `gpkafka load`.

When you use this GPSS feature:

1. (Optional) You configure a `Shadow:Key` in the gpss.json configuration file that you specify when you start the GPSS instance. For example:

    ```
    ...
        },
        "Shadow": {
            "Key": "a_very_secret_key"
        }
    ...
    ```

2. You run the gpsscli shadow command on the ETL system to interactively generate the shadowed password. For example:

    ```
    $ gpsscli shadow --config gpss.json
    please input your password
    changemeCHANGEMEchangeme
    "SHADOW:ERTBKXDWLAJHUF5UOGJY34QTXIBNYP4ULTWVHIUZIF4UYFPRIJVA"
    ```

    You can automate this step using a command similar to the following:

    ```
    $ echo changemeCHANGEMEchangeme | gpsscli shadow --config gpss.json | tail -1
    "SHADOW:ERTBKXDWLAJHUF5UOGJY34QTXIBNYP4ULTWVHIUZIF4UYFPRIJVA"
    ```

    If you do not specify the `--config gpss.json` option, or this configuration file does not include a `Shadow:Key` setting, GPSS uses its default key to generate the shadow password string.

3. You specify the shadow password string returned by `gpsscli shadow` in the `PASSWORD:` property setting of a gpsscli.yaml load configuration file. For example:

    ```
    DATABASE: testdb
    USER: testuser
    PASSWORD: "SHADOW:ERTBKXDWLAJHUF5UOGJY34QTXIBNYP4ULTWVHIUZIF4UYFPRIJVA"
    ...
    ```

    Always quote the complete shadow password string.

4. You provide the load configuration file as an option to gpsscli submit or gpsscli load when you submit the job.

5. The GPSS instance servicing the job uses its `Shadow:Key`, or the default key, to decode the shadowed password string specified in `PASSWORD:`, and connects with Greenplum Database.

Pulling Information from the Debug Server

When you specify a `DebugPort` in the `gpss.json` configuration file, or when you specify the `--debug-port` option to the `gpss` command, GPSS starts a debug server on the local host. This server makes available additional debug information about the running GPSS instance, including the call stack and performance statistics.

You can use the `curl` command to view the types of information available (HTML output):

```
$ curl http://127.0.0.1:9998/debug/pprof/ > debug_info.html
```

Or, use `curl` to view specific information:

```
$ curl http://127.0.0.1:9998/debug/pprof/heap?debug=1 > debug_heap
$ curl http://127.0.0.1:9998/debug/pprof/goroutine?debug=1 > debug_goroutine
$ curl http://127.0.0.1:9998/debug/pprof/block?debug=1 > debug_block
```

The commands above gather information and write the heap, the call stack, and locking information each to a text file in the current working directory.

The GPSS debug server can also provide CPU profiling data. The following command gathers 10 seconds of CPU profile data and writes it in binary format to the output file named `debug_cpu_profile`:

```
$ curl http://127.0.0.1:9999/debug/pprof/profile?seconds=10 > debug_cpu_profile
```

You may be asked to send the binary output file to support. Alternatively, you can run the Go Profiling Tool on the file to parse and graph the results:

```
$ go tool pprof debug_cpu_profile
(pprof) web
```

The `web` command creates a graph of the profile data in SVG format, and opens the graph in a web browser.

Configuring the Streaming Server for Encryption and Authentication

GPSS supports authenticating with Kerberos to obtain both Kafka and Greenplum Database credentials. GPSS supports authenticating with LDAP to obtain Kafka credentials. GPSS also supports using TLS/SSL to encrypt communication between Kafka and GPSS, between RabbitMQ and GPSS, between the GPSS client and server, and on the data and control channels between GPSS/gpkafka and Greenplum.

GPSS does not support Kerberos authentication to RabbitMQ.

Configuring gpss and gpkafka for TLS-Encrypted Communications with Kafka

If your Kafka version 0.9 and newer cluster is configured to use TLS encryption, you must configure GPSS to use this encryption method when communicating with Kafka. You perform this configuration at both the GPSS service instance and client levels.

Refer to the Apache Kafka Encryption and Authetication using SSL documentation for more information about SSL configuration.

1. Create Kafka client keys for the `gpss` or `gpkafka` instance.
2. Specify the location of the GPSS client certificates via Kafka properties in the `PROPERTIES` or `rdkafka_prop` (version 3 (Beta)) block of the load configuration file. For example:

   ```
   PROPERTIES:
       security.protocol: SSL
       ssl.ca.location: /path/to/cert/kafka-ca.crt
   ```

```
    ssl.certificate.location: /path/to/cert/gpssclient.crt
    ssl.key.location: /path/to/cert/gpssclient.key
```

3. If you are using the `gpsscli` subcommands to load data, ensure that the `ListenAddress:Host` that you specify for the GPSS server identifies the common name (CN) in the certificate.

Configuring gpss for TLS-Encrypted Communications with RabbitMQ

GPSS supports TLS only when loading from a RabbitMQ queue. GPSS does not support TLS when loading from a RabbitMQ stream.

If your RabbitMQ cluster is configured to use TLS encryption, you may configure GPSS to use this encryption method when it communicates with RabbitMQ. You perform this configuration at both the GPSS service instance and client levels.

Refer to the RabbitMQ TLS Support documentation for more information about TLS configuration.

1. Create RabbitMQ client keys for the `gpss` instance. Refer to the *TLS Certificate Generator* github repository for information about generating RabbitMQ certificates.
2. Specify the RabbitMQ server in the load configuration file `SERVER` or `server` (version 3 (Beta)) property; you must omit the user name and password. For example (version 2):

   ```
   SERVER: localhost:5672
   ```

3. Specify the location of the GPSS client certificates via RabbitMQ properties in the `PROPERTIES` or `properties` (version 3 (Beta)) block of the load configuration file. For example:

   ```
   PROPERTIES:
       use.ssl: true
       ssl_options.servername: "hostname"
       ssl_options.cacertfile: /path/to/cert/rabbit/ca/cacert.pem
       ssl_options.certfile: /path/to/cert/rabbit/client/rabbit-client.cert.pem
       ssl_options.keyfile: /path/to/cert/rabbit/client/rabbit-client.key.pem
   ```

4. Optional RabbitMQ TLS configuration properties and example settings include the following:

   ```
   ssl_options.verify:  verify_peer
   ssl_options.fail_if_no_peer_cert: true
   ```

5. If you are using the `gpsscli` subcommands to load data, ensure that the `ListenAddress:Host` that you specify for the GPSS server identifies the common name (CN) in the certificate.

Configuring gpss and gpkafka for SSL-Encrypted Communications with Greenplum

There are two communication channels between GPSS and Greenplum Database: a control channel and a data channel. GPSS supports SSL encryption on both channels.

Configuring SSL for the Data Channel

GPSS supports SSL encryption on the data channel to Greenplum using the `gpfdists` protocol for encrypted communications.

If your Greenplum Database cluster is configured to use SSL, you must configure GPSS to use this encryption method for the data channel when it communicates with Greenplum.

1. Create GPSS keys for the `gpfdist` protocol instance.
2. Configure the `gpfdist` protocol to use SSL encryption to Greenplum by providing a `Gpfdist:Certificate` block in the GPSS configuration file, and identify the file system location of the SSL certificates. You can also specify a minimum TLS version. Sample `gpss.json` or `gpfdistconfig.json` excerpt:

   ```
   "Gpfdist": {
       "Host": "127.0.0.1",
       "Port": 5001,
       "Certificate": {
           "CertFile": "/home/gpadmin/cert/gpss.crt",
           "KeyFile": "/home/gpadmin/cert/gpss.key",
           "CAFile": "/home/gpadmin/cert/root_client.crt"
           "MinTLSVersion": "1.2"
       }
   }
   ```

3. If you are using `gpkafka` to load data, ensure that the `Gpfdist:Host` that you specify identifies the common name (CN) in the certificate.

Configuring SSL for the Control Channel

GPSS supports SSL encryption on the control channel to Greenplum as described in the PostgreSQL SSL Support documentation. GPSS uses SSL only for encryption on the control channel; it does not check the certificates. The default SSL mode that GPSS uses on this channel is the `prefer` mode.

You have two options for configuring GPSS client SSL for the control channel:

1. Option 1: Instruct GPSS to re-use the data channel (Gpfdist) certificate. Configuration steps include:
   1. Locating the `Gpfdist:Certificate` block in the `gpss.json` GPSS server configuration file.
   2. Setting the `DBClientShared` property to `true`:

      ```
      "Gpfdist": {
          ...
          "Certificate": {
              ...
              "DBClientShared": true
          }
      }
      ```

2. Option 2: Locate the certificates and keys in the `~/.postgresql` directory on the GPSS server host. Configuration steps include:

1. Copying the root certificate to the `~/.postgresql` directory on the GPSS server host.
2. Generating a private key and client certificate for the GPSS server and copying both to the `~/.postgresql` directory on the GPSS server host.

Configuring gpss and gpsscli for Encrypted gRPC Communications

GPSS supports encrypting communications between the `gpsscli` client and `gpss` server.

To use encrypted gRPC on connections between `gpsscli` and `gpss`, you must create server and client keys, and provide the keys via configuration files that you provide to the commands.

1. Create server keys for the `gpss` server instance.
2. Create client keys for the `gpsscli` client.
3. Configure the `gpss` service instance to use SSL encryption to the client by providing a `ListenAddress:Certificate` block in the `gpss.json` GPSS configuration file. The properties in this block should identify the file system location of the SSL server keys. You can also specify the minimum TLS version. Sample `gpss.json` excerpt:

   ```
   "ListenAddress": {
       "Host": "",
       "Port": 5019,
       "Certificate": {
           "CertFile": "/home/gpadmin/cert/gpss.crt",
           "KeyFile": "/home/gpadmin/cert/gpss.key",
           "CAFile": "/home/gpadmin/cert/root_cli.crt"
           "MinTLSVersion": "1.2"
       }
   }
   ```

4. Configure the GPSS client to use SSL encryption to the server by specifying the client keys in the `ListenAddress:Certificate` block of a GPSS configuration file that you provide to the `gpsscli` subcommand via the `--config gpsscliconfig.json` option. Sample `gpsscliconfig.json` excerpt:

   ```
   "ListenAddress": {
       "Host": "",
       "Port": 5019,
       "Certificate": {
           "CertFile": "/home/gpadmin/cert/gpsscli.crt",
           "KeyFile": "/home/gpadmin/cert/gpsscli.key",
           "CAFile": "/home/gpadmin/cert/root.crt"
       }
   }
   ```

If you encrypt communications between the GPSS client and server, but you want to deactivate certificate verification, specify the `--no-check-ca` option when you run the `gpsscli` subcommand.

Configuring gpss and gpkafka for Kerberos Authentication to Greenplum

If Kerberos authentication is enabled for Greenplum Database, you must configure `gpss` to authenticate with Kerberos.

GPSS uses a kerberos ticket, and the `USER` name specified in the load configuration file, to connect to Greenplum Database.

1. Create a Kerberos principal for each Greenplum Database user that will use GPSS to load data into Greenplum.
2. Specify the principal name in the load configuration file `USER` property value.
3. Generate a Kerberos ticket for this principal before you submit a load job with the `gpsscli submit`, `gpsscli load`, or `gpkafka load` commands.

If your Greenplum Database Kerberos service name is not the default (`postgres`), set the `PGKRBSRVNAME` environment variable to the correct service name before you start the `gpss` service instance or run `gpkafka load`.

Configuring gpss for Kerberos Authentication to Kafka

If your Kafka version 0.9 and newer cluster is configured for Kerberos authentication, you must configure GPSS to use this authentication method. You perform this configuration at both the `gpss` service instance level and the GPSS client level.

GPSS is a Kafka client. You must create a Kerberos principal for the `gpss` server instance accessing Kafka, and generate a keytab file for this principal. By default, GPSS runs `kinit` using this principal and keytab to generate the Kerberos ticket.

You must set certain Kafka properties in your load configuration file to use Kerberos user authentication to Kafka. The following table identifies keywords and values that you can add to the `PROPERTIES` or `rdkafka_prop` (version 3 (Beta)) block in your load configuration file:

Keyword	Value
security.protocol	The Kafka security protocol. Obtain the value from the Kafka server `server.properties` configuration file. GPSS supports the `SASL_SSL` (Kerberos and SSL) and `SASL_PLAINTEXT` (Kerberos, no SSL) protocols.
sasl.kerberos.keytab	The absolute path to the GPSS or user Kerberos keytab file for Kafka on the local system.
sasl.kerberos.kinit.cmd	The Kerberos `kinit` command string. If this property is not specified, GPSS uses the default value as described in `librdkafka` Global configuration properties when it runs the `kinit` command. If you do not want GPSS to run `kinit`, set the `sasl.kerberos.kinit.cmd` property to an empty value (`""`) or no value.
sasl.kerberos.principal	The GPSS or user Kerberos service principal name; typically of the format `<name>@<realm>` or `<primary>/<instance>@<realm>`.
sasl.kerberos.service.name	The Kafka Kerberos principal name. Obtain the value from the Kafka server `server.properties` configuration file. The default Kafka Kerberos service name is `kafka`.

For example:

```
PROPERTIES:
    security.protocol: SASL_PLAINTEXT
    sasl.kerberos.service.name: kafka
    sasl.kerberos.keytab: /var/kerberos/krb5kdc/gpss.keytab
    sasl.kerberos.principal: gpss/localhost@REALM.COM
    sasl.kerberos.kinit.cmd:
```

If you are accessing Kafka using both Kerberos authentication and SSL encryption, you must also specify the Kafka SSL properties identified in Configuring gpss and gpkafka for SSL-Encrypted Communications with Kafka.

Configuring gpss for LDAP Authentication to Kafka

If your Kafka cluster is configured for LDAP authentication, you must configure GPSS to use this authentication method. Set Kafka properties in your load configuration file as follows:

1. Edit the `BROKERS` to specify a broken that supports LDAP authentication.
2. Add keywords and values to the `PROPERTIES` or `rdkafka_prop` (version 3 (Beta)) block in your load configuration file as needed:

Keyword	Value
security.protocol	Specify `SASL_PLAINTEXT` as the Kafka security protocol when authenticating to LDAP.
sasl.mechanism	Specify `PLAIN` for the security mechanism.
sasl.username	Enter the LDAP user name.
sasl.password	Enter the LDAP user password.

GPSS also supports using a shadow string for the LDAP user password with the gpsscli shadow command. If you generate a shadowed password, specify the password using the format:

```
sasl.password: "SHADOW:<shadow_password_string>"
```

Configuring the Streaming Server for Client-to-Server Authentication

You can configure a user name and password that GPSS will use for client-to-server authentication. When you configure GPSS client authentication, you identify the credentials in the `gpss.json` server configuration file. Users that access the GPSS server instance must then provide (a variation of) the credentials to any `gpsscli` subcommand that they invoke as described below.

You can configure client authentication as follows:

1. Identify a user name for authenticating.
2. Specify the user name in the `Authentication:Username` property setting of the `gpss.json` GPSS server configuration file. For example:

```
"Authentication": {
    "Username": "client_auth_username",
```

```
    "Password": "<shadow_passwd_string>"
}
```

3. Identify a password for authenticating.
4. Optionally configure a `Shadow:Key`, and then generate a shadowed password string from the password as described in steps 1 and 2 of Shadowing the Greenplum Database Password.
5. Specify the shadow password string returned by `gpsscli shadow` in the `Authentication:Password` property setting of the `gpss.json` GPSS server configuration file. For example:

   ```
   "Authentication": {
       "Username": "client_auth_username",
       "Password": "SHADOW:ERTBKXDWLAJHUF5UOGJY34QTXIBNYP4ULTWVHIUZIF4UYFPRIJVA"
   }
   ```

   Always quote the complete shadow password string.
6. (Re)Start the GPSS server.
7. Notify users of the GPSS server instance of the new client authentication requirements, and provide them the user name and original (not shadowed) password.

All `gpsscli` subcommands directed to a GPSS server instance that is configured for client authentication must specify the authentication credentials via the `-U client_auth_username` and `-P original_passwd` command line options. For example:

```
$ gpsscli submit -U client_auth_username -P changeme loadcfg.yaml
```

Enabling Prometheus Metrics Collection

With the Greenplum Streaming Server's out-of-the-box Prometheus integration, you can obtain runtime metrics for a `gpss` server instance when you enable Prometheus monitoring for the UNIX process.

The GPSS metrics available from Prometheus are:

Name	Description
gpss_jobs_total	The total number of jobs the `gpss` server instance is servicing.
gpss_jobs_running	The number of jobs that are currently in the *Running* state.
gpss_process_cpu_seconds_total	The total user and system CPU time (in seconds) for the GPSS server process.
gpss_process_open_fds	The number of open file descriptors in the GPSS server instance process.
gpss_process_max_fds	The maximum number of open file descriptors allowed for the GPSS server instance process.
gpss_process_virtual_memory_bytes	The virtual memory size (in bytes) of the GPSS server instance process.
gpss_process_virtual_memory_max_bytes	The maximum virtual memory size (in bytes) of the GPSS server instance process.
gpss_process_resident_memory_bytes	The resident memory size (in bytes) of the GPSS server instance process.

Name	Description
gpss_process_start_time_seconds	The start time of the GPSS server instance process since epoch (in seconds).

Prerequisites

GPSS uses the Prometheus stable HTTP API `/api/v1`. Before you enable Prometheus integration with GPSS, ensure that:

- You are running a Prometheus server compatible with the v1 API. If you are new to Prometheus, you can download and install a Prometheus server as described in the Getting Started topic in the Prometheus documentation.
- Connectivity exists between the host on which you run the GPSS server instance and the host on which the Prometheus server is running.

Enabling Prometheus Integration with GPSS

Prometheus collects metrics from monitored processes by scraping HTTP endpoints exposed by the processes. Scraping pulls published statistics that allow you to aggregate and record time series data, or to generate alerts.

To enable Prometheus integration with GPSS, you must:

1. Identify the host name or IP Address and port number on which Prometheus will pull statistics from the GPSS server instance. Both GPSS and Prometheus require this configuration information.
2. Add a `Monitor` property block to the `gpss.json` GPSS server configuration file that identifies the address and port number from which GPSS will allow Prometheus to pull the server's statistics. For example:

   ```
   Monitor: {
       Prometheus: {
           Listening: "0.0.0.0:5001"
       }
   }
   ```

   The `Listening` property value specified in this block allows any Prometheus server host to pull from port 5001.
3. Create a new, or update an existing, YAML-format Prometheus configuration file to include a GPSS scrape target. For example, to configure Prometheus metrics collection for the GPSS server instance running on the host named `etlhost` that was configured with the `Monitor` block above, copy/paste the following text into a file named `prometheus_gpss_cfg.yml`:

   ```
   global:
     scrape_interval:     15s # By default, scrape targets every 15 seconds.

   # A scrape configuration containing exactly one endpoint to scrape, GPPSS
   scrape_configs:
     # The job name is added as a label `job=<job_name>` to any timeseries
     # scraped from this config.
     - job_name: 'gpss'
   ```

```
    static_configs:
      - targets: ['etlhost:5001']
```

This configuration instructs Prometheus to pull metrics from port `5001` on `etlhost` (the GPSS server instance) every fifteen seconds.

4. Start the Prometheus server, specifying the configuration file that you updated or created for GPSS:

```
$ prometheus --config.file=prometheus_gpss_cfg.yml &
```

5. Start the GPSS server:

```
$ gpss --config gpss.json &
```

Viewing GPSS Metrics

When you run a Prometheus server, metrics for integrated services are available at the default port `9090`. To view metrics for all services integrated with Prometheus, navigate to `http://<prometheus_host>:9090` in your web browser of choice.

To view the metrics for a specific GPSS server instance, navigate to **Status->Targets**, and select the GPSS *Endpoint* of interest.

The default Prometheus web user interface is rather low level. Once enabled, you can use the Prometheus expression browser to generate and run ad-hoc queries on GPSS time series data.

To view graphs of GPSS metrics, enable Grafana integration as described in Grafana Support for Prometheus in the Prometheus documentation.

About Loading Data with the Streaming Server

You will perform the following tasks when you use the Greenplum Streaming Server (GPSS) to load data into a Greenplum Database table:

1. Ensure that you meet the prerequisites, and that you have configured and started the Greenplum Streaming Server.
2. Identify the source and the format of the data and construct the load configuration file (optional).
3. Create the target Greenplum Database table.
4. Assign Greenplum Database role permissions to the table, if required, as described in Configuring Greenplum Database Role Privileges.
5. Run the GPSS client.
6. Verify the load operation as described in Checking for Load Errors.

Constructing the Load Configuration File

The Greenplum Streaming Server requires a load configuration file when you use the `gpsscli` or `gpkafka` client utilities to load data into Greenplum Database. A load configuration file is not required if you are using the VMware Greenplum Connector for Apache NiFi or a custom GPSS client application.

You configure a load operation from a data source to Greenplum Database via a YAML-formatted configuration file as described in gpsscli.yaml. This configuration file includes properties that identify the data source and format, information about the Greenplum Database connection and target table, and error and commit thresholds for the operation.

GPSS supports some pre-defined data formats, including Avro, binary, CSV, and JSON. GPSS also supports custom data formats. Refer to Understanding Custom Formatters for information on developing and using a custom formatter with GPSS.

GPSS supports version 1 (deprecated), version 2, and version 3 (Beta) load configuration file formats. Versions 1 and 2 use a similar YAML structure. Version 3 introduces a new YAML structure, organization, and keywords.

Refer to Constructing the gpkafka.yaml Configuration File for the YAML file format for a Kafka data source, Constructing the filesource.yaml Configuration File for the YAML file format for a file data source, or Constructing the rabbitmq.yaml Configuration File for a RabbitMQ data source.

Creating the Target Greenplum Table

You are required to pre-create the target Greenplum table before you initiate a data load operation to Greenplum Database from a GPSS client. You must be able to identify both the schema name and table name of the target table.

The column data types that you specify for the target Greenplum Database table are informed by the data formats supported by the GPSS client.

Configuring Greenplum Database Role Privileges

If you load data to Greenplum Database from a GPSS client using a non-admin Greenplum user/role, the Greenplum administrator must assign the role certain privileges:

- The role must have `USAGE` and `CREATE` privileges on any non-public database schema where:
    - The role writes data to a table in the schema, or
    - `gpss` creates external tables.

    For example:

```
=# GRANT USAGE, CREATE ON SCHEMA <schema_name> TO <role_name>;
```

- If the role writing to Greenplum Database is not a database or table owner, the role must have `SELECT` and `INSERT` privileges on each Greenplum Database table to which the role will write data:

```
=# GRANT SELECT, INSERT ON <schema_name>.<table_name> TO <role_name>;
```

- The role must have permission to create readable external tables using the Greenplum Database `gpfdist` protocol:

```
=# ALTER ROLE <role_name> CREATEEXTTABLE(type = 'readable', protocol = 'gpfdis
t');
```

Refer to the VMware Tanzu Greenplum Managing Roles and Privileges documentation for further information on assigning privileges to Tanzu Greenplum users.

Do not directly `SELECT` from an external table that GPSS creates for your job. Any data that you read in this manner will not be loaded into the Greenplum Database table.

Running the Client

You run a GPSS client to use the Tanzu Greenplum streaming server to load data into Tanzu Greenplum. Installation, configuration, and run procedures for a GPSS client are client-specific. For example, refer to the VMware Tanzu Greenplum Connector for Apache NiFi Documentation for the installation, configuration, and run procedures for the Connector for Apache NiFi.

You can also use the gpsscli client command line utility to load data into Greenplum Database.

Using the gpsscli Client Utility

The Greenplum Streaming Server (GPSS) includes the `gpsscli` client command utility. `gpsscli` provides subcommands to manage Greenplum Streaming Server load jobs and to view job status and progress:

Subcommand	Description
convert	Convert a version 1 or 2 load configuration file to version 3 (Beta) format
dryrun	Perform a trial load without writing to Greenplum Database
help	Display command help
list	List jobs and their status
load	Run one or more single-command load
progress	Show job progress
remove	Remove one or more jobs
start	Start one or more jobs
status	Show job status
stop	Stop one or more jobs
submit	Submit one or more jobs
wait	Wait for a job to stop

All subcommands include options that allow you to specify the host and/or port number of the GPSS instance that you want to service the request (`--config` or `--gpss-host` and `--gpss-port`). You can also specify the directory to which GPSS writes `gpsscli` client log files (`--log-dir`).

The Greenplum Streaming Server includes a client command utility named `gpkafka`. `gpkafka` is a wrapper around the `gpss` and `gpsscli` utilities that provides data load capabilities to Greenplum from a Kafka data source. Refer to Loading Kafka Data into Greenplum for information about loading data from Kafka into Greenplum Database.

A typical command workflow when using `gpsscli` to load data into Greenplum Database follows:

1. Submit a Greenplum Streaming Server job.
2. Start the Greenplum Streaming Server job.
3. (Optional) Check the status or progress of the Greenplum Streaming Server job.
4. (Optional) Wait for a Greenplum Streaming Server job to complete.
5. Stop the Greenplum Streaming Server job.
6. Remove the Greenplum Streaming Server job.

Alternatively, you can run a single-command load operation that submits a GPSS job on your behalf, starts the job, displays job progress, and may stop the GPSS job. See Running a Single-Command Load.

About the gpsscli Return Codes

All `gpsscli` subcommands return zero (`0`) on success and non-zero on failure. The specific return code values and failure descriptions are identified in the table below.

Return Code	Description
0	Success
1	Internal error
2	RPC error
3	Job error; the status of one or more jobs that the subcommand operated on is *Error*

About GPSS Job Identification

You identify a job by a name that you specify or a unique identifier that GPSS assigns. Job names must be unique. Use the name to manage the job throughout its lifecycle.

GPSS uses a data source-specific combination of properties specified in the load configuration file to internally identify a job. For example, when it loads from a Kafka data source, GPSS uses the Kafka topic name, and the target Greenplum database, schema, and table names for internal job identification. GPSS creates internal and external tables for each job that are keyed off of these properties; these tables keep track of the progress of the load operation. GPSS considers any load configuration file submitted with the same value for these job-identifying properties to be the same internal job.

About External Table Naming and Lifecycle

GPSS creates a unique external table to load data for a specific job directly into Greenplum Database segments. Kafka job external table names begin with the prefix `gpkafkaloadext_`, file job external table names begin with `gpfileloadext_`. RabbitMQ job external table names begin with the prefix `gprabbitmqloadext_`. And by default, GPSS reuses this external table each time you restart the job.

The complete name and the lifecycle of a GPSS external table depends on the `ReuseTables` setting in the `gpss.json` server configuration file as described below.

ReuseTables=true

When `ReuseTables` is `true` (the default setting), GPSS reuses and does not drop an external table for a job.

GPSS creates the external table using the following naming format: `gp<datasource>loadext_<hash>`.

GPSS calculates the <hash> based on values of server configuration properties and load/job configuration properties that would change the external table definition. These properties include:

- Gpfdist host and port number
- Target Greenplum Database schema name
- Target Greenplum Database table name
- Target Greenplum Database table definition
- Gpfdist use of encrypted communications

- Source data format
- Source data types
- Formatter options
- Job key; GPSS generates the job key as follows for the different data sources:
    - For file and s3 jobs, GPSS generates the job key from the database name, metadata schema name, target table name, and source URL.
    - For Kafka jobs, GPSS generates the job key from the output schema and table names, or the target schema names and table names plus source topic when the load job targets multiple outputs.
    - For RabbitMQ jobs, GPSS generates the job key from the database name, output schema and table names, the RabbitMQ stream or queue name, and the RabbitMQ virtual host.
- Error Limit

ReuseTables=false

When `ReuseTables` is `false`, GPSS drops an external table, if one exists, when a job is (re)started.

Repeated drop/create of external tables may cause bloating in the `pg_attribute` and `pg_class` system catalog tables; be sure to `VACUUM` these tables frequently.

GPSS creates the external table using the job name instead of a hash. The default job name is the base name of the YAML load configuration file. You set a custom job name when you specify the `--name <jobname>` option to the `gpsscli submit` command.

The external table naming format when `ReuseTables` is `false` follows: `gp<datasource>loadext_<jobname>`.

Submitting a Job

To register a data load operation to Greenplum Database, you submit a job to the Greenplum Streaming Server using the gpsscli submit subcommand. When you submit a job, you provide a YAML-formatted configuration file that defines the properties of the load operation. Load properties include Greenplum-specific options, as well as properties that are specific to the data source. See gpsscli.yaml.

You identify a GPSS job by a name that you provide via the `--name` option. If you do not specify a job name, GPSS assigns and returns the base name of the load configuration file as the job identifier. You use this name or identifier to identify the job throughout its lifecycle.

The following example submits a GPSS job named `order_upload` whose load properties are defined in the configuration file named `loadcfg.yaml`:

```
$ gpsscli submit --name order_upload loadcfg.yaml
```

A newly-submitted GPSS job is in the *Submitted* state.

Starting a Job

To start a GPSS job, you run the gpsscli start subcommand. When you start a job, GPSS initiates the data load operation from the client. It sets up the connection to Greenplum Database and creates the external tables that it uses to load data directly into Greenplum segments.

The following example starts the GPSS job named `order_upload`:

```
$ gpsscli start order_upload
```

A job that starts successfully enters the *Running* state.

The default behaviour of `gpsscli start` is to return immediately. When you specify the `--quit-at-eof` option, the command reads data until it receives an EOF, and then stops the job. In this scenario, the job transitions to the *Success* or *Error* state when the command exits.

Checking Job Status, Progress, History

GPSS provides several commands to check the status of a running job(s):

- The gpsscli list subcommand lists running (or all) jobs and their status:

  ```
  $ gpsscli list --all
  JobName         JobID                                GPHost      GPPort     DataBas
  e     Schema      Table             Topic            Status
  order_upload    d577cf37890b5b6bf4e713a9586e86c9     sys1        5432       testdb
  public      order_sync        orders           JOB_RUNNING
  file_jobf3      cfb985bcdc8884352b4b8853f5d06bbe     sys1        5432       testdb
  public      from_csvfile      file:///tmp/data.csvJOB_STOPPED
  ```

- The gpsscli status subcommand displays the status of a specific job:

  ```
  $ gpsscli status order_upload
  ...,101565,info,Job order_upload, status JOB_RUNNING, errmsg [], time 2020-08-0
  4T16:44:03.376216533Z
  ```

  Use the `gpsscli status` command to determine the status or success or failure of the operation. If the job status is *Error*, you will want to examine command output and log file messages for additional information. See Checking for Load Errors.

- The gpsscli progress subcommand displays the progress of a running Kafka job. The command waits, and displays the job commit history and transfer speed at runtime. `gpsscli progress` returns when the job stops.

  GPSS currently supports job progress tracking only for Kafka data sources.

  ```
  $ gpsscli progress order_upload
  StartTime                       EndTime                         MsgNum    MsgSize
  InsertedRecords RejectedRecords Speed
  2019-10-15T21:56:49.950134Z    2019-10-15T21:56:49.964751Z     1000      78134
  1000            0                 735.13KB/sec
  2019-10-15T21:56:49.976231Z    2019-10-15T21:56:49.984311Z     1000      77392
  1000            0                 701.58KB/sec
  ```

```
2019-10-15T21:56:49.993607Z   2019-10-15T21:56:50.003602Z    1000     77194
1000            0               723.32KB/sec
```

By default, `gpsscli progress` displays job progress by batch. To display job progress by partition, specify the `--partition` option to the subcommand:

```
$ gpsscli progress order_upload --partition
PartitionID   StartTime                     EndTime                       Begi
nOffset   EndOffset    MsgSize    Speed
0             2019-10-15T21:56:54.80469Z    2019-10-15T21:56:54.830441Z    2420
00        243000       81033      652.29KB/sec
0             2019-10-15T21:56:54.846354Z   2019-10-15T21:56:54.880517Z    2430
00        244000       81021      675.12KB/sec
0             2019-10-15T21:56:54.893097Z   2019-10-15T21:56:54.904745Z    2440
00        245000       80504      673.67KB/sec
```

GPSS also keeps track of the progress of each Kafka load job in a separate CSV-format log file. The job progress log files, named `progress_<jobname>_<jobid>_<date>.log`, reside in the GPSS server log directory. Refer to the Kafka data source Checking the Progress of a Load Operation topic for more information.

Waiting for a Job to Complete

You can use the gpsscli wait subcommand to wait for a running job to complete. A job is complete when there is no more data to read, or when an error is returned. Such jobs transition from the *Running* state to the *Success* or *Error* state.

```
$ gpsscli wait  order_upload
```

`gpsscli wait` exits when the job completes.

Stopping a Job

Use the gpsscli stop subcommand to stop a specific job. When you stop a job, GPSS writes any unwritten batched data to the Greenplum Database table and stops actively reading new data from the data source.

```
$ gpsscli stop  order_upload
```

A job that you stop enters the *Stopped* state.

Removing a Job

The gpsscli remove subcommand removes a GPSS job. When you remove a job, GPSS unregisters the job from its job list and releases all job-related resources.

```
$ gpsscli remove order_upload
```

Running a Single-Command Load

The gpsscli load subcommand initiates a data load operation. When you run `gpsscli load`, GPSS submits, starts, and displays the progress (Kafka job only) of a job on your behalf.

By default, `gpsscli load` loads all available data and then waits indefinitely for new messages to load. In the case of user interrupt or exit, the GPSS job remains in the *Running* state. You must explicitly stop the job with `gpsscli stop` when running in this mode.

When you provide the `--quit-at-eof` option to the command, the utility exits after it reads all published data, writes the data to Greenplum Database, and stops the job. The GPSS job is in the *Success* or *Error* state when the command returns.

Similar to the `gpsscli submit` command, `gpsscli load` takes as input an optional name and a YAML-format configuration file that defines the load properties:

```
$ gpsscli load --quit-at-eof loadcfg.yaml
```

Because the above command does not specify the `--name` option, GPSS assigns and returns the job identifier `loadcfg` when you run it.

About GPSS Job Initiation and Scheduling

In the default configuration, GPSS relies on the `gpsscli` subcommands that you submit to initiate and stop jobs. Once a job is started, GPSS does not automatically (re)start the job, and GPSS stops a job only when you have specified the `--quit-at-eof` option to the `gpsscli` subcommand.

You can configure GPSS to automatically stop and restart failed and running jobs via scheduling properties that you specify in the load configuration file.

About Registering for Job Stopped Notification

You can register to be notified when a job is stopped for any reason (success, error, completed, user-initiated stop) via alert properties that you specify in the load configuration file. When a job stops, GPSS will invoke a command that you specify, after an optional period of time that you configure elapses.

About Retrying a Failed Job

You can configure GPSS to restart a *failed* job after a period of time that you specify; you can also configure the maximum number of times that GPSS retries the job. The load configuration file properties that govern failed job retry are located in the `SCHEDULE:` block.

Refer to Auto-Restarting a Failed Job for additional information about retrying a failed job.

About Job Scheduling

You can configure GPSS to automatically stop a running job after it has run for a period of time, or at a specific clock time after receiving an EOF. You can also configure a restart interval and the maximum number of times GPSS should restart a job that it stopped. The load configuration file properties that govern job scheduling in GPSS are also located in the `SCHEDULE:` block.

Checking for Load Errors

The Greenplum Streaming Server cannot directly return success or an error to the client. You can obtain success and error information for a GPSS load operation from `gpsscli` subcommand output, and from

messages that GPSS writes to `stdout` or writes to the server, progress (Kafka jobs only), and/or client log files.

You can also view data formatting-specific errors encountered during a load operation in the error log.

Error checking activity may include:

- Examining GPSS Log Files
- Determining Batch Load Status
- Diagnosing an Error with a Trial Load
- Reading the Error Log
- Auto-Restarting a Failed Job
- Redirecting Data to a Backup Table when GPSS Encounters Expression Evaluation Errors
- Preventing External Table Reuse

Examining GPSS Log Files

GPSS writes server and client log messages to files as described in Managing GPSS Log Files.

GPSS Version	Log File Content
1.4.0 and older	`<date>:<time> <proc>:<user>:<host>:<proc_pid>-[<severity>]:-<message>`
1.4.1 - 1.9.x	`timestamp,pid,level,message` (header row, CSV format)
1.10.0 and newer	`timestamp,pid,level,message` (client log file header row, CSV format) `timestamp,job_id,pid,level,message` (server log file header row, CSV format)

Example message in a `gpss` log file:

```
20230427 15:17:22.95110,-,31424,info,gpss listening on :5000
```

GPSS writes at most the first 8 characters of a job identifier, or writes `-` when the message is not job-specific.

Example message in a `gpsscli` log file:

```
20230427 16:28:46.39607,1305,info,"JobID: 593347a306a1f9439a127b982b2f891f,JobName: ni
ghtly_load"
```

Determining Batch Load Status

To determine if GPSS loaded one or more batches of data to Greenplum Database successfully, first examine the status and progress of the job in question. The `gpsscli status` and `gpsscli progress` command output will identify if any known error conditions exist.

Also examine `gpss` command output and logs, searching for messages that identify the number of rows inserted and rejected. For a Kafka or RabbitMQ data source, search for:

```
... -[INFO]:- ... Inserted 9 rows
... -[INFO]:- ... Rejected 0 rows
```

Or, for load jobs originating from a file data source:

```
... -[INFO]:- ... inserted 5, rejected 1
```

Diagnosing an Error with a Trial Load

When a Kafka, file, or S3 job fails, you may choose to perform a trial run of the load operation to help diagnose the cause of the failure. The gpsscli dryrun command reads the data from the source and prepares to load the data, but does not actually insert it into Greenplum Database. The command returns the results of this processing, as well as the SQL commands that GPSS would run to complete the job.

Sample command that specifies a Kafka load configuration file:

```
$ gpsscli dryrun --include-error-process kjobcfgv3.yaml
```

Sample command output:

```
jobid: 01ba08c0f7fc8e3a49e2ad1ee48ef899
jobname: kjobcfgv3
jobtype: KafkaJob
tracking table name: gpkafka_tbl_1_column_text_01ba08c0f7fc8e3a49e2ad1ee48ef899
progress log file name: progress_kjobcfgv3_01ba08c0_20220218.log

Extension version checking:
    <SQL commands to check extension versions>

SQL of job:
    <SQL commands to fullfil the load operation>
```

Because the `--include-error-process` flag was specified for the Kafka job dry run, the output may include the following text:

```
Check format error:
    error sql query: <query>
    clean up error table: <query>

Check failed batch with expression error:
    get failing batch query: <query>
```

Reading the Error Log

If `gpss` command or log output indicates that rows were rejected, the output will identify an SQL query that you can run to view the data formatting errors that GPSS encountered while inserting data into a Greenplum Database table.

GPSS uses the `LOG ERRORS` feature of Greenplum Database external tables to detect and log data rows with formatting errors. The functions that you use to access and manage the error log, and the persistence of the error data, depend on the version of Greenplum Database that you are running GPSS against and the `ReuseTables` setting in effect when you started the `gpss` server.

If you are running GPSS against Greenplum Database versions 5.26+ or 6.6+ and you started the `gpss` server with `ReuseTables=false`:

- GPSS automatically specifies `LOG ERRORS PERSISTENTLY` when it creates external tables for a job.

- You use the `gp_read_persistent_error_log()` function to retrieve the error data.
- The error data persists in the error log, and stays around until you explicitly remove it.

If you are running GPSS against older 5.x and 6.x versions of Greenplum, or you started the `gpss` server with `ReuseTables=true`:

- You use the `gp_read_error_log()` function to retrieve the error data.
- The error data is accessible from the error log until GPSS drops the external table. (If the `gpss` server is started with `ReuseTables=true`, GPSS does not drop an external table for a job. If `ReuseTables=false`, GPSS drops an external table, if one exists, when a job is (re)started.)

Refer to the Tanzu Greenplum CREATE EXTERNAL TABLE documentation for more information about the external table error logs and error log management.

When you run the query to view the error log, you specify the name of the external table that GPSS used for the load operation. You identify the name of the external table by examining the `gpss` command output and/or log file messages. For best results, use the (short) time interval identified in the `gpss` output.

Kafka job external table names begin with `gpkafkaloadext_`, file job external table names begin with `gpfileloadext_`, and RabbitMQ job external table names begin with `gprabbitmqloadext_`.

The following example query displays the number of errors returned in a Kafka load job:

```
SELECT count(*) FROM gp_read_error_log('"public"."gpkafkaloadext_ae0eac9f8c94a487f30f7
49175c3afbf"')
    WHERE cmdtime > '2018-08-10 18:44:23.814651+00';
```

Warning Do not directly `SELECT` from an external table that GPSS creates for your job. Any data that you read in this manner will not be loaded into the Greenplum Database table.

Auto-Restarting a Failed Job

A job may fail for temporary reasons. You can configure GPSS to automatically restart an errored job. GPSS automatic job restart is deactivated by default. When the YAML-format load configuration file submitted for a job includes the non-default `SCHEDULE:` block `MAX_RETRIES` and `RETRY_INTERVAL` configuration settings, GPSS will attempt to restart the job if the job enters the *Error* state after it starts.

GPSS stops trying to restart the job when the configured retry limit is reached, or if the job is removed or the job configuration is updated during retry.

Jobs that you start via the `gpsscli start`, `gpsscli load`, and `gpkafka load` commands are eligible for automatic job restart on error. If you provide the `--quit-at-eof` flag or one of the `--force-reset-*xxx*` flags when you run the command and the Kafka or RabbitMQ job load configuration file specifies failed job retry settings, GPSS ignores the flag on any retry attempts that it initiates.

Redirecting Data to a Backup Table when GPSS Encounters Expression Evaluation Errors

GPSS catches data formatting errors during loading, and you can view these errors with `gp_read_error_log()` as described in Reading the Error Log.

There may be cases where your data is formatted correctly, but GPSS encounters an error when it evaluates a mapping or a filter expression. If the evaluation fails, GPSS cannot log and propagate the error

back to the user.

For example, if you specify the following mapping expression in your load configuration file:

```
EXPRESSION: (jdata->>'id')::int
```

and the content of `jdata->>'id'` is a string that includes non-integer characters, the expression will fail when Greenplum Database evaluates it.

The load configuration property `COMMIT: SAVE_FAILING_BATCH` (versions 2 and 3 load configuration file formats only) governs whether or not GPSS saves a batch of data into a backup table before it writes the data to Greenplum Database. Saving the data in this manner aids loading recovery when GPSS encounters errors during the evaluation of expressions.

By default, `SAVE_FAILING_BATCH` is `false`, and GPSS immediately terminates a load job when it encounters an expression error.

When you set `SAVE_FAILING_BATCH` to `true`, GPSS writes all data in the batch to a backup table named `gpssbackup_<jobhash>`. GPSS writes both good and bad data to the backup table.

A backup table has the following columns:

Column Name	Description
data	The data associated with the row that GPSS attempted to load into Greenplum Database.
gpss_save_timestamp	The time that GPSS inserted the row into the backup table.
gpss_expression_error	The error that GPSS encountered when it ran the expression specified in the column mapping.

Sample backup table content for a failed load operation follows:

```
test=# SELECT * from gpssbackup_e0c5991570303703450bbac2ee8816bc;

-[ RECORD 1 ]---------+----------------------------------------------------------------------
-----------------------------------------
data                  | {"device": "agkNtzFnHIVASYNvo", "humidity": 91.3, "temperatur
e": 9, "time": "2019-09-24T15:33:57.054175"}
gpss_save_timestamp   | 2022-07-04 07:51:17.798469+00
gpss_expression_error | division by zero

-[ RECORD 2 ]---------+----------------------------------------------------------------------
-----------------------------------------
data                  | {"device": "LcZQGnVXhORIKxWY", "humidity": 46.289, "temperatur
e": 9, "time": "2019-09-24T15:33:57.054561"}
gpss_save_timestamp   | 2022-07-04 07:51:17.798469+00
gpss_expression_error | division by zero
```

GPSS continues to process Kafka messages even after it encounters an expression error. When GPSS encounters one or more expression errors in a batch, none of the good data in the batch is written to Greenplum. You can set the `RECOVER_FAILING_BATCH` (Beta) configuration property to instruct GPSS to automatically reload the good data in the batch, and retain only the error data in the backup table. GPSS displays additional information about the recovery process when you set this option.

Using a backup table in this manner to hedge against expression errors may impact performance, especially when the data that you are loading has not been cleaned.

Preventing External Table Reuse

GPSS creates a unique external table to load data for a specific job directly into Greenplum Database segments. By default, GPSS reuses this external table each time you restart the job. If the structure of either the source data or the destination Greenplum Database table is altered, GPSS may not be able to reuse the external table it initially created for the job.

You can configure GPSS to create a new external table for all new and restarted jobs submitted to a `gpss` service instance by setting the `ReuseTables` configuration property to `false` in the `gpss.json` file.

Understanding Custom Formatters

This topic describes custom formatters and how to use them with Greenplum Streaming Server.

A custom formatter is a C function that performs specific formatting or processing on data that is accessed by a Greenplum Database external table. A custom formatter may support options that you provide to direct the processing performed by the function. Greenplum includes built-in and custom formatters. You can also develop your own custom formatter.

You compile the C custom formatter functions that you develop into a shared library. These functions are available to Greenplum Database users after the shared library is installed in the Greenplum Database cluster and the custom formatter functions are registered as SQL UDFs.

This topic includes the following sections:

- Developing a Custom Formatter for GPSS
- Using a Custom Formatter in GPSS

Developing a Custom Formatter for GPSS

A custom formatter is a PostgreSQL C language function; refer to C-Language Functions in the PostgreSQL documentation for detailed information about developing C language functions.

Important header files for custom formatter development include: `postgres.h`, `access/formatter.h`, and `fmgr.h`. These headers define the functions and macros required to interact with PostgreSQL and formatter C structures.

For an example Greenplum Database custom formatter implementation, refer to the [formatter_fixedwidthhttps://github.gwd.broadcom.net/TNZ/gp-gpdb/tree/main/contrib/formatter_fixedwidth) example in the Greenplum Database open source `github` repository.

See Custom Formatter for Kafka for a GPSS- and Kafka-specific custom formatter example.

You can develop and test a custom formatter against a Greenplum Database external table that specifies the `file:` protocol in the `LOCATION` URI. Any custom formatter that you develop and test in this fashion should be compatible with GPSS.

The remainder of this topic describes special considerations when developing a custom formatter for use with GPSS.

About Data Boundaries

GPSS handles data boundaries from the source; a custom formatter can expect to receive a complete Kafka message. Certain formatter API calls are expected to behave differently than a typical Greenplum Database formatter:

- `FORMATTER_SET_DATACURSOR()` has no effect.
- `FORMATTER_GET_DATALEN()` always returns the full size of the message. The message is guaranteed to be complete.
- `FORMATTER_GET_DATACURSOR()` always returns `0`.
- GPSS throws an error when the custom formatter returns `FMT_NEED_MORE_DATA`.

Handling Bad Data

When the custom formatter encounters an unrecoverable error, it should invoke `ereport()` with the error code `ERRCODE_INTERNAL_ERROR` (or its siblings) to indicate that the process should be terminated. The bad row data will not be written to the error log in this case.

If the custom formatter encounters an ignorable error and the data loading should continue, it should invoke `ereport()` with the error code `ERRCODE_DATA_EXCEPTION` (or its siblings). In this scenario, GPSS writes the bad row data to the error log automatically.

The GPSS extension invokes the `FORMATTER_SET_BAD_ROW_DATA()` function; the function has no effect when invoked by the custom formatter.

Known Issues

Greenplum Database truncates bad row data written to the error log at the first 0x00 byte. As a result, the `gp_read_error_log()` and `gp_read_persistent_error_log()` functions may return an incomplete `rawbytes`.

Building the Custom Formatter Shared Library with PGXS

You compile the custom formatter function that you write into a shared library that the Greenplum Database server loads on demand.

You can use the PostgreSQL build extension infrastructure (PGXS) to build the source code for your custom formatter function against a Greenplum Database installation. This framework automates common build rules for simple modules. If you have a more complicated use case, you must write your own build system.

To use the PGXS infrastructure to generate a shared library for a custom formatter function that you develop, create a simple `Makefile` that sets PGXS-specific variables.

Refer to Extension Building Infrastructure in the PostgreSQL documentation for information about the `Makefile` variables supported by PGXS.

For example, the following `Makefile` generates a shared library named `customfmt_example.so` from a C source file named `customfmt.c`:

```
MODULE_big = customfmt_example
OBJS = customfmt.o
PG_CPPFLAGS = -I$(shell $(PG_CONFIG) --includedir)
SHLIB_LINK = -L$(shell $(PG_CONFIG) --libdir)

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

`MODULE_big` identifes the base name of the shared library generated by the `Makefile`.

`PG_CPPFLAGS` adds the Greenplum Database installation include directory to the compiler header file search path.

`SHLIB_LINK` adds the Greenplum Database installation library directory to the linker search path.

The `PG_CONFIG` and `PGXS` variable settings and the `include` statement are required and typically reside in the last three lines of the `Makefile`.

Registering the Custom Formatter Function with Greenplum Database

Before you can use a custom formatter, you must register the function with Greenplum Database.

Registering the function involves mapping the function signature to a SQL user-defined function. You define this mapping with the `CREATE FUNCTION .. AS` command specifying the shared library name. You may choose to use the same name or differing names for the custom formatter and SQL functions.

Sample `CREATE FUNCTION ... AS` syntax follows:

```
CREATE FUNCTION <sql_function_name>(<arg>[, ...]) RETURNS <return_type>
  AS '<shared_library_path>'[, '<formatter_function_name>']
LANGUAGE C STABLE;
```

You may omit the shared library `.so` extension when you specify `shared_library_path`.

The following command registers a C function named `customfmt_import()` to a SQL UDF named `customfmt_in()` when the function is compiled and linked in a shared library named `customfmt_example.so`:

```
CREATE FUNCTION customfmt_in() RETURNS record
  AS 'customfmt_example.so', 'customfmt_import'
LANGUAGE C STABLE;
```

Using a Custom Formatter in GPSS

Any Greenplum Database external table custom formatter is compatible with, and may be specified for, a value-only GPSS job.

You can use a custom formatter in your GPSS load job by identifying the `custom` data format and providing the formatter function name and parameters in the load configuration file. The names of these properties

differ in version 2 and version 3 (Beta) format configuration files. Version 2 format example:

```
FORMAT: custom
CUSTOM_OPTION:
  NAME: formatter_in
  PARAMSTR: aaa="test",bbb="123"
```

Version 3 (Beta) format example:

```
custom:
  columns:
    - name: value
      type: text
  name: formatter_in
  options:
    - aaa="test"
    - bbb="123"
```

When you specify a custom formatter in your GPSS load configuration file, GPSS invokes the formatter to process the data before loading it into Greenplum Database.

Understanding Transformer Plugins

A transformer plugin is a set of `go` functions that perform specific formatting or processing on data after is read from a Kafka or RabbitMQ data source.

This topic describes transformer plugins and how to use them with Greenplum Streaming Server:

- Developing a Transformer Plugin for GPSS
- Using a Transformer Plugin in GPSS

Developing a Transformer Plugin for GPSS

A transformer plugin is a set of `go` functions. You compile the `go` functions that you develop into a shared library. Users of the plugin specify the file system path to this library in the GPSS load configuration file.

The GPSS transformer plugin framework exposes two entry points:

- Plugin initialization function - GPSS invokes this function once when it loads the transformer plugin.
- Plugin transform function - GPSS invokes this function for every message it reads from the source.

The framework supports specifying properties that direct the processing performed by the transformer. GPSS passes any transformer properties specified in the load configuration file to the `go` functions. The transformer-related load configuration properties are described further in Using a Transformer Plugin in GPSS.

Refer to the gp-stream-server-plugin github repository for an example Greenplum Streaming Server transformer plugin implementation.

Using a Transformer Plugin in GPSS

To use a transformer plugin in a Kafka or RabbitMQ data job, you must specify an `INPUT:TRANSFORMER` (version 2) or `sources:<source>:transformer` (version 3 (Beta)) block in the load configuration file. The

properties in this block identify the file system path to the transformer plugin library, the initialization and transform function names, and transform-specific properties that GPSS passes to the functions.

Version 2 format syntax:

```
[TRANSFORMER:
   PATH: <path_to_plugin_transform_library>
   ON_INIT: <plugin_transform_init_name>
   TRANSFORM: <plugin_transform_name>
   PROPERTIES:
     <plugin_transform_property_name>: <property_value>
     [ ... ] ]
```

Version 3 (Beta) format syntax:

```
transformer:
  path: <path_to_plugin_transform_library>
  on_init: <plugin_transform_init_name>
  transform: <plugin_transform_name>
  properties:
    <plugin_transform_property_name>: <property_value>
    ...
```

When you specify a transformer plugin in your GPSS load configuration file, GPSS invokes the transform function to process the data after it applies an input filter (if specified).

Understanding UDF Transformers

A user-defined function (UDF) transformer is a function that perform specific formatting or processing on data before it is written to Greenplum Database. All GPSS data sources (file, Kafka, RabbitMQ, s3) support UDF transformers.

This topic describes UDF transformers and how to use them with Greenplum Streaming Server:

- Developing a UDF Transformer for GPSS
- Using a UDF Transformer in GPSS

Developing a UDF Transformer for GPSS

A UDF transformer is a function. You create and register the function with Greenpum Database. Users of the UDF specify the name of the function in the GPSS load configuration file.

The framework supports specifying properties that direct the processing performed by the UDF. GPSS passes any transformer properties specified in the load configuration file to the function. The UDF transformer-related load configuration properties are described further in Using a UDF Transformer in GPSS.

A UDF transformer function signature follows:

```
<function_name>(<s> anyelement, <properties> json)
```

The `anyelement` input argument and the function return columns must have the same table structure as the output table.

Using a UDF Transformer in GPSS

To use a UDF transformer in a GPSS job, you must specify an `OUTPUT:TRANSFORMER` (version 2) or `targets:gpdb:tables:table:transformer` (version 3 (Beta)) block in the load configuration file. The properties in this block identify the UDF transform function name and transform-specific properties that GPSS passes to the function.

Version 2 format syntax:

```
[TRANSFORMER:
   PATH: <path_to_plugin_transform_library>
   ON_INIT: <plugin_transform_init_name>
   TRANSFORM: <plugin_transform_name>
   PROPERTIES:
      <plugin_transform_property_name>: <property_value>
      [ ... ] ]
```

Version 3 (Beta) format syntax:

```
transformer:
  transform: <udf_transform_udf_name>
  properties:
    <udf_transform_property_name>: <property_value>
    ...
  columns:
    - <udf_transform_column_name>
    ...
```

GPSS currently supports specifying only one of the `mapping` or (UDF) `transformer` blocks in the load configuration file, not both.

Example

The following UDF adds a prefix to a name and adds an increment to an age. The prefix is specified in a function property named `name-prefix`, the age increment in a property named `age-increment`:

```
CREATE FUNCTION simple_mapping(s anyelement, properties json)
  RETURNS table(id bigint, name text, age int)
  AS $$ SELECT ((s.key)->>'id')::bigint, (properties->>'name-prefix')::text||((s.valu
e)->>'name')::text, (properties->>'ag-increment')::int+(s.value)->>'age'::int;
$$ LANGUAGE sql;
```

To use this UDF in a GPSS job, specify the following in a version 2 load configuration file:

```
TRANSFORMER:
  TRANSFORM: simple_mapping
  PROPERTIES:
    name-prefix: 'Dear '
    age-increment: 10
```

With this configuration, the prefix `Dear` is prepended to the name and `10` is added to the age before the data is written to Greenplum Database

Internally, GPSS writes to the target table using the following SQL command:

```
INSERT INTO tbl_target(id,name,age) FROM
  SELECT f.id, f.name, f.age FROM
    tbl_source s, simple_mapping(s,'{age-increment":"10","name-prefix":"Dear "}'::jso
n) f;
```

Loading Kafka Data into Greenplum

The Greenplum Streaming Server Kafka data source is also known as the *Greenplum-Kafka Integration*.

Apache Kafka is a fault-tolerant, low-latency, distributed publish-subscribe message system. The Greenplum Streaming Server supports loading Kafka data from the Apache and Confluent Kafka distributions. Refer to the Apache Kafka Documentation for more information about Apache Kafka.

A Kafka message may include a key and a value, and may be comprised of a single line or multiple lines. Kafka stores streams of messages (or records) in categories called topics. A Kafka producer publishes records to partitions in one or more topics. A Kafka consumer subscribes to a topic and receives records in the order that they were sent within a given Kafka partition. Kafka does not guarantee the order of data originating from different Kafka partitions.

You can use the gpsscli or gpkafka load utilities to load Kafka data into Greenplum Database.

`gpkafka load` is a wrapper around the Greenplum Streaming Server gpss and gpsscli commands. VMware recommends that you migrate to using these utilities directly.

Both the `gpss` server and the `gpkafka load` utilities are a Kafka consumer. They ingest streaming data from a single Kafka topic, using Greenplum Database readable external tables to transform and insert or update the data into a target Greenplum table. You identify the Kafka source, data format, and the Greenplum connection options and target table definition in a YAML-formatted load configuration file that you provide to the utility. In the case of user interrupt or exit, the utility resumes a subsequent data load operation specifying the same Kafka topic and target Greenplum Database table names from the last recorded offset.

Requirements

The Greenplum Streaming Server requires Kafka version 0.11 or newer for exactly-once delivery assurance. You can run with an older version of Kafka (but lose the exactly-once guarantee) by adding the following `PROPERTIES` or `rdkafka_prop` (v3) block to your `gpkafka.yaml` load configuration file:

```
PROPERTIES:
      api.version.request: false
      broker.version.fallback: 0.8.2.1
```

Load Procedure

You will perform the following tasks when you use the Greenplum Streaming Server to load Kafka data into a Greenplum Database table:

1. Ensure that you meet the Prerequisites.
2. Register the Greenplum Streaming Server extension.
3. Identify the format of the Kafka data.
4. (Optional) Register custom data formatters.
5. Construct the load configuration file.
6. Create the target Greenplum Database table.
7. Assign Greenplum Database role permissions to the table, if required.
8. Run the gpkafka load command to load the Kafka data into Greenplum Database.
9. Check the progress of the load operation.
10. Check for load errors. (Note that the naming format for `gpkafka` log files is `gpkafka_*date*.log`.)

Prerequisites

Before using the `gpsscli` or `gpkafka` utilities to load Kafka data to Greenplum Database, ensure that you:

- Meet the Prerequisites documented for the Greenplum Streaming Server, and configure and start the server.
- Have access to a running Kafka cluster with ZooKeeper, and that you can identify the hostname(s) and port number(s) of the Kafka broker(s) serving the data.
- Can identify the Kafka topic of interest.
- Can run the command on a host that has connectivity to:
    - Each Kafka broker host in the Kafka cluster.
    - The Greenplum Database coordinator and all segment hosts.

About Supported Kafka Message Data Formats

The Greenplum Streaming Server supports Kafka message key and value data in the following formats:

Format	Description
avro	Avro-format data. GPSS supports: • Loading Kafka message key or value data from a single-object encoded Avro file. • Using the Avro schema of a Kafka message key and/or value registered in a Confluent Schema Registry to load Avro-format key and/or value data. • Using the Avro schema specified in a separate `.avsc` file located on each Greenplum Database segment host file system to load Avro-format key or value data, but not both. In all cases, GPSS reads Avro data from Kafka only as a single JSON-type column. GPSS supports `libz`-, `lzma`- and `snappy`-compressed Avro data from Kafka.
binary	Binary format data. GPSS reads binary data from Kafka only as a single bytea-type column.
csv	Comma-delimited text format data.

Format	Description
custom	Data of a custom format, parsed by a custom formatter function.
delimited	Text data separated by a configurable delimiter. The `delimited` data format supports a multi-byte delimiter.
json, jsonl (version 2 only)	JSON- or JSONB-format data. Specify the `json` format when the file is in JSON or JSONB format. GPSS can read JSON data as a single object or can read a single JSON record per line. You must define a mapping if you want GPSS to write the data into specific columns in the target Greenplum Database table.  Note: GPSS supports JSONB-format data only when loading to Greenplum 6.  Note: Specify `FORMAT: jsonl` in version 2 format load configuration files. Specify `json` with `is_jsonl: true` in version 3 (Beta) format load configuration files.

To write Kafka message data into a Greenplum Database table, you must identify the data format in the load configuration file.

Avro

Specify the `avro` format when your Kafka message data is a single-object encoded Avro file or you are using the Confluent Schema Registry to load Avro message key and/or value data. (If the schema registry is SSL-secured, refer to Accessing an SSL-Secured Schema Registry for configuration details.) GPSS reads Avro data from Kafka and loads it into a single JSON-type column. You must define a mapping if you want GPSS to write the data into specific columns in the target Greenplum Database table.

Binary

Use the `binary` format when your Kafka message data is a stream of bytes. GPSS reads binary data from Kafka and loads it into a single bytea-type column.

CSV

Use the `csv` format when your Kafka message data is comma-delimited text and conforms to RFC 4180. The message content may not contain line ending characters (CR and LF).

Data in `csv` format may appear in Kafka messages as follows:

```
"1313131","12","backorder","1313.13"
"3535353","11","shipped","761.35"
"7979797","11","partial","18.72"
```

Custom

The Greenplum Streaming Server provides a custom data formatter plug-in framework for Kafka messages using user-defined functions. The type of Kafka message data processed by a custom formatter is formatter-specific. For example, a custom formatter may process compressed or complex data.

Refer to Custom Formatter for Kafka for an example custom formatter that loads Kafka data into Greenplum Database.

Delimited Text

The Greenplum Streaming Server supports loading Kafka data delimited by one or more characters that you specify. Use the `delimited` format for such data. The delimiter may be a multi-byte value and up to 32 bytes in length. You can also specify quote and escape characters, and an end-of-line prefix.

The delimiter may not contain the quote or escape characters.

When you specify a quote character:

- The left and right quotes are the same.
- Each data element must be quoted. GPSS does not support mixed quoted and unquoted content.
- You must also define an escape character.
- GPSS keeps the original format of any character between the quotes, except the quote and escape characters. This especially applies to the delimiter and `\n`, which do not require additional escape if they are quoted.
- The quote character is presented as the escape character plus the quote character (for example, \").
- The escape character is presented as the escape character plus the escape character (for example, \)
- GPSS parses multiple escape characters from left to right.

When you do not specify a quote character:

- The escape character is optional.
- If you do not specify an escape character, GPSS treats the delimiter as the column separator, and treats any end-of-line prefix plus `\n` as the row separator.
- If you do specify an escape character:
    - GPSS uses the escape character plus the delimiter as the column separator.
    - GPSS uses the escape character plus the end-of-line prefix plus `\n` as the row separator.
    - The escape character plus the escape character is the escape character itself.
    - GPSS parses multiple escape characters from left to right.

Sample data using a pipe ('|') delimiter character follows:

```
1313131|12|backorder|1313.13
3535353|11|shipped|761.35
7979797|11|partial|18.72
```

JSON (single object)

Specify the `json` format when your Kafka message data is in JSON or JSONB format and you want GPSS to read JSON data from Kafka as a single object into a single column (per the JSON specification, newlines

and white space are ignored). You must define a mapping if you want GPSS to write the data into specific columns in the target Greenplum Database table.

GPSS supports JSONB-format data only when loading to Greenplum 6.

JSON (single record per line)

Specify `FORMAT: jsonl` in version 2 format load configuration files or specify `json` with `is_jsonl: true` in version 3 (Beta) format load configuration files when your Kafka message data is in JSON format, single JSON record per line. You must define a mapping if you want GPSS to write the data into specific columns in the target Greenplum Database table.

Sample JSON message data:

```
{ "cust_id": 1313131, "month": 12, "amount_paid":1313.13 }
{ "cust_id": 3535353, "month": 11, "amount_paid":761.35 }
{ "cust_id": 7979797, "month": 11, "amount_paid":18.82 }
```

About Multiple-Line Kafka Messages

A Kafka message may contain a single line or multiple lines.

GPSS supports the following combinations of single and multiple line messages for the key and value data input components:

Key	Value
single-line	none
none	single-line
single-line	single-line
multi-line	none
none	multi-line

GPSS does not support multiple-line messages for both the key and value.

Registering a Custom Formatter

A custom data formatter for Kafka messages is a user-defined function. If you are using a custom formatter, you must create the formatter function and register it in each database in which you will use the function to write Kafka data to Greenplum tables.

Constructing the gpkafka.yaml Configuration File

You configure a data load operation from Kafka to Greenplum Database via a YAML-formatted configuration file. This configuration file includes parameters that identify the source Kafka data and information about the Greenplum Database connection and target table, as well as error and commit thresholds for the operation.

The Greenplum Streaming Server supports three versions of the YAML configuration file: version 1 (deprecated), version 2, and version 3 (Beta). Versions 2 and 3 of the configuration file format supports all features of Version 1 of the configuration file, and introduce support for loading both the Kafka message key and value to Greenplum, as well as loading meta data.

Refer to the gpkafka.yaml reference page for Version 1 configuration file contents and syntax. Refer to the gpkafka-v2.yaml reference page for Version 2 configuration file format and the configuration parameters that this version supports. gpkafka-v3.yaml describes the Version 3 (Beta) format.

Contents of a sample Version 2 YAML configuration file named `loadcfg2.yaml` follows:

```
DATABASE: ops
USER: gpadmin
PASSWORD: changeme
HOST: mdw-1
PORT: 5432
VERSION: 2
KAFKA:
   INPUT:
      SOURCE:
         BROKERS: kbrokerhost1:9092
         TOPIC: customer_expenses2
         PARTITIONS: (1, 2...4, 7)
      VALUE:
         COLUMNS:
           - NAME: c1
             TYPE: json
         FORMAT: avro
         AVRO_OPTION:
           SCHEMA_REGISTRY_ADDR: http://localhost:8081
      KEY:
         COLUMNS:
           - NAME: key
             TYPE: json
         FORMAT: avro
         AVRO_OPTION:
           SCHEMA_REGISTRY_ADDR: http://localhost:8081
      FILTER: (c1->>'month')::int = 11
      ERROR_LIMIT: 25
   OUTPUT:
      SCHEMA: payables
      TABLE: expenses2
      MAPPING:
        - NAME: customer_id
          EXPRESSION: (c1->>'cust_id')::int
        - NAME: newcust
          EXPRESSION: ((c1->>'cust_id')::int > 5000000)::boolean
        - NAME: expenses
          EXPRESSION: (c1->>'expenses')::decimal
        - NAME: tax_due
          EXPRESSION: ((c1->>'expenses')::decimal * .075)::decimal
   METADATA:
      SCHEMA: gpkafka_internal
   COMMIT:
      MINIMAL_INTERVAL: 2000
```

Greenplum Database Options (Version 2-Focused)

You identify the Greenplum Database connection options via the `DATABASE`, `USER`, `PASSWORD`, `HOST`, and `PORT` parameters.

The `VERSION` parameter identifies the version of the GPSS YAML configuration file. The default version is Version 1. You must specify version `2` or version `v3`.

KAFKA:INPUT Options

Specify the Kafka brokers and topic of interest using the `SOURCE` block. *You must create the Kafka topic prior to loading data.* By default, GPSS reads Kafka messages from all partitions. You may specify a single, a comma-separated list, and/or a range of partition numbers to restrict the partitions from which GPSS reads messages. The `PARTITIONS` property is supported only for version 2 and 3 load configuration file formats.

You must configure different jobs that load from the same Kafka topic to the same Greenplum Database table with non-overlapping `PARTITIONS` values.

When you provide a `VALUE` block, you must specify the `COLUMNS` and `FORMAT` parameters. The `VALUE:COLUMNS` block includes the name and type of each data element in the Kafka message. The default source-to-target data mapping behaviour of GPSS is to match a column name as defined in `COLUMNS:NAME` with a column name in the target Greenplum Database `OUTPUT:TABLE`:

- You must identify the Kafka data elements in the order in which they appear in the Kafka message.
- You may specify `NAME: __IGNORED__` to omit a Kafka message value data element from the load operation.
- You must provide the same name for each non-ignored Kafka data element and its associated Greenplum Database table column.
- You must specify an equivalent data type for each non-ignored Kafka data element and its associated Greenplum Database table column.

The `VALUE:FORMAT` keyword identifies the format of the Kafka message value. GPSS supports comma-delimited text format (`csv`) and data that is separated by a configurable delimiter (`delimited`). GPSS also supports binary (`binary`), single object or single record per line JSON/JSONB (`json` or `jsonl`), custom (`custom`), and Avro (`avro`) format value data.

When you provide a `META` block, you must specify a single JSON-type `COLUMNS` and the `FORMAT: json`. Meta data for Kafka includes the following properties:

- `topic` - `text`
- `partition` - `int`
- `offset` - `bigint`
- `timestamp` - `bigint`

When you provide a `KEY` block, you must specify the `COLUMNS` and `FORMAT` parameters. The `KEY:COLUMNS` block includes the name and type of each element of the Kafka message key, and is subject to the same restrictions as identified for `VALUE:COLUMNS` above. The `KEY:FORMAT` keyword identifies the format of the

Kafka message key. GPSS supports `avro`, `binary`, `csv`, `custom`, `delimited`, `json`, and `jsonl` format key data.

The `FILTER` parameter identifies a filter to apply to the Kafka input messages before the data is loaded into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. The message is dropped if the filter evaluates to `false`. The filter string must be a valid SQL conditional expression and may reference one or more `KEY` or `VALUE` column names.

The `ERROR_LIMIT` parameter identifies the number of errors or the error percentage threshold after which GPSS should exit the load operation. The default `ERROR_LIMIT` is zero; the load operation is stopped when the first error is encountered.

KAFKA:OUTPUT Options

You must specify only one of the `OUTPUT` or `OUTPUTS` blocks.

You identify the target Greenplum Database schema name and table name via the `KAFKA:OUTPUT: SCHEMA` and `TABLE` parameters. *You must pre-create the Greenplum Database table before you attempt to load Kafka data.*

The default load mode is to insert Kafka data into the Greenplum Database table. GPSS also supports updating and merging Kafka message data into a Greenplum table. You specify the load `MODE`, the `MATCH_COLUMNS` and `UPDATE_COLUMNS`, and any `UPDATE_CONDITION`s that must be met to merge or update the data. In `MERGE MODE`, you can also specify `ORDER_COLUMNS` to filter out duplicates and a `DELETE_CONDITION`.

You can override the default mapping of the `INPUT VALUE:COLUMNS` and `KEY:COLUMNS` by specifying a `MAPPING` block in which you identify the association between a specific column in the target Greenplum Database table and a Kafka message value or key data element. You can also map the `META` data columns, and map a Greenplum Database table column to a value expression.

When you specify a `MAPPING` block, ensure that you provide entries for all Kafka data elements of interest - GPSS does not automatically match column names when you provide a `MAPPING`.

Loading to Multiple Greenplum Database Tables

(version 2) You must specify only one of the `OUTPUT` or `OUTPUTS` blocks.

If you want to load from a single Kafka topic to multiple Greenplum Database tables, you provide an `OUTPUTS:TABLE` (version 2) or `targets:gpdb:tables:table` (version 3 (Beta)) block for each table, and specify the properties that identify the data targeted to each.

About the Merge Load Mode

`MERGE` mode is similar to an `UPSERT` operation; GPSS may insert new rows in the database, or may update an existing database row that satisfies match and update conditions. GPSS deletes rows in `MERGE` mode when the data satisfies an optional `DELETE_CONDITION` that you specify.

GPSS stages a merge operation in a temporary table, generating the SQL to populate the temp table from the set of `OUTPUT` configuration properties that you provide.

GPSS uses the following algorithm for `MERGE` mode processing:

1. Create a temporary table like the target table.
2. Generate the SQL to insert the source data into the temporary table.
    1. Add the `MAPPINGS`.
    2. Add the `FILTER`.
    3. Use `MATCH_COLUMNS` and `ORDER_COLUMNS` to filter out duplicates.
3. Update the target table from rows in the temporary table that satisfy `MATCH_COLUMNS`, `UPDATE_COLUMNS`, and `UPDATE_CONDITION`.
4. Insert non-matching rows into the target table.
5. Delete rows in the target table that satisfy `MATCH_COLUMNS` and the `DELETE_CONDITION`.
6. Truncate the temporary table.

Other Options

The `KAFKA:METADATA:SCHEMA` parameter specifies the name of the Greenplum Database schema in which GPSS creates external and history tables.

GPSS commits Kafka data to the Greenplum Database table at the row and/or time intervals that you specify in the `KAFKA:COMMIT: MAX_ROW` and/or `MINIMAL_INTERVAL` parameters. If you do not specify these properties, GPSS commits data at the default `MINIMAL_INTERVAL`, 5000ms.

You can configure GPSS to run a task (user-defined function or SQL commands) after GPSS reads a configurable number of batches from Kafka. Use the `KAFKA:TASK: POST_BATCH_SQL` and `BATCH_INTERVAL` configuration parameters to specify the task and the batch interval.

Specify a `KAFKA:PROPERTIES` block to set Kafka consumer configuration properties. GPSS sends the property names and values to Kafka when it instantiates a consumer for the load operation.

About KEYs, VALUEs, and FORMATs

You can specify any data format in the Version 2 configuration file `KEY:FORMAT` and `VALUE:FORMAT` parameters, with some restrictions. The Greenplum Streaming Server supports the following `KEY:FORMAT` and `VALUE:FORMAT` combinations:

KEY:FORMAT	VALUE:FORMAT	Description
any	none (`VALUE` block omitted)	GPSS loads only the Kafka message key data, subject to any `MAPPING` that you specify, to Greenplum Database.

KEY:FORMAT	VALUE:FORMAT	Description
none (`KEY` block omitted)	any	Equivalent to configuration file Version 1. GPSS ignores the Kafka message key and loads only the Kafka message value data, subject to any `MAPPING` that you specify, to Greenplum Database.
csv	any	Not permitted.
any	csv	Not permitted.
avro, binary, delimited, json, jsonl	avro, binary, delimited, json, jsonl	Any combination is permitted. GPSS loads both the Kafka message key and value data, subject to any `MAPPING` that you specify, to Greenplum Database.

About the JSON Format and Column Type

When you specify `FORMAT: json` or `FORMAT: jsonl`, valid `COLUMN:TYPE`s for the data include `json` or `jsonb`. You can also specify the new GPSS `gp_jsonb` (Beta) or `gp_json` (Beta) column types.

- `gp_jsonb` is an enhanced JSONB type that adds support for `\u` escape sequences and unicode. For example, `gp_jsonb` can escape `\uDD8B` and `\u0000` as text format, but `jsonb` treats these characters as illegal.
- `gp_json` is an enhanced JSON type that can tolerate certain illegal unicode sequences. For example, `gp_json` automatically escapes incorrect surrogate pairs and processes `\u0000` as `\\u0000`. Note that unicode escape values cannot be used for code point values above `007F` when the server encoding is not `UTF8`.

You can use the `gp_jsonb` (Beta) and `gp_json` (Beta) data types as follows:

- As the `COLUMN:TYPE` when the target Greenplum Database table column type is `json` or `jsonb`.
- In a `MAPPING` when the target Greenplum Database column is `text` or `varchar`. For example:

  ```
  EXPRESSION: (j->>'a')::text
  ```

- In a `MAPPING` when `FORMAT: avro` and the target Greenplum Database column is `json` or `jsonb`. For example:

  ```
  EXPRESSION: j::gp_jsonb
  ```

  or

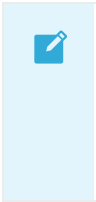
  ```
  EXPRESSION: j::gp_json
  ```

- In a `MAPPING` when `FORMAT: avro` and the target Greenplum Database column is `text` or `varchar`. For example:

  ```
  EXPRESSION: (j::gp_jsonb->>'a')::text
  ```

  or

  ```
  EXPRESSION: (j::gp_json->>'a')::text
  ```



The `gp_jsonb` (Beta) and `gp_json` (Beta) data types are defined in an extension named `dataflow`. You must `CREATE EXTENSION dataflow;` in each database in which you choose to use these (Beta) data types.

Preserving Ill-Formed JSON Escape Sequences

GPSS exposes a configuration parameter that you can use with the `gp_jsonb` and `gp_json` types. The name of this parameter is `gpss.json_preserve_ill_formed_prefix`. When set, GPSS does not return an error when it encounters an ill-formed JSON escape sequence with these types, but instead prepends it with the prefix that you specify.

For example, if `gpss.json_preserve_ill_formed_prefix` is set to the string `"##"` as follows:

```
SET gpss.json_preserve_ill_formed_prefix = "##";
```

and GPSS encounters an ill-formed JSON sequence such as the orphaned low surrogate `\ude04X`, GPSS writes the data as `##\ude04X` instead.

About Transforming and Mapping Kafka Input Data

You can define a `MAPPING` between the Kafka input data (`VALUE:COLUMNS`, `KEY:COLUMNS`, and `META:COLUMNS`) and the columns in the target Greenplum Database table. Defining a mapping may be useful when you have a multi-field input column (such as a JSON-type column), and you want to assign individual components of the input field to specific columns in the target table.

You might also use a `MAPPING` to assign a value expression to a target table column. The expression must be one that you could specify in the `SELECT` list of a query, and can include a constant value, a column reference, an operator invocation, a built-in or user-defined function call, and so forth.

If you choose to map more than one input column in an expression, you can can create a user-defined function to parse and transform the input column and return the columns of interest.

For example, suppose a Kafka producer emits the following JSON messages to a topic:

```
{ "customer_id": 1313131, "some_intfield": 12 }
{ "customer_id": 77, "some_intfield": 7 }
{ "customer_id": 1234, "some_intfield": 56 }
```

You could define a user-defined function, `udf_parse_json()`, to parse the data as follows:

```
=> CREATE OR REPLACE FUNCTION udf_parse_json(value json)
     RETURNS TABLE (x int, y text)
   LANGUAGE plpgsql AS $$
     BEGIN
        RETURN query
        SELECT ((value->>'customer_id')::int), ((value->>'some_intfield')::text);
     END $$;
```

This function returns the two fields in each JSON record, casting the fields to integer and text, respectively.

An example `MAPPING` for the topic data in a JSON-type `KAFKA:INPUT:COLUMNS` named `jdata` follows:

```
MAPPING:
  cust_id: (jdata->>'customer_id')
  field2: ((jdata->>'some_intfield') * .075)::decimal
  j1, j2: (udf_parse_json(jdata)).*
```

The Greenplum Database table definition for this example scenario is:

```
=> CREATE TABLE t1map( cust_id int, field2 decimal(7,2), j1 int, j2 text );
```

About Mapping Avro Bytes Fields to Base64-Encoded Strings

When you specify `AVRO_OPTION:BYTES_TO_BASE64`, GPSS maps Avro bytes fields to base64-encoded strings. You can provide a `MAPPING` to decode these strings and write the data to a Greenplum `bytea` column.

For example, if the Avro schema is:

```
{
    "type": "record",
    "name": "bytes_test",
    "fields": [
                {"name": "id", "type": "long"},
                {"name": "string", "type": "string"},
                {"name": "bytes",  "type": "bytes"},
        {
            "name": "inner_record",
            "type": {
                "type": "map",
                "values": {
                                                "type": "bytes",
                                                "name": "nested_bytes"
                }
            }
        }
    ]
}
```

And if your load configuration file includes these input property settings:

```
VALUE:
    COLUMNS:
      - NAME: c1
        TYPE: json
    FORMAT: avro
    AVRO_OPTION:
      SCHEMA_REGISTRY_ADDR: http://localhost:8081
      BYTES_TO_BASE64: true
```

You can define a `MAPPING` to decode the encoded strings as follows:

```
MAPPING:
  - NAME: id
    EXPRESSION: (c1->>'id')::int
  - NAME: bytes1
    EXPRESSION: (decode(c1->>'bytes', 'base64'))
```

```
  - NAME: bytes2
    EXPRESSION: (decode((c1->>'inner_record')::json->>'nested_bytes', 'base64'))
```

This mapping decodes the `bytes1` and `bytes2` fields to the Greenplum `bytea` data type. GPSS would expect to load these mapped fields to a Greenplum table with the following definition:

```
CREATE TABLE avbyte( id int, bytes1 bytea, bytes2 bytea);
```

Creating the Greenplum Table

You must pre-create the Greenplum table before you load Kafka data into Greenplum Database. You use the `KAFKA:OUTPUT: SCHEMA` and `TABLE` load configuration file parameters to identify the schema and table names.

The target Greenplum table definition must include each column that GPSS will load into the table. The table definition may include additional columns; GPSS ignores these columns, and loads no data into them.

The name and data type that you specify for a column of the target Greenplum Database table must match the name and data type of the related, non-ignored Kafka message element. If you have defined a column mapping, the name of the Greenplum Database column must match the target column name that you specified for the mapping, and the type must match the target column type or expression that you define.

The `CREATE TABLE` command for the target Greenplum Database table receiving the Kafka topic data defined in the `loadcfg2.yaml` file presented in the Constructing the gpkafka.yaml Configuration File section follows:

```
testdb=# CREATE TABLE payables.expenses2( customer_id int8, newcust bool,
           expenses decimal(9,2), tax_due decimal(7,2) );
```

Running the gpkafka load Command

`gpkafka load` is a wrapper around the Greenplum Streaming Server (GPSS) `gpss` and `gpsscli` utilities. Starting in Greenplum Streaming Server version 1.3.2, `gpkafka load` no longer launches a `gpss` server instance, but rather calls the backend server code directly.

When you run `gpkafka load`, the command submits, starts, and stops a GPSS job on your behalf.

VMware recommends that you migrate to using the GPSS utilities directly.

You run the `gpkafka load` command to load Kafka data to Greenplum. When you run the command, you provide the name of the configuration file that defines the parameters of the load operation. For example:

```
$ gpkafka load loadcfg2.yaml
```

The default mode of operation for `gpkafka load` is to read all pending messages and then to wait for, and then consume, new Kafka messages. When running in this mode, `gpkafka load` waits indefinitely; you can interrupt and exit the command with Control-c.

To run the command in batch mode, you provide the `--quit-at-eof` option. In this mode, `gpkafka load` exits when there are no new messages in the Kafka stream.

`gpkafka load` resumes a subsequent data load operation specifying the same Kafka topic and target Greenplum Database table names from the last recorded offset.

Refer to the gpkafka load reference page for additional information about this command.

GPSS cannot detect the addition of a new Kafka partition while a load operation is in progress. You must stop, and then restart the load operation to read Kafka messages published to the new partition.

Configuring the gpfdist Server Instance

The `gpkafka load` command uses the `gpfdist` or `gpfdists` protocol to load data into Greenplum. You can configure the protocol used for the load request by providing the `--config gpfdistconfig.json` option to the command, where `gpfdistconfig.json` identifies a GPSS configuration file that specifies `gpfdist` configuration in a `Gpfdist` protocol block. Refer to Configuring the Greenplum Streaming Server in the Greenplum Streaming Server documentation for detailed information about the file format and properties supported.

`gpkafka load` reads the configuration specified in the `Gpfdist` protocol block of the `gpfdistconfig.json` file; it ignores the GPSS configuration specified in the `ListenAddress` block of the file.

Or, you may choose to provide `gpfdist` host or port configuration settings on the `gpkafka load` command line by specifying the `--gpfdist-host hostaddr` or `--gpfdist-port portnum` options to the command. Any options that you specify on the command line override settings provided in the `gpfdistconfig.json` file.

About Kafka Offsets, Message Retention, and Loading

Kafka maintains a partitioned log for each topic, assigning each record/message within a partition a unique sequential id number. This id is referred to as an *offset*. Kafka retains, for each `gpkafka load` invocation specifying the same Kafka topic and Greenplum Database table names, the last offset within the log consumed by the load operation. The Greenplum Streaming Server also records this offset value. Refer to Understanding Kafka Message Offset Management for more detailed information about how GPSS manages message offsets.

Kafka persists a message for a configurable retention time period and/or log size, after which it purges messages from the log. Kafka topics or messages can also be purged on demand. This may result in an offset mismatch between Kafka and the Greenplum Streaming Server.

`gpkafka load` returns an error if its recorded offset for the Kafka topic and Greenplum Database table combination is behind that of the current earliest Kafka message offset for the topic, or when the earliest and latest offsets do not match.

When you receive one of these messages, you can choose to:

- Resume the load operation from the earliest available message published to the topic by specifying the `--force-reset-earliest` option to `gpkafka load`:

```
$ gpkafka load --force-reset-earliest loadcfg2.yaml
```

- Load only new messages published to the Kafka topic, by specifying the `--force-reset-latest` option with the command:

```
$ gpkafka load --force-reset-latest loadcfg2.yaml
```

- Load messages published since a specific timestamp (milliseconds since epoch), by specifying the `--force-reset-timestamp` option to `gpkafka load`. To determine the create time epoch timestamp for a Kafka message, run the Kafka console consumer on the topic specifying the `--property print.timestamp=true` option, and review the output. You can also use a converter such as EpocConverter to convert a human-readable date to epoch time.

```
$ gpkafka load --force-reset-timestamp 1571066212000 loadcfg2.yaml
```

Specifying the `--force-reset-<xxx>` options when loading data may result in missing or duplicate messages. Use of these options outside of the offset mismatch scenario is discouraged.

Alternatively, you can provide the `FALLBACK_OPTION` (version 2) or `fallback_option` (version 3 (Beta)) property in the load configuration file to instruct GPSS to automatically read from the specified offset when it detects a mismatch.

Checking the Progress of a Load Operation

Starting in version 1.4.1, GPSS keeps track of the progress of each Kafka load job in a separate CSV-format log file. The progress log file for a specific job is named `progress_*jobname*_*jobid*_*date*.log`, and resides in the following log directory:

- If you are loading Kafka data to Greenplum with the `gpkafka load` command, GPSS writes the progress log file to the directory that you specified with the `-l | --log-dir` option to the command, or to the `$HOME/gpAdminLogs` directory.
- If you are loading Kafka data to Greenplum with the `gpsscli` commands, GPSS writes the progress log file to the directory that you specified with the `-l | --log-dir` option *when you started the GPSS server instance*, or to the `$HOME/gpAdminLogs` directory.

A progress log file includes information and statistics about the load time, data size, and speed. It also includes the number of rows written to the Greenplum table, the number of rows rejected by Greenplum, and the total number of rows operated on by GPSS (inserted rows plus rejected rows).

A progress log file includes the following header row:

```
timestamp,pid,batch_id,start_time,end_time,total_byte,speed,total_read_count,inserted_
rows,rejected_rows,total_rows
```

Example Kafka progress log message:

```
20220704 10:17:00.52827,101417,1,2022-07-04 17:16:33.421+00,2022-07-04 17:17:00.497+0
0,79712,2.88KB,997,991,6,997
```

When GPSS reads Kafka data in `jsonl`, `delimited`, or `csv` formats, a Kafka message may contain multiple rows. For these formats, the progress log `total_read_count` identifies the Kafka message number, while `total_rows` identifies the total number of inserted and rejected rows.

Understanding Kafka Message Offset Management

As a Kafka consumer, GPSS must manage the progress of each load operation.

Legacy Consumer

The default behaviour of GPSS is that of a legacy Kafka consumer; it always stores the message offset for each load job in a history table in Greenplum Database.

High-Level Consumer

GPSS can also act as a high-level consumer when you specify a consumer group using the `group.id` Kafka client configuration property. High-level consumers take advantage of Kafka broker-based offset management. When the `enable.auto.commit` Kafka client property is also enabled (the default), GPSS automatically commits offsets to the Kafka broker by group. This allows you to monitor the Kafka consumed offset directly from the broker.

Recall that you specify Kafka client properties in the `PROPERTIES` (version 2) and `rdkafka_prop` (version 3 (Beta)) load configuration file block. For example:

```
PROPERTIES:
  group.id: gpss
```

Or,

```
rdkafka_prop:
  group.id: gpss
  enable.auto.commit: false
```

When acting as a high-level consumer, GPSS uses the `CONSISTENCY` (version 2) or `consistency` (version 3 (Beta)) load configuration file property and client `enable.auto.commit` settings to govern how it manages offsets. The `CONSISTENCY/consistency` setting identifies how, when (before commit, after commit, or never), and where (history table, broker, both, nowhere) GPSS writes the offset.

GPSS supports the following `CONSISTENCY` settings:

```
CONSISTENCY: { strong | at-least | at-most| none }
```

Summary

The following table summarizes the offset commit behaviour of GPSS:

Consistency Value	Legacy Consumer	High-Level Consumer
strong [or empty]	GPSS stores offsets in a history table.	GPSS stores offsets in both a history table and the Kafka broker.
at-least	GPSS stores offsets in a history table.	GPSS stores offsets in the Kafka broker before `Commit()`.
at-most	GPSS stores offsets in a history table.	GPSS stores offsets in the broker after `Commit()`.
none	GPSS stores offsets in a history table.	When `enable.auto.commit=true`, GPSS stores offsets in the broker automatically. When `enable.auto.commit=false`, GPSS does not store offsets anywhere.

Accessing an SSL-Secured Schema Registry

You must specify certain configuration properties when your Kafka data load operation accesses a secured Confluent Schema Registry service. GPSS exposes these properties in the `AVRO_OPTION:` block of the version 2 Kafka load configuration file, and the `avro:` block of the version 3 (beta) Kafka load configuration file.

About the Configuration Properties

You can specify the following version 2 configuration properties to identify the certificates and keys required to access an SSL-secured schema registry service:

The version 3 configuration property names are lowercase.

- SCHEMA_CA_ON_GPDB - The file system path to the CA certificate that GPSS uses to verify the peer.
- SCHEMA_CERT_ON_GPDB - The file system path to the client certificate that GPSS uses to connect to the HTTPS schema registry.
- SCHEMA_KEY_ON_GPDB - The file system path to the private key file that GPSS uses to connect to the HTTPS schema registry.
- SCHEMA_MIN_TLS_VERSION - The minimum transport layer security (TLS) version that GPSS requests on the connection to the registry. The default minimum TLS version is `1.0`; you can specify `1.0`, `1.1`, `1.2`, or `1.3`.

The schema registry's `ssl.client.auth` property controls client authentication requirements for the service:

- When `ssl.client.auth=false` for the registry, you need only specify the `SCHEMA_CA_ON_GPDB`.
- When `ssl.client.auth=true` for the registry, you must also specify `SCHEMA_CERT_ON_GPDB` and `SCHEMA_KEY_ON_GPDB` in addition to the `SCHEMA_CA_ON_GPDB`.

All certificate and key files must reside in the specified location on all Greenplum Database segment hosts.

Be sure to also specify the `SCHEMA_MIN_TLS_VERSION` if the default value of `1.0` is not sufficient for your requirements.

Additional Considerations

Take the following into consideration when you use GPSS to access an SSL-secured Kafka schema registry:

- Even though you can specify multiple registry addresses in `SCHEMA_REGISTRY_ADDRESS`, GPSS supports specifying only a single set of SSL certificate and key properties. GPSS uses the specified (same) CA, certificate, and key regardless of the registry accessed.
- The file system paths that you specify for the CA, certificate, and key are limited to 64 characters each.

Examples

The following examples illustrate how to load different formats of data into Greenplum Database using the `gpkafka` and `gpsscli` utilities:

- Loading CSV Data from Kafka
- Loading JSON Data from Kafka (Simple)
- Loading JSON Data from Kafka (with Mapping)
- Loading Avro Data from Kafka
- Loading JSON Data from Kafka Using gpsscli
- Merging Data from Kafka into Greenplum Using gpsscli

Loading CSV Data from Kafka

In this example, you load data from a Kafka topic named `topic_for_gpkafka` into a Greenplum Database table named `data_from_kafka`. You perform the load as the Greenplum role `gpadmin`. The table `data_from_kafka` resides in the `public` schema in a Greenplum database named `testdb`.

A producer of the Kafka `topic_for_gpkafka` topic emits customer expense messages in CSV format that include the customer identifier (integer), the month (integer), and an expense amount (decimal). For example, a message for a customer with identifier 123 who spent $456.78 in the month of September follows:

```
"123","09","456.78"
```

You will run a Kafka console producer to emit customer expense messages, and use the Greenplum Streaming Server `gpkafka load` command to transform and load the data into the `data_from_kafka` table and verify the load operation.

Prerequisites

Before you start this procedure, ensure that you:

- Have administrative access to running Kafka and Greenplum Database clusters.
- Have configured connectivity as described in the loading Prerequisites.
- Identify and note the ZooKeeper hostname and port.
- Identify and note the hostname and port of the Kafka broker(s).
- Identify and note the hostname and port of the Greenplum Database coordinator node.

This procedure assumes that you have installed the Apache Kafka distribution. If you are using a different Kafka distribution, you may need to adjust certain commands in the procedure.

Procedure

1. Log in to a host in your Kafka cluster. For example:

   ```
   $ ssh kafkauser@kafkahost
   kafkahost$
   ```

2. Create a Kafka topic named `topic_for_gpkafka`. For example:

   ```
   kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-topics.sh --create \
       --zookeeper localhost:2181 --replication-factor 1 --partitions 1 \
       --topic topic_for_gpkafka
   ```

3. Open a file named `sample_data.csv` in the editor of your choice. For example:

   ```
   kafkahost$ vi sample_data.csv
   ```

4. Copy/paste the following text to add CSV-format data into the file, and then save and exit:

   ```
   "1313131","12","1313.13"
   "3535353","11","761.35"
   "7979797","10","4489.00"
   "7979797","11","18.72"
   "3535353","10","6001.94"
   "7979797","12","173.18"
   "1313131","10","492.83"
   "3535353","12","81.12"
   "1313131","11","368.27"
   ```

5. Stream the contents of the `sample_data.csv` file to a Kafka console producer. For example:

   ```
   kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-console-producer.sh \
       --broker-list localhost:9092 \
       --topic topic_for_gpkafka < sample_data.csv
   ```

6. Verify that the Kafka console producer published the messages to the topic by running a Kafka console consumer. For example:

   ```
   kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-console-consumer.sh \
       --bootstrap-server localhost:9092 --topic topic_for_gpkafka \
   ```

```
    --from-beginning
```

7. Open a new terminal window, log in to the Greenplum Database coordinator host as the `gpadmin` administrative user, and set up the Greenplum environment. For example:

```
$ ssh gpadmin@gpcoord
gpcoord$ . /usr/local/greenplum-db/greenplum_path.sh
```

8. Construct the load configuration file. Open a file named `firstload_cfg.yaml` in the editor of your choice. For example:

```
gpcoord$ vi firstload_cfg.yaml
```

9. Fill in the load configuration parameter values based on your environment. This example assumes:
    - Your Greenplum Database coordinator hostname is `gpcoord`.
    - The Greenplum Database server is running on the default port.
    - Your Kafka broker host and port is `localhost:9092`.
    - You want to write the Kafka data to a Greenplum Database table named `data_from_kafka` located in the `public` schema of a database named `testdb`.
    - You want to write the customer identifier and expenses data to Greenplum. You also want to calculate and write the tax due (7.25%) on the expense data. The `firstload_cfg.yaml` file would include the following contents:

```
DATABASE: testdb
USER: gpadmin
HOST: gpcoord
PORT: 5432
KAFKA:
   INPUT:
     SOURCE:
        BROKERS: localhost:9092
        TOPIC: topic_for_gpkafka
     COLUMNS:
        - NAME: cust_id
          TYPE: int
        - NAME: __IGNORED__
          TYPE: int
        - NAME: expenses
          TYPE: decimal(9,2)
     FORMAT: csv
     ERROR_LIMIT: 125
   OUTPUT:
     TABLE: data_from_kafka
     MAPPING:
        - NAME: customer_id
          EXPRESSION: cust_id
        - NAME: expenses
          EXPRESSION: expenses
        - NAME: tax_due
          EXPRESSION: expenses * .0725
   COMMIT:
     MINIMAL_INTERVAL: 2000
```

10. Create the target Greenplum Database table named `data_from_kafka`. For example:

```
gpcoord$ psql -d testdb

testdb=# CREATE TABLE data_from_kafka( customer_id int8, expenses decimal(9,2),
           tax_due decimal(7,2) );
```

11. Exit the `psql` subsystem:

```
testdb=# \q
```

12. Run the `gpkafka load` command to batch load the CSV data published to the `topic_for_gpkafka` topic into the Greenplum table. For example:

```
gpcoord$ gpkafka load --quit-at-eof ./firstload_cfg.yaml
```

    The command exits after it reads all data published to the topic.

13. Examine the command output, looking for messages identifying the number of rows inserted/rejected. For example:

```
... -[INFO]:- ... Inserted 9 rows
... -[INFO]:- ... Rejected 0 rows
```

14. Run the `gpkafka load` command again, this time in streaming mode. For example:

```
gpcoord$ gpkafka load ./firstload_cfg.yaml
```

    The command waits for a producer to publish new messages to the topic.

15. Navigate back to your Kafka host terminal window. Stream the contents of the `sample_data.csv` file to the Kafka console producer once more:

```
kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-console-producer.sh \
--broker-list localhost:9092 \
--topic topic_for_gpkafka < sample_data.csv
```

16. Notice the activity in your Greenplum Database coordinator terminal window. `gpkafka load` consumes the new round of messages and waits.

17. Interrupt and exit the waiting `gpkafka load` command by entering Control-c in the Greenplum Database coordinator host terminal window.

18. View the contents of the Greenplum Database target table `data_from_kafka`:

```
gpcoord$ psql -d testdb

testdb=# SELECT * FROM data_from_kafka WHERE customer_id='1313131'
           ORDER BY expenses;
 customer_id | expenses | tax_due
-------------+----------+---------
     1313131 |   368.27 |   26.70
     1313131 |   368.27 |   26.70
     1313131 |   492.83 |   35.73
```

```
     1313131 |   492.83 |   35.73
     1313131 |  1313.13 |   95.20
     1313131 |  1313.13 |   95.20
(6 rows)
```

The table contains two entries for each expense because the producer published the `sample_data.csv` file twice.

Loading JSON Data from Kafka (Simple)

In this example, you load JSON format data from a Kafka topic named `topic_json` into a single column Greenplum Database table named `single_json_column`. You perform the load as the Greenplum role `gpadmin`. The table `single_json_column` resides in the `public` schema in a Greenplum database named `testdb`.

A producer of the Kafka `topic_json` topic emits customer expense messages in JSON format that include the customer identifier (integer), the month (integer), and an expense amount (decimal). For example, a message for a customer with identifier 123 who spent $456.78 in the month of September follows:

```
{ "cust_id": 123, "month": 9, "amount_paid":456.78 }
```

You will run a Kafka console producer to emit JSON-format customer expense messages, and use the Greenplum Streaming Server `gpkafka load` command to load each Kafka message into a row in the `single_json_column` table.

Prerequisites

Before you start this procedure, ensure that you:

- Have administrative access to running Kafka and Greenplum Database clusters.
- Have configured connectivity as described in the loading Prerequisites.
- Identify and note the ZooKeeper hostname and port.
- Identify and note the hostname and port of the Kafka broker(s).
- Identify and note the hostname and port of the Greenplum Database coordinator node.

This procedure assumes that you have installed the Apache Kafka distribution. If you are using a different Kafka distribution, you may need to adjust certain commands in the procedure.

Procedure

1. Login to a host in your Kafka cluster. For example:

   ```
   $ ssh kafkauser@kafkahost
   kafkahost$
   ```

2. Create a Kafka topic named `topic_json`. For example:

   ```
   kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-topics.sh --create \
       --zookeeper localhost:2181 --replication-factor 1 --partitions 1 \
   ```

```
    --topic topic_json
```

3. Open a file named `sample_data.json` in the editor of your choice. For example:

```
kafkahost$ vi sample_data.json
```

4. Copy/paste the following text to add JSON-format data into the file, and then save and exit:

```
{ "cust_id": 1313131, "month": 12, "expenses": 1313.13 }
{ "cust_id": 3535353, "month": 11, "expenses": 761.35 }
{ "cust_id": 7979797, "month": 10, "expenses": 4489.00 }
{ "cust_id": 7979797, "month": 11, "expenses": 18.72 }
{ "cust_id": 3535353, "month": 10, "expenses": 6001.94 }
{ "cust_id": 7979797, "month": 12, "expenses": 173.18 }
{ "cust_id": 1313131, "month": 10, "expenses": 492.83 }
{ "cust_id": 3535353, "month": 12, "expenses": 81.12 }
{ "cust_id": 1313131, "month": 11, "expenses": 368.27 }
```

5. Stream the contents of the `sample_data.json` file to a Kafka console producer. For example:

```
kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-console-producer.sh \
    --broker-list localhost:9092 \
    --topic topic_json < sample_data.json
```

6. Verify that the Kafka console producer published the messages to the topic by running a Kafka console consumer. For example:

```
kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-console-consumer.sh \
    --bootstrap-server localhost:9092 --topic topic_json \
    --from-beginning
```

7. Open a new terminal window, log in to the Greenplum Database coordinator host as the `gpadmin` administrative user, and set up the Greenplum environment. For example:

```
$ ssh gpadmin@gpcoord
gpcoord$ . /usr/local/greenplum-db/greenplum_path.sh
```

8. Construct the load configuration file. Open a file named `simple_jsonload_cfg.yaml` in the editor of your choice. For example:

```
gpcoord$ vi simple_jsonload_cfg.yaml
```

9. Fill in the load configuration parameter values based on your environment. This example assumes:
    - Your Greenplum Database coordinator hostname is `gpcoord`.
    - The Greenplum Database server is running on the default port.
    - Your Kafka broker host and port is `localhost:9092`.
    - You want to write the Kafka data to a Greenplum Database table named `single_json_column` located in the `public` schema of a database named `testdb`.
    - You want to write the data to Greenplum as a single `json` type column. The `simple_jsonload_cfg.yaml` file would include the following contents:

```
DATABASE: testdb
USER: gpadmin
HOST: gpcoord
PORT: 5432
KAFKA:
   INPUT:
     SOURCE:
        BROKERS: localhost:9092
        TOPIC: topic_json
     FORMAT: json
     ERROR_LIMIT: 10
   OUTPUT:
     TABLE: single_json_column
   COMMIT:
     MINIMAL_INTERVAL: 1000
```

10. Create the target Greenplum Database table named `single_json_column`. For example:

```
gpcoord$ psql -d testdb

testdb=# CREATE TABLE single_json_column( value json );
```

11. Exit the `psql` subsystem:

```
testdb=# \q
```

12. Run the `gpkafka load` command to batch load the JSON data published to the `topic_json` topic into the Greenplum table. For example:

```
gpcoord$ gpkafka load --quit-at-eof ./simple_jsonload_cfg.yaml
```

    The command exits after it reads all data published to the topic.

13. Examine the command output, looking for messages that identify the number of rows inserted/rejected. For example:

```
... -[INFO]:- ... Inserted 9 rows
... -[INFO]:- ... Rejected 0 rows
```

14. View the contents of the Greenplum Database target table `single_json_column`:

```
gpcoord$ psql -d testdb

testdb=# SELECT * FROM single_json_column;
                          value
-------------------------------------------------------------
 { "cust_id": 7979797, "month": 10, "expenses": 4489.00 }
 { "cust_id": 7979797, "month": 11, "expenses": 18.72 }
 { "cust_id": 3535353, "month": 12, "expenses": 81.12 }
 { "cust_id": 3535353, "month": 11, "expenses": 761.35 }
 { "cust_id": 1313131, "month": 12, "expenses": 1313.13 }
 { "cust_id": 3535353, "month": 10, "expenses": 6001.94 }
 { "cust_id": 7979797, "month": 12, "expenses": 173.18 }
 { "cust_id": 1313131, "month": 10, "expenses": 492.83 }
```

```
 { "cust_id": 1313131, "month": 11, "expenses": 368.27 }
(9 rows)
```

15. Use `json` operators to view the expenses associated with a specific customer:

```
 testdb=# SELECT (value->>'expenses')::decimal AS expenses FROM single_json_col
umn
            WHERE (value->>'cust_id')::int = 1313131;
 expenses
----------
  1313.13
   492.83
   368.27
(3 rows)
```

Loading JSON Data from Kafka (with Mapping)

In this example, you load JSON format data from a Kafka topic named `topic_json_gpkafka` into a Greenplum Database table named `json_from_kafka`. You perform the load as the Greenplum role `gpadmin`. The table `json_from_kafka` resides in the `public` schema in a Greenplum database named `testdb`.

A producer of the Kafka `topic_json_gpkafka` topic emits customer expense messages in JSON format that include the customer identifier (integer), the month (integer), and an expense amount (decimal). For example, a message for a customer with identifier 123 who spent $456.78 in the month of September follows:

```
{ "cust_id": 123, "month": 9, "amount_paid":456.78 }
```

You will run a Kafka console producer to emit JSON-format customer expense messages, and use the Greenplum Streaming Server `gpkafka load` command to transform and load the data into the `json_from_kafka` table.

Prerequisites

Before you start this procedure, ensure that you:

- Have administrative access to running Kafka and Greenplum Database clusters.
- Have configured connectivity as described in the loading Prerequisites.
- Identify and note the ZooKeeper hostname and port.
- Identify and note the hostname and port of the Kafka broker(s).
- Identify and note the hostname and port of the Greenplum Database coordinator node.

This procedure assumes that you have installed the Apache Kafka distribution. If you are using a different Kafka distribution, you may need to adjust certain commands in the procedure.

Procedure

1. Login to a host in your Kafka cluster. For example:

```
$ ssh kafkauser@kafkahost
kafkahost$
```

2. Create a Kafka topic named `topic_json_gpkafka`. For example:

```
kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-topics.sh --create \
    --zookeeper localhost:2181 --replication-factor 1 --partitions 1 \
    --topic topic_json_gpkafka
```

3. Open a file named `sample_data.json` in the editor of your choice. For example:

```
kafkahost$ vi sample_data.json
```

4. Copy/paste the following text to add JSON-format data into the file, and then save and exit:

```
{ "cust_id": 1313131, "month": 12, "expenses": 1313.13 }
{ "cust_id": 3535353, "month": 11, "expenses": 761.35 }
{ "cust_id": 7979797, "month": 10, "expenses": 4489.00 }
{ "cust_id": 7979797, "month": 11, "expenses": 18.72 }
{ "cust_id": 3535353, "month": 10, "expenses": 6001.94 }
{ "cust_id": 7979797, "month": 12, "expenses": 173.18 }
{ "cust_id": 1313131, "month": 10, "expenses": 492.83 }
{ "cust_id": 3535353, "month": 12, "expenses": 81.12 }
{ "cust_id": 1313131, "month": 11, "expenses": 368.27 }
```

5. Stream the contents of the `sample_data.json` file to a Kafka console producer. For example:

```
kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-console-producer.sh \
    --broker-list localhost:9092 \
    --topic topic_json_gpkafka < sample_data.json
```

6. Verify that the Kafka console producer published the messages to the topic by running a Kafka console consumer. For example:

```
kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-console-consumer.sh \
    --bootstrap-server localhost:9092 --topic topic_json_gpkafka \
    --from-beginning
```

7. Open a new terminal window, log in to the Greenplum Database coordinator host as the `gpadmin` administrative user, and set up the Greenplum environment. For example:

```
$ ssh gpadmin@gpcoord
gpcoord$ . /usr/local/greenplum-db/greenplum_path.sh
```

8. Construct the load configuration file. Open a file named `jsonload_cfg.yaml` in the editor of your choice. For example:

```
gpcoord$ vi jsonload_cfg.yaml
```

9. Fill in the load configuration parameter values based on your environment. This example assumes:
    - Your Greenplum Database coordinator hostname is `gpcoord`.
    - The Greenplum Database server is running on the default port.

- Your Kafka broker host and port is `localhost:9092`.
- You want to write the Kafka data to a Greenplum Database table named `json_from_kafka` located in the `public` schema of a database named `testdb`.
- You want to write the customer identifier and expenses data to Greenplum. The `jsonload_cfg.yaml` file would include the following contents:

```
DATABASE: testdb
USER: gpadmin
HOST: gpcoord
PORT: 5432
KAFKA:
   INPUT:
     SOURCE:
        BROKERS: localhost:9092
        TOPIC: topic_json_gpkafka
     COLUMNS:
        - NAME: jdata
          TYPE: json
     FORMAT: json
     ERROR_LIMIT: 10
   OUTPUT:
     TABLE: json_from_kafka
     MAPPING:
        - NAME: customer_id
          EXPRESSION: (jdata->>'cust_id')::int
        - NAME: month
          EXPRESSION: (jdata->>'month')::int
        - NAME: amount_paid
          EXPRESSION: (jdata->>'expenses')::decimal
   COMMIT:
     MINIMAL_INTERVAL: 2000
```

10. Create the target Greenplum Database table named `json_from_kafka`. For example:

```
gpcoord$ psql -d testdb

testdb=# CREATE TABLE json_from_kafka( customer_id int8, month int4, amount_pai
d decimal(9,2) );
```

11. Exit the `psql` subsystem:

```
testdb=# \q
```

12. Run the `gpkafka load` command to batch load the JSON data published to the `topic_json_gpkafka` topic into the Greenplum table. For example:

```
gpcoord$ gpkafka load --quit-at-eof ./jsonload_cfg.yaml
```

    The command exits after it reads all data published to the topic.

13. Examine the command output, looking for messages that identify the number of rows inserted/rejected. For example:

```
... -[INFO]:- ... Inserted 9 rows
... -[INFO]:- ... Rejected 0 rows
```

14. View the contents of the Greenplum Database target table `json_from_kafka`:

```
gpcoord$ psql -d testdb

testdb=# SELECT * FROM json_from_kafka WHERE customer_id='1313131'
           ORDER BY amount_paid;
 customer_id | month | amount_paid
-------------+-------+-------------
     1313131 |    11 |      368.27
     1313131 |    10 |      492.83
     1313131 |    12 |     1313.13
(3 rows)
```

Loading Avro Data from Kafka

In this example, you load Avro-format key and value data as JSON from a Kafka topic named `topic_avrokv` into a Greenplum Database table named `avrokv_from_kafka`. You perform the load as the Greenplum role `gpadmin`. The table `avrokv_from_kafka` resides in the `public` schema in a Greenplum database named `testdb`.

A producer of the Kafka `topic_avrokv` topic emits customer expense messages in JSON format that include the customer identifier (integer), the year (integer), and one or more expense amounts (decimal). For example, a message with key 1 for a customer with identifier 123 who spent $456.78 and $67.89 in the year 1997 follows:

```
1       { "cust_id": 123, "year": 1997, "expenses":[456.78, 67.89] }
```

You will use the Confluent Schema Registry and run a Kafka Avro console producer to emit keys and Avro JSON-format customer expense messages, and use the Greenplum Streaming Server `gpkafka load` command to load the data into the `avrokv_from_kafka` table.

Prerequisites

Before you start this procedure, ensure that you:

- Have administrative access to running Confluent Kafka and Greenplum Database clusters
- Have configured connectivity as described in the loading Prerequisites.
- Identify and note the ZooKeeper hostname and port.
- Identify and note the address of the Confluent Schema Registry server(s).
- Identify and note the hostname and port of the Kafka broker(s).
- Identify and note the hostname and port of the Greenplum Database coordinator node.

This procedure assumes that you have installed the Confluent Kafka distribution.

Procedure

1. Login to a host in your Kafka cluster. For example:

```
$ ssh kafkauser@kafkahost
kafkahost$
```

2. Create a Kafka topic named `topic_json_gpkafka`. For example:

```
kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-topics --create \
    --zookeeper localhost:2181 --replication-factor 1 --partitions 1 \
    --topic topic_avrokv
```

3. Start a Kafka Avro console producer. You will manually input message data to this producer. For example:

```
kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-avro-console-producer \
    --broker-list localhost:9092 \
    --topic topic_avrokv \
    --property parse.key=true --property key.schema='{"type" : "int", "name" :
"id"}' \
    --property value.schema='{ "type" : "record", "name" : "example_schema", "n
amespace" : "com.example", "fields" : [ { "name" : "cust_id", "type" : "int",
"doc" : "Id of the customer account" }, { "name" : "year", "type" : "int", "do
c" : "year of expense" }, { "name" : "expenses", "type" : {"type": "array", "it
ems": "float"}, "doc" : "Expenses for the year" } ], "doc:" : "A basic schema f
or storing messages" }'
```

   The producer waits for messages.

4. Input the following messages to the Avro console producer.

   You must enter a tab between the key and value. Replace `TAB` with a tab.

```
1 TAB {"cust_id":1313131, "year":2012, "expenses":[1313.13, 2424.24]}
2 TAB {"cust_id":3535353, "year":2011, "expenses":[761.35, 92.18, 14.41]}
3 TAB {"cust_id":7979797, "year":2011, "expenses":[4489.00]}
```

5. Verify that the Kafka Avro console producer published the messages to the topic by running a Kafka Avro console consumer. Specify the `print.key` property to have the consumer display the Kafka key. For example:

```
kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-avro-console-consumer \
    --bootstrap-server localhost:9092 --topic topic_avrokv \
    --from-beginning --property print.key=true
```

6. Open a new terminal window, log in to the Greenplum Database coordinator host as the `gpadmin` administrative user, and set up the Greenplum environment. For example:

```
$ ssh gpadmin@gpcoord
gpcoord$ . /usr/local/greenplum-db/greenplum_path.sh
```

7. Construct the load configuration file. Open a file named `avrokvload_cfg.yaml` in the editor of your choice. For example:

```
gpcoord$ vi avrokvload_cfg.yaml
```

8. Fill in the load configuration parameter values based on your environment. This example assumes:
    - Your Greenplum Database coordinator hostname is `gpcoord`.
    - The Greenplum Database server is running on the default port.
    - Your Kafka broker host and port is `localhost:9092`.
    - Your Confluent Schema Registry address is `http://localhost:8081`. The `avrokvload_cfg.yaml` file might include the following contents:

```
DATABASE: testdb
USER: gpadmin
HOST: gpcoord
PORT: 5432
VERSION: 2
KAFKA:
   INPUT:
     SOURCE:
        BROKERS: localhost:9092
        TOPIC: topic_avrokv
     VALUE:
        COLUMNS:
          - NAME: c1
            TYPE: json
        FORMAT: avro
        AVRO_OPTION:
          SCHEMA_REGISTRY_ADDR: http://localhost:8081
     KEY:
        COLUMNS:
          - NAME: id
            TYPE: json
        FORMAT: avro
        AVRO_OPTION:
          SCHEMA_REGISTRY_ADDR: http://localhost:8081
     ERROR_LIMIT: 0
   OUTPUT:
     TABLE: avrokv_from_kafka
     MAPPING:
        - NAME: id
          EXPRESSION: id
        - NAME: customer_id
          EXPRESSION: (c1->>'cust_id')::int
        - NAME: year
          EXPRESSION: (c1->>'year')::int
        - NAME: expenses
          EXPRESSION: array(select json_array_elements(c1->'expenses')::text::f
loat)
   COMMIT:
     MINIMAL_INTERVAL: 2000
```

    The mapping in this configuration assigns each message value field to a separate column and ignores the message key.

9. Create the target Greenplum Database table named `avrokv_from_kafka`. For example:

```
gpcoord$ psql -d testdb

testdb=# CREATE TABLE avrokv_from_kafka( id json, customer_id int, year int, ex
penses decimal(9,2)[] );
```

10. Exit the `psql` subsystem:

```
testdb=# \q
```

11. Run the `gpkafka load` command to batch load the JSON data published to the `topic_json_gpkafka` topic into the Greenplum table. For example:

```
gpcoord$ gpkafka load --quit-at-eof ./avrokvload_cfg.yaml
```

The command exits after it reads all data published to the topic.

12. Examine the command output, looking for messages that identify the number of rows inserted/rejected. For example:

```
... -[INFO]:- ... Inserted 3 rows
... -[INFO]:- ... Rejected 0 rows
```

13. View the contents of the Greenplum Database target table `avrokv_from_kafka`:

```
gpcoord$ psql -d testdb

testdb=# SELECT * FROM avrokv_from_kafka ORDER BY customer_id;
 id | customer_id | year |       expenses
----+-------------+------+----------------------
 1  |     1313131 | 2012 | {1313.13,2424.24}
 2  |     3535353 | 2011 | {761.35,92.18,14.41}
 3  |     7979797 | 2011 | {4489.00}
(3 rows)
```

Loading JSON Data from Kafka Using gpsscli

This example uses the Greenplum Streaming Server client utility, `gpsscli`, rather than the `gpkafka` utility, to load JSON-format data from Kafka into Greenplum Database.

In this example, you load JSON format data from a Kafka topic named `topic_json_gpkafka` into a Greenplum Database table named `json_from_kafka`. You perform the load as the Greenplum role `gpadmin`. The table `json_from_kafka` resides in the `public` schema in a Greenplum database named `testdb`.

A producer of the Kafka `topic_json_gpkafka` topic emits customer expense messages in JSON format that include the customer identifier (integer), the month (integer), and an expense amount (decimal). For example, a message for a customer with identifier 123 who spent $456.78 in the month of September follows:

```
{ "cust_id": 123, "month": 9, "amount_paid":456.78 }
```

You will run a Kafka console producer to emit JSON-format customer expense messages, start a Greenplum Streaming Server instance, and use the GPSS `gpsscli` subcommands to load the data into the `json_from_kafka` table.

Prerequisites

Before you start this procedure, ensure that you:

- Have administrative access to running Kafka and Greenplum Database clusters.
- Have configured connectivity as described in both the Greenplum Streaming Server Prerequisites section and the Kafka Prerequisites.
- Identify and note the ZooKeeper hostname and port.
- Identify and note the hostname and port of the Kafka broker(s).
- Identify and note the hostname and port of the Greenplum Database coordinator node.
- Register the GPSS extension.

This procedure assumes that you have installed the Apache Kafka distribution. If you are using a different Kafka distribution, you may need to adjust certain commands in the procedure.

Procedure

1. Login to a host in your Kafka cluster. For example:

   ```
   $ ssh kafkauser@kafkahost
   kafkahost$
   ```

2. Create a Kafka topic named `topic_json_gpkafka`. For example:

   ```
   kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-topics.sh --create \
       --zookeeper localhost:2181 --replication-factor 1 --partitions 1 \
       --topic topic_json_gpkafka
   ```

3. Open a file named `sample_data.json` in the editor of your choice. For example:

   ```
   kafkahost$ vi sample_data.json
   ```

4. Copy/paste the following text to add JSON-format data into the file, and then save and exit:

   ```
   { "cust_id": 1313131, "month": 12, "expenses": 1313.13 }
   { "cust_id": 3535353, "month": 11, "expenses": 761.35 }
   { "cust_id": 7979797, "month": 10, "expenses": 4489.00 }
   { "cust_id": 7979797, "month": 11, "expenses": 18.72 }
   { "cust_id": 3535353, "month": 10, "expenses": 6001.94 }
   { "cust_id": 7979797, "month": 12, "expenses": 173.18 }
   { "cust_id": 1313131, "month": 10, "expenses": 492.83 }
   { "cust_id": 3535353, "month": 12, "expenses": 81.12 }
   { "cust_id": 1313131, "month": 11, "expenses": 368.27 }
   ```

5. Stream the contents of the `sample_data.json` file to a Kafka console producer. For example:

```
kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-console-producer.sh \
    --broker-list localhost:9092 \
    --topic topic_json_gpkafka < sample_data.json
```

6. Verify that the Kafka console producer published the messages to the topic by running a Kafka console consumer. For example:

```
kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-console-consumer.sh \
    --bootstrap-server localhost:9092 --topic topic_json_gpkafka \
    --from-beginning
```

7. Open a new terminal window, log in to the Greenplum Database coordinator host as the `gpadmin` administrative user, and set up the Greenplum environment. For example:

```
$ ssh gpadmin@gpcoord
gpcoord$ . /usr/local/greenplum-db/greenplum_path.sh
```

8. Construct the Greenplum Streaming Server configuration file. For example, open a file named `gpsscfg_ex.json` in the editor of your choice:

```
gpcoord$ vi gpsscfg_ex.json
```

9. Designate a GPSS listen port number of 5019 and a `gpfdist` port number of 8319 in the configuration file. For example, copy/paste the following into the `gpsscfg_ex.json` file, and then save and exit the editor:

```
{
    "ListenAddress": {
        "Host": "",
        "Port": 5019
    },
    "Gpfdist": {
        "Host": "",
        "Port": 8319
    }
}
```

10. Start the Greenplum Streaming Server instance in the background, specifying the log directory `./gpsslogs`. For example:

```
gpcoord$ gpss --config gpsscfg_ex.json --log-dir ./gpsslogs &
```

11. Construct the load configuration file. Open a file named `jsonload_cfg.yaml` in the editor of your choice. For example:

```
gpcoord$ vi jsonload_cfg.yaml
```

12. Fill in the load configuration parameter values based on your environment. This example assumes:
    - Your Greenplum Database coordinator hostname is `gpcoord`.
    - The Greenplum Database server is running on the default port.
    - Your Kafka broker host and port is `localhost:9092`.

- You want to write the Kafka data to a Greenplum Database table named `json_from_kafka` located in the `public` schema of a database named `testdb`.
- You want to write the customer identifier and expenses data to Greenplum. The `jsonload_cfg.yaml` file would include the following contents:

```
DATABASE: testdb
USER: gpadmin
HOST: gpcoord
PORT: 5432
KAFKA:
   INPUT:
     SOURCE:
        BROKERS: localhost:9092
        TOPIC: topic_json_gpkafka
     COLUMNS:
        - NAME: jdata
          TYPE: json
     FORMAT: json
     ERROR_LIMIT: 10
   OUTPUT:
     TABLE: json_from_kafka
     MAPPING:
        - NAME: customer_id
          EXPRESSION: (jdata->>'cust_id')::int
        - NAME: month
          EXPRESSION: (jdata->>'month')::int
        - NAME: amount_paid
          EXPRESSION: (jdata->>'expenses')::decimal
   COMMIT:
     MINIMAL_INTERVAL: 2000
```

13. Create the target Greenplum Database table named `json_from_kafka`. For example:

```
gpcoord$ psql -d testdb

testdb=# CREATE TABLE json_from_kafka( customer_id int8, month int4, amount_pai
d decimal(9,2) );
```

14. Exit the `psql` subsystem:

```
testdb=# \q
```

15. Submit the Kafka data load job to the GPSS instance running on port number 5019. (You may consider opening a new terminal window to run the command.) For example to submit a job named `kafkajson2gp`:

```
gpcoord$ gpsscli submit --name kafkajson2gp --gpss-port 5019 ./jsonload_cfg.yam
l
20200804 12:54:19.25262,116652,info,JobID: d577cf37890b5b6bf4e713a9586e86c9,Job
Name: kafkajson2gp
```

16. List all GPSS jobs. For example:

```
gpcoord$ gpsscli list --all --gpss-port 5019
JobName          JobID                              GPHost       GPPort   Data
Base    Schema      Table              Topic               Status
kafkajson2gp     d577cf37890b5b6bf4e713a9586e86c9    localhost    5432     test
db      public      json_from_kafka    topic_json_gpkafka  JOB_SUBMITTED
```

The `list` subcommand displays all jobs. Notice the entry for the `kafkajson2gp` that you just submitted, and that the job is in the *Submitted* state.

17. Start the job named `kafkajson2gp`. For example:

```
gpcoord$ gpsscli start kafkajson2gp --gpss-port 5019
20200804 12:57:57.35153,117918,info,Job kafkajson2gp is started
```

18. Stop the job named `kafkajson2gp`. For example:

```
gpcoord$ gpsscli stop kafkajson2gp --gpss-port 5019
20200804 13:05:09.24280,117506,info,stop job: kafkajson2gp success
```

19. Examine the `gpss` command output and log file, looking for messages that identify the number of rows inserted/rejected. For example:

```
... -[INFO]:- ... Inserted 9 rows
... -[INFO]:- ... Rejected 0 rows
```

20. View the contents of the Greenplum Database target table `json_from_kafka`:

```
gpcoord$ psql -d testdb

testdb=# SELECT * FROM json_from_kafka WHERE customer_id='1313131'
           ORDER BY amount_paid;
 customer_id | month | amount_paid
-------------+-------+-------------
     1313131 |    11 |      368.27
     1313131 |    10 |      492.83
     1313131 |    12 |     1313.13
(3 rows)
```

Merging Data from Kafka into Greenplum Using gpsscli

In this example, you merge data from a Kafka topic named `customer_orders` into a Greenplum Database table named `customer_orders_tbl`. You perform the operation as the Greenplum role `gpadmin`. The table `customer_orders_tbl` resides in the `public` schema in a Greenplum database named `testdb`.

A producer of the Kafka `customer_orders` topic emits customer order messages in CSV format that include the customer identifier (integer) and an order amount (decimal). For example, a message for a customer with identifier 123 who spent $456.78 follows:

```
"123","456.78"
```

You will run a Kafka console producer to emit customer order messages, start a Greenplum Streaming Server instance, and use the GPSS `gpsscli` subcommands to merge and load the data into the

`customer_orders_tbl` Greenplum table. This table has pre-existing data that the merge will overwrite.

Prerequisites

Before you start this procedure, ensure that you:

- Have administrative access to run Kafka and Greenplum Database clusters.
- Have configured connectivity as described in both the Greenplum Streaming Server Prerequisites section and the Kafka Prerequisites.
- Identify and note the ZooKeeper hostname and port.
- Identify and note the hostname and port of the Kafka broker(s).
- Identify and note the hostname and port of the Greenplum Database coordinator node.
- Register the GPSS extension.

This procedure assumes that you have installed the Apache Kafka distribution. If you are using a different Kafka distribution, you may need to adjust certain commands in the procedure.

Procedure

1. Log in to a host in your Kafka cluster. For example:

   ```
   $ ssh kafkauser@kafkahost
   kafkahost$
   ```

2. Create a Kafka topic named `customer_orders`. For example:

   ```
   kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-topics.sh --create \
       --zookeeper localhost:2181 --replication-factor 1 --partitions 1 \
       --topic customer_orders
   ```

3. Open a file named `sample_customer_data.csv` in the editor of your choice. For example:

   ```
   kafkahost$ vi sample_customer_data.csv
   ```

4. Copy/paste the following text to add CSV-format data into the file, and then save and exit:

   ```
   "1313131","1000.00"
   "4444444","99.13"
   "1515151","500.05"
   "6666666","1.12"
   "1717171","3000.03"
   ```

5. Stream the contents of the `sample_customer_data.csv` file to a Kafka console producer. For example:

   ```
   kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-console-producer.sh \
       --broker-list localhost:9092 \
       --topic customer_orders < sample_customer_data.csv
   ```

6. Run the Kafka console consumer to verify that the Kafka console producer published the messages to the topic. For example:

```
kafkahost$ $KAFKA_INSTALL_DIR/bin/kafka-console-consumer.sh \
    --bootstrap-server localhost:9092 --topic customer_orders \
    --from-beginning
```

7. Open a new terminal window, log in to the Greenplum Database coordinator host as the `gpadmin` administrative user, and set up the Greenplum environment. For example:

```
$ ssh gpadmin@gpcoord
gpcoord$ . /usr/local/greenplum-db/greenplum_path.sh
```

8. Construct the the Greenplum Streaming Server configuration file. For example, open a file named `gpsscfg_ex.json` in the editor of your choice:

```
gpcoord$ vi gpsscfg_ex.json
```

9. Designate a GPSS listen port number of 5019 and a `gpfdist` port number of 8319 in the configuration file. For example, copy/paste the following into the `gpsscfg_ex.json` file, and then save and exit the editor:

```
{
    "ListenAddress": {
        "Host": "",
        "Port": 5019
    },
    "Gpfdist": {
        "Host": "",
        "Port": 8319
    }
}
```

10. Start the Greenplum Streaming Server instance in the background, specifying the log directory `./gpsslogs`. For example:

```
gpcoord$ gpss --config gpsscfg_ex.json --log-dir ./gpsslogs &
```

11. Construct the `gpkafka` load configuration file. Open a file named `custorders_cfg.yaml` in the editor of your choice. For example:

```
gpcoord$ vi custorders_cfg.yaml
```

12. Fill in the load configuration parameter values based on your environment. This example assumes:
    - Your Greenplum Database coordinator hostname is `gpcoord`.
    - The Greenplum Database server is running on the default port.
    - Your Kafka broker host and port is `localhost:9092`.
    - You want to write the Kafka data to a Greenplum Database table named `customer_orders_tbl` located in the `public` schema of a database named `testdb`.

- You want to write the customer identifier and order data to Greenplum. If the customer is already present in the table, replace the order amount with the amount read from Kafka. If the customer is not present in the table, add the customer identifier and order amount. You will set merge- and update-related properties in the file to reflect this. The `custorders_cfg.yaml` file would include the following contents:

```
DATABASE: testdb
USER: gpadmin
HOST: gpcoord
PORT: 5432
KAFKA:
   INPUT:
      SOURCE:
        BROKERS: localhost:9092
        TOPIC: customer_orders
      COLUMNS:
        - NAME: id
          TYPE: int
        - NAME: order_amount
          TYPE: decimal(9,2)
      FORMAT: csv
      ERROR_LIMIT: 25
   OUTPUT:
      TABLE: customer_orders_tbl
      MODE: MERGE
      MATCH_COLUMNS:
        - id
      UPDATE_COLUMNS:
        - amount
      MAPPING:
        - NAME: id
          EXPRESSION: id
        - NAME: amount
          EXPRESSION: order_amount
   COMMIT:
      MINIMAL_INTERVAL: 2000
```

13. Start the `psql` subsystem:

```
gpcoord$ psql -d testdb
testdb=#
```

14. Create the target Greenplum Database table named `customer_orders_tbl`, and insert two rows of data in the table. For example:

```
CREATE TABLE customer_orders_tbl( id int8, amount decimal(9,2) );
INSERT INTO customer_orders_tbl VALUES (1717171, 17.17);
INSERT INTO customer_orders_tbl VALUES (1515151, 15.15);
```

15. Exit the `psql` subsystem:

```
testdb=# \q
```

16. Submit the Kafka data load job to the GPSS instance running on port number 5019. (You may consider opening a new terminal window to run the command.) For example to submit a job named

`orders1`:

```
gpcoord$ gpsscli submit --name orders1 --gpss-port 5019 ./custorders_cfg.yaml
20200804 12:54:19.25262,116652,info,JobID: d577cf37890b5b6bf4e713a9586e86c9,Job
Name: orders1
```

17. List all GPSS jobs. For example:

```
gpcoord$ gpsscli list --all --gpss-port 5019
JobName          JobID                            GPHost       GPPort    Data
Base     Schema      Table                   Topic            Status
orders1          d577cf37890b5b6bf4e713a9586e86c9    localhost    5432      test
db       public      customer_orders_tbl     customer_orders  JOB_SUBMITTED
```

    The `list` subcommand displays all jobs. Notice the entry for the `orders1` job that you just submitted, and that the job is in the *Submitted* state.

18. Start the job named `orders1`. For example:

```
gpcoord$ gpsscli start orders1 --gpss-port 5019
20200804 12:57:57.35153,117918,info,Job orders1 is started
```

19. Stop the job named `orders1`. For example:

```
gpcoord$ gpsscli stop orders1 --gpss-port 5019
20200804 13:05:09.24280,117506,info,stop job: orders1 success
```

20. Examine the `gpss` command output and log file, looking for messages that identify the number of rows inserted/rejected. For example:

```
... -[INFO]:- ... Inserted 5 rows
... -[INFO]:- ... Rejected 0 rows
```

21. View the contents of the Greenplum Database target table `customer_orders_tbl`:

```
gpcoord$ psql -d testdb

SELECT * FROM customer_orders_tbl ORDER BY id;
   id    | amount
---------+---------
 1313131 | 1000.00
 1515151 |  500.05
 1717171 | 3000.03
 4444444 |   99.13
 6666666 |    1.12
(5 rows)
```

    Notice that the `amount` value for customers with `id`s `1515151` and `1717171` have been updated to the `total_amount` read from the Kafka message.

Custom Formatter for Kafka

In this example, you create a custom formatter that prepends a text string (provided via an option) to the Kafka data that it receives. All of the formatter code is provided for you. You register the custom formatter with Greenplum Database and use it to process incoming Kafka data.

(Refer to Understanding Custom Formatters for information on developing and using a custom formatter with GPSS.)

The custom formatter example implementation requires that the data start with a four byte header that identifies the length of the text. For example:

```
4 bytes header        content
0x03 0x00 0x00 0x00   ABC
```

To run this example, you must have access to running Kafka and Greenplum Database clusters, and you must have administrative access to Greenplum.

Procedure

Perform the following procedure to register and use a custom formatter in GPSS.

1. Log in to a Greenplum Database host as the `gpadmin` user and set up your Greenplum environment.
2. Create a work directory:

   ```
   gpadmin@gpcoord$ mkdir customfmt_work
   gpadmin@gpcoord$ cd customfmt_work
   ```

3. Open a file named `customfmt.c` in an editor and copy/paste the following custom formatter code into the file:

   ```
   #include "postgres.h"

   #include "access/formatter.h"
   #include "catalog/pg_proc.h"
   #include "fmgr.h"
   #include "funcapi.h"
   #include "utils/builtins.h"
   #include "utils/memutils.h"
   #include "utils/syscache.h"
   #include "utils/typcache.h"

   /* Do the module magic dance */
   PG_MODULE_MAGIC;
   PG_FUNCTION_INFO_V1(customfmt_import);

   Datum customfmt_import(PG_FUNCTION_ARGS);

   typedef struct
   {
           int    ncols;
           Datum* values;
           bool*  nulls;
           int    buflen;
           bytea* buffer;
   ```

```
        /* formatter options */
        /* The prefix string to be added to the data in text column, like
         * prefix='abc_' */
        char* prefix;
        /* When internal_error='1', the query will be stopped immediately. */
        bool internal_error;
        /* When data_exception='1', the query won't be stopped unless it reache
s
         * the error limit. */
        bool data_exception;
        /* When data_exception_once='1', the formatter throw the data exception
once
         * only. Unless it reaches the error limit, the query should continue.
*/
        bool data_exception_once;
} format_t;

/*
 * Our format converts all NULLs to real values, for floats that value is NaN
 */
#define NULL_FLOAT8_VALUE get_float8_nan()

static void
parse_params(FunctionCallInfo fcinfo, format_t* myData)
{
        int nargs = FORMATTER_GET_NUM_ARGS(fcinfo);
        for (int i = 0; i < nargs; i++)
        {
                /* FORMATTER_GET_NTH_ARG_KEY expects index starts from 1 */
                const char* key = FORMATTER_GET_NTH_ARG_KEY(fcinfo, i + 1);
                const char* val = FORMATTER_GET_NTH_ARG_VAL(fcinfo, i + 1);
                if (strcmp(key, "prefix") == 0)
                {
                        myData->prefix = pstrdup(val);
                }
                if (strcmp(key, "internal_error") == 0 && (strcmp(val, "1") ==
0))
                {
                        myData->internal_error = true;
                }
                if (strcmp(key, "data_exception") == 0 && (strcmp(val, "1") ==
0))
                {
                        myData->data_exception = true;
                }
                if (strcmp(key, "data_exception_once") == 0 && (strcmp(val,
"1") == 0))
                {
                        myData->data_exception_once = true;
                }
        }
}

Datum
customfmt_import(PG_FUNCTION_ARGS)
{
        HeapTuple     tuple;
        TupleDesc     tupdesc;
        MemoryContext m, oldcontext;
```

```
        format_t*     myData;
        char*         data_buf;
        int           ncolumns = 0;
        int           data_cur;
        int           data_len;
        int           i;

        /* Must be called via the external table format manager */
        if (!CALLED_AS_FORMATTER(fcinfo))
                elog(ERROR, "customfmt_import: not called by format manager");

        tupdesc = FORMATTER_GET_TUPDESC(fcinfo);

        /* Get our internal description of the formatter */
        ncolumns = tupdesc->natts;
        myData   = (format_t*)FORMATTER_GET_USER_CTX(fcinfo);

        if (myData == NULL)
        {
                myData         = palloc0(sizeof(format_t));
                myData->ncols  = ncolumns;
                myData->values = palloc(sizeof(Datum) * ncolumns);
                myData->nulls  = palloc(sizeof(bool) * ncolumns);

                /* parse parameters */
                parse_params(fcinfo, myData);

                /* misc verification */
                for (i = 0; i < ncolumns; i++)
                {
                        Oid type = tupdesc->attrs[i]->atttypid;
                        // int32 typmod = TupleDescAttr(tupdesc, i)->atttypmod;

                        /* Don't know how to format dropped columns, error for
now */
                        if (tupdesc->attrs[i]->attisdropped)
                                ereport(ERROR, (errcode(ERRCODE_INTERNAL_ERRO
R),
                                                errmsg("customfmt_import: dropp
ed columns")));

                        switch (type)
                        {
                                case FLOAT8OID:
                                case VARCHAROID:
                                case BPCHAROID:
                                case TEXTOID:
                                        break;

                                default: {
                                        ereport(ERROR, (errcode(ERRCODE_INTERNA
L_ERROR),
                                                        errmsg("customfmt_impor
t error: "
                                                               "unsupported dat
a type")));
                                        break;
                                }
                        }
```

```
                }

                FORMATTER_SET_USER_CTX(fcinfo, myData);
        }
        if (myData->ncols != ncolumns)
                ereport(ERROR, (errcode(ERRCODE_INTERNAL_ERROR),
                                errmsg("customfmt_import: unexpected change of
output "
                                       "record type")));

        /* get our input data buf and number of valid bytes in it */
        data_buf = FORMATTER_GET_DATABUF(fcinfo);
        data_len = FORMATTER_GET_DATALEN(fcinfo);
        data_cur = FORMATTER_GET_DATACURSOR(fcinfo);

        /* start clean */
        MemSet(myData->values, 0, ncolumns * sizeof(Datum));
        MemSet(myData->nulls, true, ncolumns * sizeof(bool));

        /* ================================================================
===
         *                              MAIN FORMATTING CODE
         *
         * Currently this code assumes:
         *  - Homogoneos hardware => No need to convert data to network byte or
der
         *  - Support for TEXT/VARCHAR/BPCHAR/FLOAT8 only
         *  - Length Prefixed strings
         *  - No end of record tags, checksums, or optimizations for alignment.
         *  - NULL values are cast to some sensible default value (NaN, "")
         *
         * ================================================================
===
         */
        m          = FORMATTER_GET_PER_ROW_MEM_CTX(fcinfo);
        oldcontext = MemoryContextSwitchTo(m);

        if (myData->internal_error)
        {
                /* Reporting an internal error will stop query immediately. NOT
HING will
                 * be saved into the error log.*/
                MemoryContextSwitchTo(oldcontext);
            FORMATTER_SET_BAD_ROW_DATA(fcinfo, data_buf, data_len);
            FORMATTER_SET_BYTE_NUMBER(fcinfo, data_len);
                ereport(ERROR,
                        (errcode(ERRCODE_INTERNAL_ERROR),
                         errmsg("reports error in example. data_len: %d, data_c
ur: %d",
                                data_len, data_cur)));
        }
        if (myData->data_exception)
        {
                MemoryContextSwitchTo(oldcontext);
            FORMATTER_SET_BAD_ROW_DATA(fcinfo, data_buf, data_len);
            FORMATTER_SET_BYTE_NUMBER(fcinfo, data_len);
                ereport(ERROR,
                        (errcode(ERRCODE_DATA_EXCEPTION),
                         errmsg("data exception in example. data_len: %d, data_
```

```
cur: %d",
                                data_len, data_cur)));
        }
        if (myData->data_exception_once)
        {
                int32 len;
                memcpy(&len, data_buf + data_cur, sizeof(len));
                myData->data_exception_once = false;
                MemoryContextSwitchTo(oldcontext);
            FORMATTER_SET_BAD_ROW_DATA(fcinfo, data_buf, data_len);
            FORMATTER_SET_BYTE_NUMBER(fcinfo, data_len);
                ereport(ERROR,
                        (errcode(ERRCODE_DATA_EXCEPTION),
                         errmsg("data exception in example. data_len: %d, data_
cur: %d",
                                data_len, data_cur)));
        }

        for (i = 0; i < ncolumns; i++)
        {
                Oid type      = tupdesc->attrs[i]->atttypid;
                int remaining = 0;
                int attr_len  = 0;

                remaining = data_len - data_cur;

                switch (type)
                {
                        case FLOAT8OID: {
                                float8 value;

                                attr_len = sizeof(value);

                                if (remaining < attr_len)
                                {
                                        MemoryContextSwitchTo(oldcontext);
                                        ereport(ERROR, (errcode(ERRCODE_DATA_EX
CEPTION),
                                                        errmsg("incomplete dat
a")));
                                }

                                memcpy(&value, data_buf + data_cur, attr_len);

                                if (value != NULL_FLOAT8_VALUE)
                                {
                                        myData->nulls[i]  = false;
                                        myData->values[i] = Float8GetDatum(valu
e);
                                }

                                /* TODO: check for nan? */

                                break;
                        }

                        case TEXTOID:
                        case VARCHAROID:
                        case BPCHAROID: {
```

```
                                text* value;
                                int32 len;
                                bool  nextlen = remaining >= sizeof(len);

                                if (nextlen)
                                {
                                        memcpy(&len, data_buf + data_cur, sizeo
f(len));

                                        if (len < 0)
                                        {
                                                MemoryContextSwitchTo(oldcontex
t);
                                                ereport(ERROR,
                                                        (errcode(ERRCODE_DATA_E
XCEPTION),
                                                         errmsg("invalid length
of varlen datatype: %d",
                                                                len)));
                                        }
                                }

                                /* if len or data bytes don't exist in this buf
fer, return */
                                if (!nextlen || (nextlen && (remaining - sizeof
(len) < len)))
                                {
                                        MemoryContextSwitchTo(oldcontext);
                                        /* gpss extension handled the data inte
grity already. This
                                         * should not happen.*/
                                        ereport(ERROR, (errcode(ERRCODE_DATA_EX
CEPTION),
                                                        errmsg("incomplete dat
a")));
                                }

                                if (len > 0)
                                {
                                        int prefixlen = 0;
                                        /* Add the prefix if it has been set in
the formatter
                                         * options. */
                                        if (myData->prefix)
                                        {
                                                prefixlen = strlen(myData->pref
ix);
                                        }
                                        value = (text*)palloc(len + prefixlen +
VARHDRSZ);
                                        SET_VARSIZE(value, len + prefixlen + VA
RHDRSZ);

                                        memcpy(VARDATA(value), myData->prefix,
prefixlen);
                                        memcpy(VARDATA(value) + prefixlen,
                                               data_buf + data_cur + sizeof(le
n), len);
```

```
                                        myData->nulls[i]  = false;
                                        myData->values[i] = PointerGetDatum(val
ue);
                                }

                                attr_len = len + sizeof(len);

                                break;
                        }

                        default:
                                MemoryContextSwitchTo(oldcontext);
                                ereport(ERROR, (errcode(ERRCODE_INTERNAL_ERRO
R),
                                                errmsg("customfmt_import: unsup
ported "
                                                       "datatype, id %d:",
                                                       type)));
                                break;
                }

                /* add byte length of last attribute to the temporary cursor */
                data_cur += attr_len;
        }
        /* ============================================================================
===
         */

        MemoryContextSwitchTo(oldcontext);
    FORMATTER_SET_DATACURSOR(fcinfo, data_cur);

        tuple = heap_form_tuple(tupdesc, myData->values, myData->nulls);

        /* hack... pass tuple here. don't free prev tuple - the executor does i
t  */
        ((FormatterData*)fcinfo->context)->fmt_tuple = tuple;

        FORMATTER_RETURN_TUPLE(tuple);
}
```

4. Save the file and exit the editor.
5. Open a file named `Makefile` in an editor and copy/paste the following directives into the file:

```
MODULE_big = customfmt_example
OBJS = customfmt.o
PG_CPPFLAGS = -I$(shell $(PG_CONFIG) --includedir)
SHLIB_LINK = -L$(shell $(PG_CONFIG) --libdir)

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

6. Save the file and exit the editor.
7. Generate the custom formatter function definition. Open a file named `customfmt_example.sql` in an editor and copy/paste the following `CREATE FUNCTION` call into the file:

```
CREATE OR REPLACE FUNCTION customfmt_in() RETURNS record
AS '$libdir/customfmt_example.so', 'customfmt_import'
LANGUAGE C STABLE;
```

8. Save the file and exit the editor.
9. Copy the file to your Greenplum Database installation; you must have administrative privileges to copy the file:

```
gpadmin@gpcoord$ cp customfmt_example.sql /usr/local/greenplum-db/lib/postgresq
l/
```

10. Build the custom formatter shared library:

```
gpadmin@gpcoord$ make
```

The `make` command generates a shared library named `customfmt_example.so` in the current directory.

11. Copy the shared library to your Greenplum Database installation; you must have administrative privileges to copy the file:

```
gpadmin@gpcoord$ cp customfmt_example.so /usr/local/greenplum-db/lib/postgresq
l/
```

12. Create a test database:

```
gpadmin@gpcoord$ createdb testdb
```

13. Register the custom formatter function in this database:

```
gpadmin@gpcoord$ psql -d testdb -U gpadmin -f $GPHOME/share/postgresql/customfm
t_example.sql
```

14. Create a Greenplum table in the database:

```
gpadmin@gpcoord$ psql -d testdb -U gpadmin -c 'CREATE TABLE test_table( str_col
umn text );'
```

15. Create a Kafka topic named `customtest`.
16. Start a GPSS server:

```
gpadmin@gpcoord$ gpss &
```

17. Create a version 2 Kafka load configuration file; copy/paste the following into a file named `kafka_custom_formatter.yml`:

```
DATABASE: testdb
USER: gpadmin
HOST: localhost
PORT: 15432
VERSION: 2
KAFKA:
```

```
   INPUT:
      SOURCE:
        BROKERS: localhost:9092
        TOPIC: test
      VALUE:
        COLUMNS:
        - NAME: value
          TYPE: text
        FORMAT: custom
        CUSTOM_OPTION:
          NAME: customfmt_in
          PARAMSTR: prefix="kafka_msg_"
      ERROR_LIMIT: 2
   OUTPUT:
      TABLE: test_table
      MODE: INSERT
      MAPPING:
        - NAME: str_column
          EXPRESSION: value
```

18. Submit the job:

```
gpadmin@gpcoord$ gpsscli submit kafka_custom_formatter.yml
```

19. Start the job:

```
gpadmin@gpcoord$ gpsscli start kafka_custom_formatter
```

20. Generate a binary test data record and save to a file named `data_example.bin`.

```
gpadmin@gpcoord$ cat "0x03 0x00 0x00 0x00 0x41 0x42 0x43" > input.txt
gpadmin@gpcoord$ xxd -r -p input.txt data_example.bin
```

21. Load the test data into Kafka:

```
gpadmin@gpcoord$ cat data_example.bin | kafka-console-producer --broker-list lo
calhost:9292 --topic customtest
```

22. Examine the Greenplum `test_table` table. `psql -d testdb -U gpadmin -c 'SELECT * FROM test_table;'`

Best Practices

This topic presents best practices to follow when you use the Greenplum Streaming Server Kafka Integration.

Choosing a Commit Threshold

GPSS supports two mechanisms to control how and when it commits Kafka data to Greenplum Database: a time period or a number of rows. You specify one or both of `MINIMAL_INTERVAL` or `MAX_ROW` in the Kafka load configuration file.

For best results, try various settings of `MINIMAL_INTERVAL` to determine what value works best in your environment.

When message flow is heavy, GPSS may receive and buffer many messages during the `MINIMAL_INTERVAL` time period. In this situation, also providing a `MAX_ROW` setting may mitigate any high memory usage scenarios.

Loading File Data into Greenplum

You can use the gpsscli utility to load data from a file or from the `stdout` of a command into Greenplum Database.

Load Procedure

You will perform the following tasks when you use the Greenplum Streaming Server to load file or command output data into a Greenplum Database table:

1. Ensure that you meet the Prerequisites.
2. Register the Greenplum Streaming Server extension.
3. Identify the format of the data.
4. Construct the load configuration file.
5. Create the target Greenplum Database table.
6. Assign Greenplum Database role permissions to the table, if required.
7. Run the gpsscli Client Commands to load the data into Greenplum Database.
8. Check for load errors.

Prerequisites

Before using the `gpsscli` utilities to load file or command output data to Greenplum Database, ensure that:

- Your systems meet the Prerequisites documented for the Greenplum Streaming Server.
- The file or command is accessible on the ETL server host, and the operating system user running the `gpss` server process has the appropriate permissions to access the file or run the command.

About Supported Data Formats

To write file or command output data into a Greenplum Database table, you must identify the format of the data in the load configuration file.

The Greenplum Streaming Server supports loading files of the following formats:

Format	Description
avro	Avro-format data. Specify the `avro` format when you want to load a single-object encoded Avro file. GPSS reads Avro data from the file and loads it into a single JSON-type column. You must define a mapping if you want GPSS to write the data into specific columns in the target Greenplum Database table. GPSS supports `libz`-, `lzma`-, and `snappy`-compressed Avro data.

Format	Description
binary	Binary format data. Specify the `binary` format when your file is binary format. GPSS reads binary data from the file and loads it into a single bytea-type column.
csv	Comma-delimited text format data. Specify the `csv` format when your file data is comma-delimited text and conforms to RFC 4180. The file may not contain line ending characters (CR and LF).
custom	Data of a custom format, parsed by a custom formatter function.
delimited	Text data separated by a configurable delimiter. The `delimited` data format supports a multi-byte delimiter.
json, jsonl (version 2 only)	JSON- or JSONB-format data. Specify the `json` format when the file is in JSON or JSONB format. GPSS can read JSON data as a single object or can read a single JSON record per line. You must define a mapping if you want GPSS to write the data into specific columns in the target Greenplum Database table. **Note:** GPSS supports JSONB-format data only when loading to Greenplum 6. **Note:** Specify `FORMAT: jsonl` in version 2 format load configuration files. Specify `json` with `is_jsonl: true` in version 3 (Beta) format load configuration files.

Constructing the filesource.yaml Configuration File

You configure a data load operation from a file or command output to Greenplum Database via a YAML-formatted configuration file. This configuration file includes parameters that identify the source file or command and information about the Greenplum Database connection and target table, as well as error thresholds for the operation.

The Greenplum Streaming Server supports versions 2 and 3 (Beta) of the YAML configuration file when you load data into Greenplum from a file or command output. Versions 2 and 3 of the configuration file format support loading file/command and meta data to Greenplum.

Refer to the filesource-v2.yaml reference page for configuration file format and the configuration properties supported.

A sample version 2 file load YAML configuration file named `loadfromfile2.yaml` follows:

```
DATABASE: ops
USER: gpadmin
PASSWORD: changeme
HOST: mdw-1
PORT: 5432
VERSION: 2
FILE:
   INPUT:
      SOURCE:
         URL: file:///tmp/file.csv
      VALUE:
         COLUMNS:
           - NAME: id
             TYPE: int
           - NAME: cname
             TYPE: text
           - NAME: oname
             TYPE: text
         FORMAT: delimited
         DELIMITED_OPTION:
```

```
            DELIMITER: "\t"
            EOL_PREFIX: "$$EOL$$"
            QUOTE: '&'
            ESCAPE: '|'
      META:
         COLUMNS:
           - NAME: meta
             TYPE: json
         FORMAT: json
      ERROR_LIMIT: 25
   OUTPUT:
      SCHEMA: gpschema
      TABLE: gptable
      MODE: INSERT
      MAPPING:
        - NAME: id
          EXPRESSION: id
        - NAME: cname
          EXPRESSION: cname
        - NAME: fname
          EXPRESSION: (meta->>'filename')::text
SCHEDULE:
  RETRY_INTERVAL: 500ms
  MAX_RETRIES: 2
```

Greenplum Database Options (Version 2-Focused)

You identify the Greenplum Database connection options via the `DATABASE`, `USER`, `PASSWORD`, `HOST`, and `PORT` parameters.

The `VERSION` parameter identifies the version of the GPSS YAML configuration file.

You must specify version 2 when you load from a file or command output data source into Greenplum using this format.

Input Options

You can direct GPSS to load from a file or from the `stdout` of a command:

- Specify a file location using the `SOURCE:URL` property. GPSS supports wildcards in the file path. If you want to read all files in a directory, specify `dirname/*`.
- Alternatively, you can load the `stdout` of a command by specifying the command and execution options using the properties in the the `SOURCE:EXEC` block.

The default source-to-target data mapping behaviour of GPSS is to match a column name as defined in `VALUE:COLUMNS:NAME` with a column name in the target Greenplum Database `OUTPUT:TABLE`:

- You must identify the data elements in the order in which they appear in the file or command output.
- You must provide the same name for each data element and its associated Greenplum Database table column.

- You must specify a compatible data type for each data element and its associated Greenplum Database table column.

The `VALUE` block must specify a `FORMAT`. The `VALUE:FORMAT` keyword identifies the format of the file. GPSS supports comma-delimited text (`csv`), binary (`binary`), or JSON/JSONB (`json`), and Avro (`avro`) format files. GPSS also supports data that is separated by a configurable multi-byte delimiter (`delimited`).

When you provide a `META` block, you must specify a single JSON-type `COLUMNS` and the `FORMAT: json`. Meta data for a file is a single `text` property named `filename`. GPSS does not support meta data when loading from command output.

The `FILTER` parameter identifies a filter to apply to the data before it is loaded into Greenplum Database. If the filter evaluates to `true`, GPSS loads the data. The data is dropped if the filter evaluates to `false`. The filter string must be a valid SQL conditional expression and may reference one or more `VALUE` column names.

The `ERROR_LIMIT` parameter identifies the number of errors or the error percentage threshold after which GPSS should exit the load operation.

FILE:OUTPUT Options

You identify the target Greenplum Database schema name and table name via the `FILE:OUTPUT: SCHEMA` and `TABLE` parameters. *You must pre-create the Greenplum Database table before you attempt to load file or command output data.*

The default load mode is to insert data into the Greenplum Database table. GPSS also supports updating and merging data into a Greenplum table. You specify the load `MODE`, the `MATCH_COLUMNS` and `UPDATE_COLUMNS`, and any `UPDATE_CONDITION`s that must be met to merge or update the data. In `MERGE MODE`, you can also specify `ORDER_COLUMNS` to filter out duplicates and a `DELETE_CONDITION`.

You can override the default mapping of the `INPUT:VALUE:COLUMNS` by specifying a `MAPPING` block in which you identify the association between a specific column in the target Greenplum Database table and a data value element. You can also map the `META` data column, and map a Greenplum Database table column to a value expression.

When you specify a `MAPPING` block, ensure that you provide entries for all data elements of interest - GPSS does not automatically match column names when you provide a `MAPPING`.

About the Merge Load Mode

`MERGE` mode is similar to an `UPSERT` operation; GPSS may insert new rows in the database, or may update an existing database row that satisfies match and update conditions. GPSS deletes rows in `MERGE` mode when the data satisfies an optional `DELETE_CONDITION` that you specify.

GPSS stages a merge operation in a temporary table, generating the SQL to populate the temp table from the set of `OUTPUT` configuration properties that you provide.

GPSS uses the following algorithm for `MERGE` mode processing:

1. Create a temporary table like the target table.
2. Generate the SQL to insert the source data into the temporary table.
    1. Add the `MAPPINGS`.
    2. Add the `FILTER`.
    3. Use `MATCH_COLUMNS` and `ORDER_COLUMNS` to filter out duplicates.
3. Update the target table from rows in the temporary table that satisfy `MATCH_COLUMNS`, `UPDATE_COLUMNS`, and `UPDATE_CONDITION`.
4. Insert non-matching rows into the target table.
5. Delete rows in the target table that satisfy `MATCH_COLUMNS` and the `DELETE_CONDITION`.
6. Truncate the temporary table.

About the JSON Format and Column Type

When you specify `FORMAT: json` or `FORMAT: jsonl`, valid `COLUMN:TYPE`s for the data include `json` or `jsonb`. You can also specify the new GPSS `gp_jsonb` (Beta) or `gp_json` (Beta) column types.

- `gp_jsonb` is an enhanced JSONB type that adds support for `\u` escape sequences and unicode. For example, `gp_jsonb` can escape `\uDD8B` and `\u0000` as text format, but `jsonb` treats these characters as illegal.
- `gp_json` is an enhanced JSON type that can tolerate certain illegal unicode sequences. For example, `gp_json` automatically escapes incorrect surrogate pairs and processes `\u0000` as `\\u0000`. Note that unicode escape values cannot be used for code point values above `007F` when the server encoding is not `UTF8`.

You can use the `gp_jsonb` (Beta) and `gp_json` (Beta) data types as follows:

- As the `COLUMN:TYPE` when the target Greenplum Database table column type is `json` or `jsonb`.
- In a `MAPPING` when the target Greenplum Database column is `text` or `varchar`. For example:

    ```
    EXPRESSION: (j->>'a')::text
    ```

- In a `MAPPING` when `FORMAT: avro` and the target Greenplum Database column is `json` or `jsonb`. For example:

    ```
    EXPRESSION: j::gp_jsonb
    ```

    or

    ```
    EXPRESSION: j::gp_json
    ```

- In a `MAPPING` when `FORMAT: avro` and the target Greenplum Database column is `text` or `varchar`. For example:

    ```
    EXPRESSION: (j::gp_jsonb->>'a')::text
    ```

    or

```
EXPRESSION: (j::gp_json->>'a')::text
```

The `gp_jsonb` (Beta) and `gp_json` (Beta) data types are defined in an extension named `dataflow`. You must `CREATE EXTENSION dataflow;` in each database in which you choose to use these (Beta) data types.

Preserving Ill-Formed JSON Escape Sequences

GPSS exposes a configuration parameter that you can use with the `gp_jsonb` and `gp_json` types. The name of this parameter is `gpss.json_preserve_ill_formed_prefix`. When set, GPSS does not return an error when it encounters an ill-formed JSON escape sequence with these types, but instead prepends it with the prefix that you specify.

For example, if `gpss.json_preserve_ill_formed_prefix` is set to the string `"##"` as follows:

```
SET gpss.json_preserve_ill_formed_prefix = "##";
```

and GPSS encounters an ill-formed JSON sequence such as the orphaned low surrogate `\ude04X`, GPSS writes the data as `##\ude04X` instead.

About META, VALUEs, and FORMATs

You can specify the `avro`, `binary`, `csv`, `delimited`, or `json` data format in the Version 2 configuration file `INPUT:VALUE:FORMAT`, with some restrictions. You cannot specify a `META` block when `INPUT:VALUE:FORMAT` is `csv`.

About Transforming and Mapping Input Data

You can define a `MAPPING` between the input data (`VALUE:COLUMNS` and `META:COLUMNS`) and the columns in the target Greenplum Database table. Defining a mapping may be useful when you have a multi-field input column (such as a JSON-type column), and you want to assign individual components of the input field to specific columns in the target table.

You might also use a `MAPPING` to assign a value expression to a target table column. The expression must be one that you could specify in the `SELECT` list of a query, and can include a constant value, a column reference, an operator invocation, a built-in or user-defined function call, and so forth.

If you choose to map more than one input column in an expression, you can can create a user-defined function to parse and transform the input column and return the columns of interest.

For example, suppose you are loading a JSON file with the following contents:

```
{ "customer_id": 1313131, "some_intfield": 12 }
{ "customer_id": 77, "some_intfield": 7 }
{ "customer_id": 1234, "some_intfield": 56 }
```

You could define a user-defined function, `udf_parse_json()`, to parse the data as follows:

```
=> CREATE OR REPLACE FUNCTION udf_parse_json(value json)
     RETURNS TABLE (x int, y text)
```

```
  LANGUAGE plpgsql AS $$
    BEGIN
       RETURN query
       SELECT ((value->>'customer_id')::int), ((value->>'some_intfield')::text);
    END $$;
```

This function returns the two fields in each JSON record, casting the fields to integer and text, respectively.

An example `MAPPING` for file data in a JSON-format `FILE:INPUT:COLUMNS` named `jdata` follows:

```
MAPPING:
  cust_id: (jdata->>'customer_id')
  field2: ((jdata->>'some_intfield') * .075)::decimal
  j1, j2: (udf_parse_json(jdata)).*
```

The Greenplum Database table definition for this example scenario is:

```
=> CREATE TABLE t1map( cust_id int, field2 decimal(7,2), j1 int, j2 text );
```

Creating the Greenplum Table

You must pre-create the Greenplum table before you load data into Greenplum Database. You use the `FILE:OUTPUT: SCHEMA` and `TABLE` load configuration file parameters to identify the schema and table names.

The target Greenplum table definition must include each column that GPSS will load into the table. The table definition may include additional columns; GPSS ignores these columns, and loads no data into them.

The name and data type that you specify for a column of the target Greenplum Database table must match the name and data type of the related data element. If you have defined a column mapping, the name of the Greenplum Database column must match the target column name that you specified for the mapping, and the type must match the target column type or expression that you define.

The `CREATE TABLE` command for the target Greenplum Database table receiving the data defined in the `loadfromfile2.yaml` file presented in the Constructing the filesource.yaml Configuration File section follows:

```
testdb=# CREATE TABLE payables.expenses2( id int8, cname text, fname text );
```

Loading from S3 into Greenplum (Beta)

You can use the gpsscli utility to load data from S3 into VMware Tanzu Greenplum. The GPSS `s3` data source uses the Tanzu Greenplum s3 Protocol to read data into an `s3` external table and write it to a Greenplum table. Tanzu Greenplum segment instances read the data from S3 in parallel, and similarly write the data to Greenplum in parallel. GPSS is not involved in the data transfer.

The GPSS s3 data source does not read directly from S3.

The GPSS s3 data source can load text and CSV files residing on S3. The data source also supports loading gzip-compressed versions of these files.

Load Procedure

You will perform the following tasks when you use the Greenplum Streaming Server to load S3 data into a Greenplum Database table:

1. Ensure that you meet the Prerequisites.
2. Register the Greenplum Streaming Server extension.
3. Identify the format of the data.
4. Construct the load configuration file.
5. Create the target Greenplum Database table.
6. Assign Greenplum Database role permissions to the table, if required.
7. Run the gpsscli Client Commands to load the data into Greenplum Database.
8. Check for load errors.

Prerequisites

Before using the `gpsscli` utilities to load S3 data to Greenplum Database, ensure that:

- Your systems meet the Prerequisites documented for the Tanzu Greenplum Streaming Server.
- You have configured the `s3` protocol as described in the VMware Tanzu Greenplum s3 Protocol documentation.
- You can identify the URI of the S3 file that you want to load.
- You can identify an S3 access ID and secret key that have the permissions required to access the file.

About Supported File Formats

To write data from S3 into a Greenplum Database table, you must identify the format of the file in the load configuration file.

The Greenplum Streaming Server s3 data source supports loading files of the following formats:

Format	Description
csv	Comma-delimited text format data. Specify the `csv` format when your file data is comma-delimited text and conforms to RFC 4180. The file may not contain line ending characters (CR and LF).
text	Plain text format. Specify the `csv` format and an empty `delimiter` when you want to read a plain text file from S3.
gzipped csv and text files	Gzip-compressed file. Specify the `csv` format and the `delimiter` (the gzipped file is a csv file) or an empty `delimiter` (the gzipped file is a plain text file) when you want to read a `.gz` file from S3.

Constructing the s3source.yaml Configuration File

You configure a data load operation from a file to Greenplum Database via a YAML-formatted configuration file. This configuration file includes parameters that identify the source file and information about the Greenplum Database connection and target table, as well as error thresholds for the operation.

The Greenplum Streaming Server supports version 3 (Beta) of the YAML configuration file when you load data into Greenplum from S3. Refer to the s3source-v3.yaml reference page for the configuration file format and the configuration properties supported.

A sample version 3 s3 load YAML configuration file named `loadfroms3.yaml` follows:

```
version: v3
targets:
  - gpdb:
      host: localhost
      port: 6000
      user: bill
      password: changeme
      database: testdb
      work_schema: public
      error_limit: "25"
      filter_expression: "test_filter"
      tables:
        - table: s3_target
          schema: public
          mode:
            insert: {}
sources:
  - s3:
      uri:
        - "s3://s3-us-east-1.amazonaws.com/mydir/mybucket/data0000"
      content:
        csv:
          columns:
            - name: c1
              type: text
            - name: c2
              type: int
          delimiter: ""
```

```
    s3param:
      version: 1
      accessid: 123
      secret: 456
      chunksize: 4096
      threadnum: 4
      gpcheckcloud_newline: "\n"
      autocompress: false
      encryption: false
      verifycert: false
      low_speed_limit: 1
```

Creating the Greenplum Table

You must pre-create the Greenplum table before you load S3 data into Greenplum Database. You use the `table:schema` and `tables:table` load configuration file properties to identify the schema and table names.

The target Greenplum table definition must include each column that GPSS will load into the table. The table definition may include additional columns; GPSS ignores these columns, and loads no data into them.

The name and data type that you specify for a column of the target Greenplum Database table must match the name and data type of the related, S3 file data element. If you have defined a column mapping, the name of the Greenplum Database column must match the target column name that you specified for the mapping, and the type must match the target column type or expression that you define.

A `CREATE TABLE` command for the target Greenplum Database table receiving the file data defined in the `loadfroms3.yaml` file presented in the Constructing the s3source.yaml Configuration File section follows:

```
testdb=# CREATE TABLE s3_target(c1 text, c2 int);
```

Loading RabbitMQ Data into Greenplum

The GPSS RabbitMQ data source loads data from a RabbitMQ queue (the traditional AMQP implementation) or a RabbitMQ stream (persistent and replicated structure available in RabbitMQ version 3.9 and later) into Greenplum Database.

You can use the gpsscli utility to load RabbitMQ data/messages into Greenplum Database. The GPSS server, `gpss`, is a RabbitMQ consumer. It ingests streaming data from a single RabbitMQ queue or stream using Greenplum Database readable external tables to transform and insert or update the data into a target Greenplum table. You identify the RabbitMQ server, queue or stream name, virtual host, data format, and the Greenplum connection options and target table definition in a YAML-formatted load configuration file that you provide to the utility.

Load Procedure

You will perform the following tasks when you use the Greenplum Streaming Server to load RabbitMQ message data into a Greenplum Database table:

1. Ensure that you meet the Prerequisites.
2. Register the Greenplum Streaming Server extension.
3. Identify the format of the RabbitMQ data.
4. Construct the load configuration file.
5. Create the target Greenplum Database table.
6. Assign Greenplum Database role permissions to the table, if required.
7. Run the gpsscli Client Commands to load the data into Greenplum Database.
8. Check for load errors.

Prerequisites

Before using the `gpsscli` utility to load RabbitMQ data to Greenplum Database, ensure that you:

- Meet the Prerequisites documented for the Greenplum Streaming Server, and configure and start the server.
- Have access to a running RabbitMQ cluster, and that you can identify the hostname and port number of the RabbitMQ server serving the data.
- Can identify the name of the RabbitMQ queue or stream of interest.
- Can run the command on a host that has connectivity to:
    - Each RabbitMQ host in the RabbitMQ cluster.
    - The Greenplum Database coordinator and all segment hosts.

About Supported Message Data Formats

The Greenplum Streaming Server supports RabbitMQ message value data in the following formats:

Format	Description
binary	Binary format data. GPSS reads binary data from RabbitMQ only as a single bytea-type column.
csv	Comma-delimited text format data.
custom	Data of a custom format, parsed by a custom formatter function.
delimited	Text data separated by a configurable delimiter.
json, jsonl (version 2 only)	JSON- or JSONB-format data. Specify the `json` format when the file is in JSON or JSONB format. GPSS can read JSON data as a single object or can read a single JSON record per line. You must define a mapping if you want GPSS to write the data into specific columns in the target Greenplum Database table.  Note: GPSS supports JSONB-format data only when loading to Greenplum 6.  Note: Specify `FORMAT: jsonl` in version 2 format load configuration files. Specify `json` with `is_jsonl: true` in version 3 (Beta) format load configuration files.

To write RabbitMQ message data into a Greenplum Database table, you must identify the data format in the load configuration file.

Binary

Use the `binary` format when your RabbitMQ message data is a stream of bytes. GPSS reads binary data from RabbitMQ and loads it into a single bytea-type column.

CSV

Use the `csv` format when your RabbitMQ message data is comma-delimited text and conforms to RFC 4180. The message content may not contain line ending characters (CR and LF).

Data in `csv` format may appear in RabbitMQ messages as follows:

```
"1313131","12","backorder","1313.13"
"3535353","11","shipped","761.35"
"7979797","11","partial","18.72"
```

Custom

The Greenplum Streaming Server provides a custom data formatter plug-in framework for RabbitMQ messages using user-defined functions. The type of RabbitMQ message data processed by a custom formatter is formatter-specific. For example, a custom formatter may process compressed or complex data.

Delimited Text

The Greenplum Streaming Server supports loading RabbitMQ data delimited by one or more characters that you specify. Use the `delimited` format for such data. The delimiter may be a multi-byte value and up to 32

bytes in length.You can also specify quote and escape characters, and an end-of-line prefix.

The delimiter may not contain the quote or escape characters.

When you specify a quote character:

- The left and right quotes are the same.
- Each data element must be quoted. GPSS does not support mixed quoted and unquoted content.
- You must also define an escape character.
- GPSS keeps the original format of any character between the quotes, except the quote and escape characters. This especially applies to the delimiter and `\n`, which do not require additional escape if they are quoted.
- The quote character is presented as the escape character plus the quote character (for example, \").
- The escape character is presented as the escape character plus the escape character (for example, \)
- GPSS parses multiple escape characters from left to right.

When you do not specify a quote character:

- The escape character is optional.
- If you do not specify an escape character, GPSS treats the delimiter as the column separator, and treats any end-of-line prefix plus `\n` as the row separator.
- If you do specify an escape character:
    - GPSS uses the escape character plus the delimiter as the column separator.
    - GPSS uses the escape character plus the end-of-line prefix plus `\n` as the row separator.
    - The escape character plus the escape character is the escape character itself.
    - GPSS parses multiple escape characters from left to right.

Sample data using a pipe ('|') delimiter character follows:

```
1313131|12|backorder|1313.13
3535353|11|shipped|761.35
7979797|11|partial|18.72
```

JSON (single object)

Specify the `json` format when your RabbitMQ message data is in JSON or JSONB format and you want GPSS to read JSON data from RabbiMQ as a single object into a single column (per the JSON specification, newlines and white space are ignored). You must define a mapping if you want GPSS to write the data into specific columns in the target Greenplum Database table.

GPSS supports JSONB-format data only when loading to Greenplum 6.

JSON (single record per line)

Specify `FORMAT: jsonl` in version 2 format load configuration files or specify `json` with `is_jsonl: true` in version 3 (Beta) format load configuration files when your RabbitMQ message data is in JSON format, single JSON record per line. You must define a mapping if you want GPSS to write the data into specific columns in the target Greenplum Database table.

Sample JSON message data:

```
{ "cust_id": 1313131, "month": 12, "amount_paid":1313.13 }
{ "cust_id": 3535353, "month": 11, "amount_paid":761.35 }
{ "cust_id": 7979797, "month": 11, "amount_paid":18.82 }
```

Registering a Custom Formatter

A custom data formatter for RabbitMQ messages is a user-defined function. If you are using a custom formatter, you must create the formatter function and register it in each database in which you will use the function to write RabbitMQ data to Greenplum tables.

Constructing the rabbitmq.yaml Configuration File

You configure a data load operation from RabbitMQ to Greenplum Database via a YAML-formatted configuration file. This configuration file includes parameters that identify the source RabbitMQ data and information about the Greenplum Database connection and target table, as well as error and commit thresholds for the operation.

When loading from RabbitMQ, the Greenplum Streaming Server supports two versions of the YAML configuration file: version 2 and version 3 (Beta).

Refer to the rabbitmq-v2.yaml reference page for Version 2 configuration file format and the configuration parameters that this version supports. rabbitmq-v3.yaml describes the Version 3 (Beta) format.

Contents of a sample `gpsscli` Version 2 YAML configuration file named `loadcfgrmq2.yaml` follows:

```
DATABASE: testdb
USER: gpadmin
PASSWORD: password
HOST: localhost
PORT: 15432
VERSION: 2
RABBITMQ:
  INPUT:
    SOURCE:
      SERVER: gpdmin:changeme@localhost:5552
      STREAM: test_stream
      VIRTUALHOST: vhost_for_gpss
    DATA:
      COLUMNS:
        - NAME: c1
          TYPE: int
```

```
      - NAME: c2
        TYPE: int
      - NAME: path
        TYPE: text
    FORMAT: CSV
OUTPUT:
  SCHEMA: "public"
  TABLE: tbl_int_text_column
  MODE: MERGE
  MATCH_COLUMNS:
    - c1
  UPDATE_COLUMNS:
    - c2
  ORDER_COLUMNS:
    - path
  UPDATE_CONDITION: c2 = 11
  DELETE_CONDITION: c1 = 0
  MAPPING:
    - NAME: c1
      EXPRESSION: c1::int
    - NAME: c2
      EXPRESSION: c2::int
    - NAME: path
      EXPRESSION: path::text
METADATA:
  SCHEMA: staging_schema
COMMIT:
  MINIMAL_INTERVAL: 200
  CONSISTENCY: at-least
PROPERTIES:
  eof.when.idle: 1500
```

Greenplum Database Options (Version 2-Focused)

You identify the Greenplum Database connection options via the `DATABASE`, `USER`, `PASSWORD`, `HOST`, and `PORT` parameters.

The `VERSION` parameter identifies the version of the GPSS YAML configuration file. You must specify version `2` or version `v3`.

RABBITMQ:INPUT Options

Specify the RabbitMQ server, virtual host, and queue or stream of interest using the `SOURCE` block.

The `DATA` block that you provide must specify the `COLUMNS` and `FORMAT` parameters. The `DATA:COLUMNS` block includes the name and type of each data element in the RabbitMQ message. The default source-to-target data mapping behaviour of GPSS is to match a column name as defined in `COLUMNS:NAME` with a column name in the target Greenplum Database `OUTPUT:TABLE`:

- You must identify the RabbitMQ data elements in the order in which they appear in the RabbitMQ message.
- You may specify `NAME: __IGNORED__` to omit a RabbitMQ message value data element from the load operation.

- You must provide the same name for each non-ignored RabbitMQ data element and its associated Greenplum Database table column.
- You must specify an equivalent data type for each non-ignored RabbitMQ data element and its associated Greenplum Database table column.

The `DATA:FORMAT` keyword identifies the format of the RabbitMQ message value. GPSS supports comma-delimited text format (`csv`) and data that is separated by a configurable delimiter (`delimited`). GPSS also supports binary (`binary`), single object or single record per line JSON/JSONB (`json` or `jsonl`), and custom (`custom`) format value data.

When you provide a `META` block, you must specify a single JSON-type `COLUMNS` and the `FORMAT: json`.

The available RabbitMQ meta data properties for a streaming source include:

- `stream` (text) - the RabbitMQ stream name
- `offset` (bigint) - the message offset

The available RabbitMQ meta data properties for a queue source include:

- `queue` (text) - the RabbitMQ queue name
- `messageId` (text) - the message identifier
- `correlationId` (text) - the correlation identifier
- `timestamp` (bitint) - the time that the message was added to the RabbitMQ queue

The `FILTER` parameter identifies a filter to apply to the RabbitMQ input messages before the data is loaded into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. The message is dropped if the filter evaluates to `false`. The filter string must be a valid SQL conditional expression and may reference one or more `DATA` column names.

The `ERROR_LIMIT` parameter identifies the number of errors or the error percentage threshold after which GPSS should exit the load operation. The default `ERROR_LIMIT` is zero; the load operation is stopped when the first error is encountered.

RABBITMQ:OUTPUT Options

You identify the target Greenplum Database schema name and table name via the `RABBITMQ:OUTPUT:SCHEMA` and `TABLE` parameters. *You must pre-create the Greenplum Database table before you attempt to load RabbitMQ data.*

GPSS supports loading from a RabbitMQ data source into a single Greenplum Database table only.

The default load mode is to insert RabbitMQ data into the Greenplum Database table. GPSS also supports updating and merging RabbitMQ message data into a Greenplum table. You specify the load `MODE`, the `MATCH_COLUMNS` and `UPDATE_COLUMNS`, and any `UPDATE_CONDITION`s that must be met to merge or update the data. In `MERGE MODE`, you can also specify `ORDER_COLUMNS` to filter out duplicates and a `DELETE_CONDITION`.

You can override the default mapping of the `INPUT DATA:COLUMNS` by specifying a `MAPPING` block in which you identify the association between a specific column in the target Greenplum Database table and a RabbitMQ message value data element.

When you specify a `MAPPING` block, ensure that you provide entries for all RabbitMQ data elements of interest - GPSS does not automatically match column names when you provide a `MAPPING`.

About the Merge Load Mode

`MERGE` mode is similar to an `UPSERT` operation; GPSS may insert new rows in the database, or may update an existing database row that satisfies match and update conditions. GPSS deletes rows in `MERGE` mode when the data satisfies an optional `DELETE_CONDITION` that you specify.

GPSS stages a merge operation in a temporary table, generating the SQL to populate the temp table from the set of `OUTPUT` configuration properties that you provide.

GPSS uses the following algorithm for `MERGE` mode processing:

1. Create a temporary table like the target table.
2. Generate the SQL to insert the source data into the temporary table.
    1. Add the `MAPPINGS`.
    2. Add the `FILTER`.
    3. Use `MATCH_COLUMNS` and `ORDER_COLUMNS` to filter out duplicates.
3. Update the target table from rows in the temporary table that satisfy `MATCH_COLUMNS`, `UPDATE_COLUMNS`, and `UPDATE_CONDITION`.
4. Insert non-matching rows into the target table.
5. Delete rows in the target table that satisfy `MATCH_COLUMNS` and the `DELETE_CONDITION`.
6. Truncate the temporary table.

Other Options

The `RABBITMQ:METADATA:SCHEMA` parameter specifies the name of the Greenplum Database schema in which GPSS creates external tables.

GPSS commits RabbitMQ data to the Greenplum Database table at the row and/or time intervals that you specify in the `RABBITMQ:COMMIT: MAX_ROW` and/or `MINIMAL_INTERVAL` parameters. If you do not specify these properties, GPSS commits data at the default `MINIMAL_INTERVAL`, 5000ms.

Specify a `RABBITMQ:PROPERTIES` block to set RabbitMQ configuration properties. GPSS sends the property names and values to RabbitMQ when it instantiates a consumer for the load operation.

About the JSON Format and Column Type

When you specify `FORMAT: json` or `FORMAT: jsonl`, valid `COLUMN:TYPE`s for the data include `json` or `jsonb`. You can also specify the new GPSS `gp_jsonb` (Beta) or `gp_json` (Beta) column types.

- `gp_jsonb` is an enhanced JSONB type that adds support for `\u` escape sequences and unicode. For example, `gp_jsonb` can escape `\uDD8B` and `\u0000` as text format, but `jsonb` treats these characters as illegal.
- `gp_json` is an enhanced JSON type that can tolerate certain illegal unicode sequences. For example, `gp_json` automatically escapes incorrect surrogate pairs and processes `\u0000` as `\\u0000`. Note that unicode escape values cannot be used for code point values above `007F` when the server encoding is not `UTF8`.

You can use the `gp_jsonb` (Beta) and `gp_json` (Beta) data types as follows:

- As the `COLUMN:TYPE` when the target Greenplum Database table column type is `json` or `jsonb`.
- In a `MAPPING` when the target Greenplum Database column is `text` or `varchar`. For example:

  ```
  EXPRESSION: (j->>'a')::text
  ```

- In a `MAPPING` when `FORMAT: avro` and the target Greenplum Database column is `json` or `jsonb`. For example:

  ```
  EXPRESSION: j::gp_jsonb
  ```

  or

  ```
  EXPRESSION: j::gp_json
  ```

- In a `MAPPING` when `FORMAT: avro` and the target Greenplum Database column is `text` or `varchar`. For example:

  ```
  EXPRESSION: (j::gp_jsonb->>'a')::text
  ```

  or

  ```
  EXPRESSION: (j::gp_json->>'a')::text
  ```

The `gp_jsonb` (Beta) and `gp_json` (Beta) data types are defined in an extension named `dataflow`. You must `CREATE EXTENSION dataflow;` in each database in which you choose to use these (Beta) data types.

Preserving Ill-Formed JSON Escape Sequences

GPSS exposes a configuration parameter that you can use with the `gp_jsonb` and `gp_json` types. The name of this parameter is `gpss.json_preserve_ill_formed_prefix`. When set, GPSS does not return an error when it encounters an ill-formed JSON escape sequence with these types, but instead prepends it with the prefix that you specify.

For example, if `gpss.json_preserve_ill_formed_prefix` is set to the string `"##"` as follows:

```
SET gpss.json_preserve_ill_formed_prefix = "##";
```

and GPSS encounters an ill-formed JSON sequence such as the orphaned low surrogate `\ude04X`, GPSS writes the data as `##\ude04X` instead.

About Transforming and Mapping RabbitMQ Input Data

You can define a `MAPPING` between the RabbitMQ input data (`DATA:COLUMNS` and `META:COLUMNS`) and the columns in the target Greenplum Database table. Defining a mapping may be useful when you have a multi-field input column (such as a JSON-type column), and you want to assign individual components of the input field to specific columns in the target table.

You might also use a `MAPPING` to assign a value expression to a target table column. The expression must be one that you could specify in the `SELECT` list of a query, and can include a constant value, a column reference, an operator invocation, a built-in or user-defined function call, and so forth.

Creating the Greenplum Table

You must pre-create the Greenplum table before you load RabbitMQ data into Greenplum Database. You use the `RABBITMQ:OUTPUT: SCHEMA` and `TABLE` load configuration file parameters to identify the schema and table names.

The target Greenplum table definition must include each column that GPSS will load into the table. The table definition may include additional columns; GPSS ignores these columns, and loads no data into them.

The name and data type that you specify for a column of the target Greenplum Database table must match the name and data type of the related, non-ignored RabbitMQ message element. If you have defined a column mapping, the name of the Greenplum Database column must match the target column name that you specified for the mapping, and the type must match the target column type or expression that you define.

The `CREATE TABLE` command for the target Greenplum Database table receiving the RabbitMQ message data defined in the `loadcfgrmq2.yaml` file presented in the Constructing the rabbitmq.yaml Configuration File section follows:

```
testdb=# CREATE TABLE public.tbl_int_text_column( c1 int8, c2 int8, path text );
```

About RabbitMQ Stream Offsets, Message Retention, and Loading

This topic applies only when reading from a RabbitMQ stream.

RabbitMQ assigns each record/message within a stream a unique sequential id number. This id is referred to as an *offset*. GPSS retains, for each `gpsscli load` invocation specifying the same RabbitMQ *stream* and Greenplum Database table names, the last offset consumed by the load operation. Refer to Understanding RabbitMQ Message Offset Management for more detailed information about how GPSS manages RabbitMQ message offsets.

`gpsscli load` returns an error if its recorded offset for the RabbitMQ *stream* and Greenplum Database table combination is behind that of the current earliest RabbitMQ message offset for the topic, or when the

earliest and latest offsets do not match.

When you receive one of these messages, you can choose to:

- Resume the load operation from the earliest available message published to the stream by specifying the `--force-reset-earliest` option to `gpsscli load`:

  ```
  $ gpsscli load --force-reset-earliest loadcfg2.yaml
  ```

- Load only new messages published to the RabbitMQ stream, by specifying the `--force-reset-latest` option with the command:

  ```
  $ gpsscli load --force-reset-latest loadcfg2.yaml
  ```

- Load messages published since a specific timestamp (milliseconds since epoch), by specifying the `--force-reset-timestamp` option to `gpsscli load`:

  ```
  $ gpsscli load --force-reset-timestamp 1571066212000 loadcfg2.yaml
  ```

Specifying the `--force-reset-<xxx>` options when loading data may result in missing or duplicate messages. Use of these options outside of the offset mismatch scenario is discouraged.

Alternatively, you can provide the `FALLBACK_OPTION` (version 2) or `fallback_option` (version 3 (Beta)) property in the load configuration file to instruct GPSS to automatically read from the specified offset when it detects a mismatch.

Understanding RabbitMQ Message Offset Management

As a RabbitMQ consumer, GPSS manages the progress of each load operation using RabbitMQ's broker-based offset management. This management involves identifying how, when (before commit, after commit, or never), and where (history table, broker, both, nowhere) the offset is stored, and is directed by a combination of RabbitMQ client and GPSS load configuration properties.

RabbitMQ Properties

When loading from a RabbitMQ queue or stream, GPSS uses a default `consumer.name` of `gpss_rabbitmq_consumer_<job_id>`. You can set the consumer name by specifying this RabbitMQ client property in the `PROPERTIES` (version 2) or `properties` (version 3 (beta)) load configuration file block. For example:

```
PROPERTIES:
  consumer.name: my_rmq_consumer
```

When loading from a RabbitMQ queue, you may also choose to set the `consumer.exclusive` and qos.prefetch.count client properties, which specify the number of queue consumers and prefetch limits per consumer. GPSS sets the `consumer.exclusive` property to `false` by default.

GPSS Properties

GPSS uses the `CONSISTENCY` (version 2) or `consistency` (version 3 (Beta)) load configuration file property setting to govern how it manages offsets.

GPSS supports the following `CONSISTENCY` settings for RabbitMQ:

```
CONSISTENCY: { strong | at-least | at-most | none }
```

GPSS supports `strong` consistency for RabbitMQ streams only.

Summary

The following table summarizes the offset commit behaviour for load jobs for a GPSS RabbitMQ consumer:

Consistency Value	Behaviour
strong (available for RabbitMQ streams only)	GPSS stores offsets in a history table.
at-least [or empty]	GPSS stores in the offset in the RabbitMQ broker before commit. Note that this could result in duplicate data written to Greenplum Database when the connection to RabbitMQ closes after GPSS writes, but before it sends the response.
at-most	GPSS stores the offset in the RabbitMQ broker after commit.
none	GPSS does not store offsets anywhere.

Utility Reference

The VMware Greenplum Streaming Server includes the following utility and configuration reference pages:

gpss

Start an instance of a Greenplum Streaming Server.

Synopsis

```
gpss [-c | --config <config.json>]
    [--debug-port <portnum> ] [--clear-job-store]
    [--color] [--csv-log]
    [-l | --log-dir <directory>] [--verbose]

gpss {-h | --help}

gpss --version
```

Description

The `gpss` utility starts an instance of the Greenplum Database Streaming Server (GPSS). When you run the command, you optionally provide a JSON-formatted configuration file that defines run properties such as the GPSS server listen address, `gpfdist` host and port number, and encryption certificate and key files. You can also specify the directory to which GPSS writes its log files, as well as a job store.

When started, `gpss` waits indefinitely for client job requests. A single GPSS instance can service requests from multiple clients. Refer to the gpsscli reference page for more information about client job commands.

`gpss` keeps track of the status of each client job in memory. When you stop a GPSS server instance that did not specify a `JobStore` setting in its server configuration file, you lose all registered jobs. You must re-submit any previously-submitted jobs that you require after you restart the server instance. `gpss` will resume a job from the last load offset.

Options

-c | --config config.json

The JSON-formatted configuration file that defines the run properties of a GPSS service instance. If the filename provided is not an absolute path, GPSS assumes that the file system location is relative to the current working directory. Refer to gpss.json for the format and content of the properties that you specify in this file.

If you do not provide a GPSS configuration file, Greenplum Database starts `gpss` on default port `5000` on host `127.0.0.1`, starts `gpfdist` on default port `8080` on the local host address identified by the output of the `hostname` command, and does not use encryption.

--debug-port portnum

When you specify this option, `gpss` starts a debug server at the port identified by portnum on the local host (`127.0.0.1`); additional debug information including the call stack and performance statistics is available from the server.

A debug server port number that you specify via the `--debug-port` option to this command takes precedence over a `DebugPort` that you may have configured for the `gpss` server instance in the `gpss.json` configuration file.

--clear-job-store
: When you specify this option, `gpss` clears all jobs from the `JobStore` directory specified in the `config.json` server configuration file.

--color
: Enable the use of color when displaying front-end log messages. When specified, GPSS colors the log level in messages that it writes to `stdout`. Color is deactivated by default.

  GPSS ignores the `--color` option if you also specify `--csv-log`.

--csv-log
: Write front-end log messages in CSV format. By default, GPSS writes log messages to `stdout` using spaces between fields for a more human-readable format.

-l | --log-dir directory
: Specify the directory to which GPSS writes log files. GPSS must have write permission to the directory. GPSS creates the log directory if it does not exist.

  If you do not provide this option, GPSS writes `gpss` log files to the `$HOME/gpAdminLogs` directory.

--verbose
: The default behaviour of the command utility is to display information and error messages to `stdout`. When you specify the `--verbose` option, GPSS also outputs debug-level messages about the operation.

-h | --help
: Show command help, and then exit.

--version
: Show command version, and then exit.

Examples

Start a Greenplum Streaming Server instance in the background, specifying a log directory of `/home/gpadmin/logs.gpss`, and using the run properties defined in a configuration file named `gpsscfg.json` located in the current directory. Also start a GPSS debug server on port number 8915:

```
$ gpss --debug-port 8915 --log-dir /home/gpadmin/logs.gpss --config gpsscfg.json &
```

View the debug information available from the GPSS debug server:

```
$ curl http://127.0.0.1:8915/debug/pprof/
```

See Also

gpss.json, gpsscli

gpss.json

Greenplum Streaming Server configuration file.

Synopsis

```
{
    "ListenAddress": {
        "Host": "<gpss_host>",
        "Port": <gpss_portnum> [,
        "DebugPort": <gpss_debug_portnum>] [,
        "Certificate": {
            "CertFile": "<certfile_path>",
            "KeyFile": "<keyfile_path>",
            "CAFile": "<CAfile_path>",
            "MinTLSVersion": "<min_version>"
        }]
    },
    "Gpfdist": {
        "Host": "<gpfdist_host>",
        "Port": <gpfdist_portnum> [,
        "ReuseTables": <bool_value> ][,
        "Certificate": {
            "CertFile": "<certfile_path>",
            "KeyFile": "<keyfile_path>",
            "CAFile": "<CAfile_path>",
            "MinTLSVersion": "<min_version>",
            "DBClientShared": <bool_value>
        }] [,
        "BindAddress": <bind_addr> ]
    }[,
    "Shadow": {
        "Key": "<passwd_shadow_key>",
    }] [,
    "Authentication": {
        "Username": "<client_auth_username>",
        "Password": "SHADOW:<shadowed_passwd_string>"
    }] [,
    "JobStore": {
        "File": {
            "Directory": "<jobstore_dir>"
        }
    }] [,
    "Logging": {
        "BackendLevel": "<level>",
        "FrontendLevel": "<level>",
        "SplitByJob": "<level>",
        "Rotate": "<policy_period>"
    }] [,
    "Monitor": {
        "Prometheus": {
            "Listening": "<prom_addr>:<prom_port>"
        }
    }] [,
    "KeepAlive": {
        "MaxConnectionIdle": "<duration>",
        "Time": "<duration>",
        "Timeout": "<duration>",
        "MinTime": "<duration>",
        "PermitWithoutStream": <bool_value>
```

```
    }]
}
```

Description

You specify runtime configuration properties for a `gpss` service instance in a JSON-formatted configuration file. Run properties for GPSS include `gpss` and `gpfdist` service hosts, addresses, port numbers, and optional encryption certificates and key files. You can specify a directory in which GPSS stores job status information, addresses, port numbers, and optional encryption certificates and key files.

You can configure the log level of messages that GPSS commands write to `stdout` and to log files. Other logging-related properties allow you to configure a log rotation period and separate log files per job. You can also configure client-to-server authentication for GPSS. You can also specify a shadow key that GPSS uses to encode and decode the Greenplum Database and client authentication passwords.

This reference page uses the name `gpss.json` to refer to this file; you may choose your own name for the file.

Keywords and Values

GPSS `ListenAddress` Options

Host: gpss_host
: The host name or IP address on which GPSS listens for client connections. The default host is `127.0.0.1`.

Port: gpss_portnum
: The port number on which the `gpss` service instance listens. The default port number is 5000.

DebugPort: gpss_debug_portnum
: The port number on which `gpss` starts a debug server on the local host (`127.0.0.1`). When you specify this configuration option, `gpss` makes debug information including the call stack and performance statistics available from the debug server.

Certificates:
: When you specify certificates, GPSS uses the SSL certificates to authenticate both the client connection and the connection to Greenplum Database.

  CertFile: certfile_path
  : File system path to the server certificate.

  KeyFile: keyfile_path
  : File system path to the server key file.

  CAFile: CAfile_path
  : File system path to the Certificate Authority file. The CAfile_path must contain the entire Certificate Authority chain.

  MinTLSVersion: min_version
  : The minimum transport layer security (TLS) version that GPSS requests on the connection. The default value is `1.0`. GPSS supports minimum TLS versions of `1.0`, `1.1`, `1.2`, and `1.3`.

`Gpfdist` Options

GPSS implements the `gpfdist` protocol. The `gpss.json` file exposes configuration options that you can use to identify the service location and bind options.

Host: gpfdist_host
: The `gpfdist` service host name or IP address that GPSS sets in the external table `LOCATION` clause. This hostname or IP address must be reachable from each Greenplum Database segment host. The default value is the fully qualified distinguished name of the host on which GPSS is running.

Port: gpfdist_portnum
: The `gpfdist` service port number. The default port number is 8080.

ReuseTables: bool_value
: A boolean that identifies whether or not GPSS should reuse an external table when the job associated with a load operation is restarted. The default value is `true`, reuse the external table. When you reuse external tables, GPSS generates the external table name using a hash of various server and load configuration property values.

  If you choose not to reuse external tables, GPSS drops the external table associated with a load operation (if one exists) and creates a new external table when you (re)start the job. If you do not reuse external tables, GPSS generates the external table name using the job name.

Certificates:
: When you specify `gpfdist` certificates, GPSS uses the SSL certificates and the `gpfdists` protocol to transfer encrypted data between itself and Greenplum Database.

  CertFile: certfile_path
  : File system path to the server certificate.

  KeyFile: keyfile_path
  : File system path to the server key file.

  CAFile: CAfile_path
  : File system path to the Certificate Authority file. The CAfile_path must contain the entire Certificate Authority chain.

  MinTLSVersion: min_version
  : The minimum transport layer security (TLS) version that GPSS requests on the connection. The default value is `1.0`. GPSS supports minimum TLS versions of `1.0`, `1.1`, `1.2`, and `1.3`.

  DBClientShared: bool_value
  : Determines whether GPSS shares this Gpfdist certificate. The default value is `false`, GPSS does not share the cert. When `true`, GPSS presents the Gpfdist certificate as the client certificate for the control channel connection to Greenplum Database. This configuration may be desirable if you use `pgbouncer` to manage connections to Greenplum Database.

BindAddress: bind_addr
: The address from which GPSS listens for connections from Greenplum Database segments. Set bind_addr to an IP address that is reachable from every segment host by resolving the gpfdist_host configuration value, or set it to `0.0.0.0` to listen for connections from any host. The default bind address is `0.0.0.0`.

`Shadow` Option

The encode/decode key for the Greenplum Database password.

Shadow: passwd_shadow_key
: The key that GPSS uses to encode and decode shadow password strings. You can specify a shadowed password for the Greenplum Database user name. You can also specify a shadowed password for GPSS client to server `Authentication`. *Keep this key secret.*

`Authentication` Options

The user name and password that GPSS uses to authenticate the client program (`gpsscli`) with this GPSS server.

Username: client_auth_username
: The user name with which the GPSS server instance authenticates a client.

SHADOW:shadowed_passwd_string
: The shadowed password with which the GPSS server instance authenticates a client.

`JobStore` Option

The GPSS job information store.

File: Directory: jobstore_dir
: The file system directory on the local host in which GPSS maintains current job information and status.

`Logging` Options

The back-end (log file) and front-end (command line output) logging levels for GPSS commands. GPSS supports the following log levels (from most to least severe): `fatal`, `error`, `warn` or `warning`, `info`, and `debug`.

BackendLevel: level
: The logging level for messages that `gpss`, `gpsscli`, and `gpkafka` write to log files. The default back-end logging level is `debug`.

FrontendLevel: level
: The logging level for messages that `gpss`, `gpsscli`, and `gpkafka` write to `stdout`. The default front-end logging level is `info`.

SplitByJob: level
: Identifies if and how GPSS manages server log files. By default (no `SpitByJob` specified), GPSS creates a log file per server invocation. The only valid level is `StartTime`. When `StartTime` is specified, GPSS creates per-run server log files, and creates a new log file each time a job is started or loaded.
You may specify neither, or only one of `SplitByJob` or `Rotate`.

Rotate: policy_period
: The time period that governs the GPSS server log file rotation policy. Valid policy period values are `daily` and `hourly`. By default (no `Rotate` specified), GPSS does not rotate a server log file on its own.
You may specify neither, or only one of `SplitByJob` or `Rotate`.

`Monitor` Option

The address to which GPSS binds the Prometheus scrape target for the GPSS server instance.

Prometheus: Listening: prom_addr:prom_port
: The host name or IP address and port number from which the GPSS service instance allows Prometheus to pull the server's metrics.

`KeepAlive` **Options**

These properties control the gRPC connection between the GPSS client and GPSS server. Refer to the gRPC keepalive package documentation for more information about these properties and their default values.

MaxConnectionIdle: duration
: The amount of time after which an idle connection is closed. The default value is infinity.

Time: duration
: The amount of time of no activity after which a server pings the client to see if the transport is still alive. The default value is `2h`.

Timeout: duration
: The amount of time of no activity after a keepalive ping that prompts closure of the connection. The default value is `20s`.

MinTime: duration
: The mininum amount of time a client should wait before sending a keepalive ping. The default value is `5m`.

PermitWithoutStream: bool_value
: Determines if the server allows keepalive pings when there are no active streams. If `false`, the server will close the connection when a client sends a ping with no active streams. The default value is `false`.

Notes

When you provide the `--config gpss.json` option to the `gpsscli shadow` command, GPSS uses the `Shadow:Key` specified in the file to encode the password that you specify. If you do not provide a gpss.json file, GPSS uses a default key to encode the password. The `gpsscli shadow` command generates and outputs the encoded shadow password string.

When you provide a variant of this file to the other `gpsscli` subcommands via the `--config gpsscliconfig.json` option, GPSS uses the information provided in the `ListenAddress` block of the file to identify the GPSS server instance to which to route the request, and/or to identify the client certificates when SSL is enabled between GPSS client and server.

When you provide the `--config gpfdistconfig.json` option to the `gpkafka load` command, GPSS uses the information provided in the `Gpfdist` block of the file to specify the `gpfdist` protocol instance, and, when SSL is enabled on the data channel to Greenplum Database, to identify the GPSS SSL certificates.

Examples

Start a Greenplum Streaming Server instance with properties as defined in a configuration file named `gpss4ic.json` located in the current directory:

```
$ gpss --config gpss4ic.json
```

Example `gpss4ic.json` configuration file:

```
{
    "ListenAddress": {
        "Host": "",
        "Port": 5019
    },
    "Gpfdist": {
        "Host": "",
        "Port": 8319,
        "ReuseTables": false
    },
    "Shadow": {
        "Key": "a_very_secret_key"
    },
    "Monitor": {
        "Prometheus": {
            "Listening": "0.0.0.0:5001"
    },
    "JobStore": {
        "File": {
            "Directory": "/home/gpadmin/jobstore"
        }
    }
}
```

See Also

gpss, gpsscli, gpkafka load

gpsscli

Client command utility for the Greenplum Streaming Server.

Synopsis

```
gpsscli <subcommand> [<options>]

gpsscli convert
gpsscli dryrun
gpsscli list
gpsscli load
gpsscli progress
gpsscli remove
gpsscli start
gpsscli status
gpsscli stop
gpsscli submit
gpsscli wait

gpsscli {help | -h | --help}

gpsscli {-v | --version}
```

Description

The Greenplum Streaming Server (GPSS) includes the `gpsscli` client command utility. `gpsscli` provides subcommands to manage Greenplum Streaming Server load jobs and to view job status and progress:

Subcommand	Description
convert	Convert a version 1 or 2 load configuration file to version 3 (Beta) format
dryrun	Perform a trial load without writing to Greenplum Database
help	Display command help
list	List jobs and their status
load	Run one or more single-command load
progress	Show job progress
remove	Remove one or more jobs
start	Start one or more jobs
status	Show job status
stop	Stop one or more jobs
submit	Submit one or more jobs
wait	Wait for a job to stop

Options

--config gpsscliconfig.json

The GPSS configuration file. This file includes properties that identify the `gpss` instance that services the command. When SSL encryption is enabled between the GPSS client and server, you also use this file to identify the file system location of the client SSL certificates. Refer to gpss.json for detailed information about the format of this file and the configuration properties supported.

`gpsscli` subcommands read the configuration specified in the `ListenAddress` block of the `gpsscliconfig.json` file, and ignore the `gpfdist` configuration specified in the `Gpfdist` block of the file.

--color

Enable the use of color when displaying front-end log messages. When specified, GPSS colors the log level in messages that it writes to `stdout`. Color is deactivate by default.

GPSS ignores the `--color` option if you also specify `--csv-log`.

--csv-log

Write front-end log messages in CSV format. By default, GPSS writes log messages to `stdout` using spaces between fields for a more human-readable format.

--gpss-host host

The GPSS host. The default host address is `127.0.0.1`. If specified, overrides a `ListenAddress:Host` value provided in `gpsscliconfig.json`

--gpss-port port
: The GPSS port number. The default port number is `5000`. If specified, overrides a `ListenAddress:Port` value provided in `gpsscliconfig.json`

--no-check-ca
: Deactivate certificate verification when SSL is enabled between the GPSS client and server. By default, GPSS checks the certificate authority (CA) each time that you invoke a `gpsscli` subcommand.

-U | --username client_auth_username
: The user name with which the GPSS server instance authenticates the client.

-P | --password client_auth_passwd
: The password with which the GPSS server instance authenticates the client.

-l | --log-dir directory
: The directory to which GPSS writes client command log files. GPSS must have write permission to the directory. GPSS creates the log directory if it does not exist.

  If you do not provide this option, GPSS writes `gpsscli` client log files to the `$HOME/gpAdminLogs` directory.

--verbose
: The default behaviour of the command utility is to display information and error messages to `stdout`. When you specify the `--verbose` option, GPSS also outputs debug-level messages about the operation.

-h | --help
: Show command utility help, and then exit.

-v | --version
: Show the versions of `gpsscli` and the `gpss` server, and then exit. You may need to also specify the `--gpss-host` and/or `--gpss-port` options to identify the server.

See Also

gpsscli convert, gpsscli dryrun, gpsscli list, gpsscli load, gpsscli progress, gpsscli remove, gpsscli start, gpsscli status, gpsscli stop, gpsscli submit, gpsscli wait

gpsscli convert

Convert a version 1 or version 2 load configuration file to version 3 (Beta) format.

Synopsis

```
gpsscli convert <loadcfg.yaml>
    [-i | --edit-in-place]

gpsscli convert {-h | --help}
```

Description

The `gpsscli convert` command converts a version 1 or version 2 load configuration file to version 3 (Beta) format.

By default, the command writes the converted file to `stdout`. If you specify the `-i` or `--edit-in-place` option, the command overwrites the input loadcfg.yaml with the new version 3 format.

`gpsscli convert` does not currently support converting a load configuration file in which properties are set using template parameters.

Options

loadcfg.yaml
: The load configuration file.

-i | --edit-in-place
: Instead of writing to `stdout`, edit the file in place, overwriting the input file with the new version 3 format.

-h | --help
: Show command utility help, and then exit.

Examples

Convert the version 2 load configuration file named `etljob.yaml` to version 3 format, editting the file in place:

```
$ gpsscli convert --edit-in-place etljob.yaml
convert file in place successfully
```

See Also

gpsscli.yaml, gpsscli-v3.yaml (Beta)

gpsscli dryrun

Perform a trial load without writing to Greenplum Database.

Synopsis

```
gpsscli dryrun <loadcfgv3.yaml> [--name <job_name>]
    [-p | --property <template_var=value>]
    [--include-error-process]
    [--config <gpsscliconfig.json>]
    [--gpss-host <host>] [--gpss-port <port>]
    [--no-check-ca] [-l | --log-dir <directory>] [--verbose]

gpsscli dryrun {-h | --help}
```

Description

The `gpsscli dryrun` command performs a trial load of the job specified by loadcfgv3.yaml. You can use this command to help you diagnose a failed job.

GPSS currently supports running the `gpsscli dryrun` command only on Kafka, file, and S3 jobs.

`gpsscli dryrun` reads data from the source and prepares for the load operation without actually inserting the data into Greenplum Database. The command also outputs the underlying SQL commands that GPSS would run to fulfill the job.

If the GPSS instance to which you want to send the request is not running on the default host (`127.0.0.1`) or the default port number (`5000`), you can specify the GPSS host and/or port via command line options.

Options

loadcfgv3.yaml
: The version 3 load configuration file with which to perform a job trial run.

--name job_name
: The name to assign to the dry run job.

-p | --property template_var=value
: Substitute value for instances of the property value template {{template_var}} referenced in the loadcfgv3.yaml load configuration file.

--include-error-process

: Instructs GPSS to process errors and display error messages when `save_failing_batch` is `true`. By default, the command does not display these messages.

`gpsscli dryrun` supports this option only when trialing a Kafka load configuration file.

--config gpsscliconfig.json

: The GPSS configuration file. This file includes properties that identify the `gpss` instance that services the command. When SSL encryption is enabled between the GPSS client and server, you also use this file to identify the file system location of the client SSL certificates. Refer to gpss.json for detailed information about the format of this file and the configuration properties supported.

`gpsscli` subcommands read the configuration specified in the `ListenAddress` block of the `gpsscliconfig.json` file, and ignore the `gpfdist` configuration specified in the `Gpfdist` block of the file.

--color

Enable the use of color when displaying front-end log messages. When specified, GPSS colors the log level in messages that it writes to `stdout`. Color is deactivated by default.

GPSS ignores the `--color` option if you also specify `--csv-log`.

--csv-log
: Write front-end log messages in CSV format. By default, GPSS writes log messages to `stdout` using spaces between fields for a more human-readable format.

--gpss-host host
: The GPSS host. The default host address is `127.0.0.1`. If specified, overrides a `ListenAddress:Host` value provided in `gpsscliconfig.json`

--gpss-port port
: The GPSS port number. The default port number is `5000`. If specified, overrides a `ListenAddress:Port` value provided in `gpsscliconfig.json`

--no-check-ca
: Deactivate certificate verification when SSL is enabled between the GPSS client and server. By default, GPSS checks the certificate authority (CA) each time that you invoke a `gpsscli` subcommand.

-U | --username client_auth_username
: The user name with which the GPSS server instance authenticates the client.

-P | --password client_auth_passwd
: The password with which the GPSS server instance authenticates the client.

-l | --log-dir directory
: The directory to which GPSS writes client command log files. GPSS must have write permission to the directory. GPSS creates the log directory if it does not exist.

If you do not provide this option, GPSS writes `gpsscli` client log files to the `$HOME/gpAdminLogs` directory.

--verbose
: The default behaviour of the command utility is to display information and error messages to `stdout`. When you specify the `--verbose` option, GPSS also outputs debug-level messages about the operation.

-h | --help
: Show command utility help, and then exit.

Examples

Perform a dry run of the Kafka load job specified by the configuration file named `kjobcfgv3.yaml`; include error processing in the trial run:

```
$ gpsscli dryrun --include-error-process kjobcfgv3.yaml
```

See Also

gpsscli.yaml, gpsscli-v3.yaml (Beta)

gpsscli list

List Greenplum Streaming Server jobs and status.

Synopsis

```
gpsscli list [--all]
     [--config <gpsscliconfig.json>]
     [--gpss-host <host>] [--gpss-port <port>]
     [-U | --username <client_auth_user> -P | --password <client_auth_passwd>]
     [--no-check-ca] [-l | --log-dir <directory>] [--verbose]

gpsscli list {-h | --help}
```

Description

The `gpsscli list` command lists the running jobs serviced by a specific Greenplum Streaming Server (GPSS) instance. You can instruct the command to list all jobs regardless of state by providing the `--all` flag.

`gpsscli list` displays the job identifier and status, as well as the Greenplum Database coordinator host, port, database, schema, and table.

If the GPSS instance to which you want to send the request is not running on the default host (`127.0.0.1`) or the default port number (`5000`), you can specify the GPSS host and/or port via command line options.

Options

--all
: Show information for all jobs regardless of state.

--config gpsscliconfig.json
: The GPSS configuration file. This file includes properties that identify the `gpss` instance that services the command. When SSL encryption is enabled between the GPSS client and server, you also use this file to identify the file system location of the client SSL certificates. Refer to gpss.json for detailed information about the format of this file and the configuration properties supported.

`gpsscli` subcommands read the configuration specified in the `ListenAddress` block of the `gpsscliconfig.json` file, and ignore the `gpfdist` configuration specified in the `Gpfdist` block of the file.

--color
: Enable the use of color when displaying front-end log messages. When specified, GPSS colors the log level in messages that it writes to `stdout`. Color is deactivated by default.

  GPSS ignores the `--color` option if you also specify `--csv-log`.

--csv-log
: Write front-end log messages in CSV format. By default, GPSS writes log messages to `stdout` using spaces between fields for a more human-readable format.

--gpss-host host

The GPSS host. The default host address is `127.0.0.1`. If specified, overrides a `ListenAddress:Host` value provided in `gpsscliconfig.json`

--gpss-port port
: The GPSS port number. The default port number is `5000`. If specified, overrides a `ListenAddress:Port` value provided in `gpsscliconfig.json`

--no-check-ca
: Deactivate certificate verification when SSL is enabled between the GPSS client and server. By default, GPSS checks the certificate authority (CA) each time that you invoke a `gpsscli` subcommand.

-U | --username client_auth_username
: The user name with which the GPSS server instance authenticates the client.

-P | --password client_auth_passwd
: The password with which the GPSS server instance authenticates the client.

-l | --log-dir directory
: The directory to which GPSS writes client command log files. GPSS must have write permission to the directory. GPSS creates the log directory if it does not exist.

  If you do not provide this option, GPSS writes `gpsscli` client log files to the `$HOME/gpAdminLogs` directory.

--verbose
: The default behaviour of the command utility is to display information and error messages to `stdout`. When you specify the `--verbose` option, GPSS also outputs debug-level messages about the operation.

-h | --help
: Show command utility help, and then exit.

Examples

List all Greenplum Streaming Server jobs:

```
$ gpsscli list --all
```

See Also

gpss, gpsscli submit

gpsscli load

Load data with the Greenplum Streaming Server.

Synopsis

```
gpsscli load <jobconfig.yaml> [...]
     [--name <job_name>]
     [-f | --force] [--quit-at-eof] [--partition]
     [{--force-reset-earliest | --force-reset-latest | --force-reset-timestamp <tstamp
>}]
     [-p | --property <template_var=value>]
```

```
     [--config <gpsscliconfig.json>]
     [--gpss-host <host>] [--gpss-port <port>]
     [-U | -X-username <client_auth_user> -P | --password <client_auth_passwd>]
     [--no-check-ca] [-l | --log-dir <directory>] [--verbose]

gpsscli load {-h | --help}
```

Description

The `gpsscli load` command initiates a load job to a specific Greenplum Streaming Server (GPSS) instance. When you run `gpsscli load`, the command submits, starts, and displays the progress of a GPSS job.

You provide one or more YAML-formatted configuration files that define the job parameters when you run the command. When you specify a single load configuration file, you may choose a name to identify the job. If you do not provide a name, GPSS uses the base name of load configuration file as the job identifier. For example, if you invoke this command with the load configuration file `/dir/jobconfig.yaml` and do not provide the `--name` option, GPSS assigns the job the identifier `jobconfig`.

By default, `gpsscli load` loads all available data and then waits indefinitely for new messages to load. In the case of user interrupt or exit, the GPSS job remains in the *Running* state. You must explicitly stop the job with `gpsscli stop` when running in this mode.

When you provide the `--quit-at-eof` option to the command, the utility exits after it reads all published data, writes the data to Greenplum Database, and stops the job. The GPSS job is in the *Success* or *Error* state when the command returns.

If `gpsscli load` detects an offset mismatch when loading from a Kafka data source, you can choose to resume a load operation from the earliest available data. Or, you may choose to load only new data, or data emitted since a specific time.

If the GPSS instance to which you want to send the request is not running on the default host (`127.0.0.1`) or the default port number (`5000`), you can specify the GPSS host and/or port via command line options.

Options

jobconfig.yaml [...]

One or more YAML-formatted configuration files that define the parameters of the job. If a filename provided is not an absolute path, Greenplum Database assumes the file system location is relative to the current working directory.

GPSS uses the properties in a YAML configuration file to uniquely identify a load operation. Submit a configuration file only once. If you submit the same configuration file more than once, GPSS will create the job, but it will eventually error out.

--name job_name

Use job_name to identify the job. If you do not provide a name, the default job identifier is the base name of the load configuration file. Job names must be unique.

GPSS does not support specifying a job_name when you provide more than one jobconfig.yaml load configuration file to the command.

-f | --force

Force GPSS to reload the configuration of a running job. GPSS stops the job, updates the job with the configuration specified in in `jobconfig.yaml`, and then restarts the job. If you previously named the job, you must provide `--name job\_name` when you force job configuration reload with this option.

Do not attempt to update a configuration property that GPSS uses to uniquely identify a job. If you change any such configuration property, GPSS creates a new internal job and loads all available messages.

--quit-at-eof

When you specify this option, `gpsscli load` exits after it reads all of the source data. The default behaviour of `gpsscli load` is to wait indefinitely for, and then consume, new data from the source.

`gpsscli load` ignores job retry `SCHEDULE` configuration settings when it is invoked with the `--quit-at-eof` flag.

--force-reset-earliest

`gpsscli load` returns an error if its recorded offset does not match that of the data source. Re-run `gpsscli load` and specify the `--force-reset-earliest` option to resume the load operation from the earliest available data offset known to the data source.

`gpsscli load` supports this option only when loading from a Kafka or RabbitMQ stream data source.

`--force-reset-earliest` specified on the command line takes precedence over a `FALLBACK_OFFSET/fallback_offset` set in the jobconfig.yaml.

--force-reset-latest

`gpsscli load` returns an error if its recorded offset does not match that of the data source. Re-run `gpsscli load` and specify the `--force-reset-latest` option to load only new data emitted from the data source.

`gpsscli load` supports this option only when loading from a Kafka or RabbitMQ

stream data source.

`--force-reset-latest` specified on the command line takes precedence over a `FALLBACK_OFFSET/fallback_offset` set in the jobconfig.yaml.

--force-reset-timestamp tstamp

Specify the `--force-reset-timestamp` option to load messages published since the specified time. tstamp must specify epoch time in milliseconds, and is bounded by the earliest message time and the current time.

`gpsscli load` supports this option only when loading from a Kafka or RabbitMQ stream data source.

--partition

By default, GPSS outputs the Kafka job progress by batch, and displays the start and end times, the message number and size, the number of inserted and rejected rows, and the transfer speed per batch. When you specify the `--partition` option, GPSS outputs the job progress by partition, and displays the partition identifier, the start and end times, the beginning and ending offsets, the message size, and the transfer speed per partition.

`gpsscli load` supports this option only when loading from a Kafka data source.

-p | --property template_var=value

Substitute value for instances of the property value template {{template_var}} referenced in the jobconfig.yaml load configuration file.

--config gpsscliconfig.json

The GPSS configuration file. This file includes properties that identify the `gpss` instance that services the command. When SSL encryption is enabled between the GPSS client and server, you also use this file to identify the file system location of the client SSL certificates. Refer to gpss.json for detailed information about the format of this file and the configuration properties supported.

`gpsscli` subcommands read the configuration specified in the `ListenAddress` block of the `gpsscliconfig.json` file, and ignore the `gpfdist` configuration specified in the `Gpfdist` block of the file.

--color

Enable the use of color when displaying front-end log messages. When specified, GPSS colors the log level in messages that it writes to `stdout`. Color is deactivated by default.

GPSS ignores the `--color` option if you also specify `--csv-log`.

--csv-log
: Write front-end log messages in CSV format. By default, GPSS writes log messages to `stdout` using spaces between fields for a more human-readable format.

--gpss-host host
: The GPSS host. The default host address is `127.0.0.1`. If specified, overrides a `ListenAddress:Host` value provided in `gpsscliconfig.json`

--gpss-port port
: The GPSS port number. The default port number is `5000`. If specified, overrides a `ListenAddress:Port` value provided in `gpsscliconfig.json`

--no-check-ca
: Deactivate certificate verification when SSL is enabled between the GPSS client and server. By default, GPSS checks the certificate authority (CA) each time that you invoke a `gpsscli` subcommand.

-U | --username client_auth_username
: The user name with which the GPSS server instance authenticates the client.

-P | --password client_auth_passwd
: The password with which the GPSS server instance authenticates the client.

-l | --log-dir directory
: The directory to which GPSS writes client command log files. GPSS must have write permission to the directory. GPSS creates the log directory if it does not exist.

  If you do not provide this option, GPSS writes `gpsscli` client log files to the `$HOME/gpAdminLogs` directory.

--verbose
: The default behaviour of the command utility is to display information and error messages to `stdout`. When you specify the `--verbose` option, GPSS also outputs debug-level messages about the operation.

-h | --help
: Show command utility help, and then exit.

Examples

Submit a GPSS load job from Kafka named `from_topic1` whose load parameters are defined by the configuration file named `loadcfg.yaml`:

```
$ gpsscli load --name from_topic1 loadcfg.yaml
```

See Also

gpss, gpsscli.yaml, gpsscli submit, gpsscli start, gpsscli progress, gpsscli stop

gpsscli progress

Check the progress of a Greenplum Streaming Server job.

Synopsis

```
gpsscli progress {<job_name> | <job_id>}
     [--partition]
     [--config <gpsscliconfig.json>]
     [--gpss-host <host>] [--gpss-port <port>]
     [-U | --username <client_auth_user> -P | --password <client_auth_passwd>]
     [--no-check-ca] [-l | --log-dir <directory>] [--verbose]

gpsscli progress {-h | --help}
```

Description

GPSS currently supports job progress tracking only for Kafka data sources.

The `gpsscli progress` command displays the progress of a Kafka job submitted to a Greenplum Streaming Server (GPSS) instance. The command displays time, offset information, message size, and transfer speed for each data load operation real-time. By default, `gpsscli progress` displays information per-batch. You can specify the `--partition` option to display progress information on a per-partition basis.

`gpsscli progress` waits indefinitely; the command exits when the job is stopped or GPSS encounters an error.

If the GPSS instance to which you want to send the request is not running on the default host (`127.0.0.1`) or the default port number (`5000`), you can specify the GPSS host and/or port via command line options.

Options

job_name | job_id
: The identifier of a previously-submitted GPSS job. You can specify a job name when you run `gpsscli submit`, or the command returns a unique job identifier.

--partition
: By default, GPSS outputs the job progress by batch, and displays the start and end times, the message number and size, the number of inserted and rejected rows, and the transfer speed per batch. When you specify the `--partition` option, GPSS outputs the job progress by partition, and displays the partition identifier, the start and end times, the beginning and ending offsets, the message size, and the transfer speed per partition.

--config gpsscliconfig.json
: The GPSS configuration file. This file includes properties that identify the `gpss` instance that services the command. When SSL encryption is enabled between the GPSS client and server, you also use this file to identify the file system location of the client SSL certificates. Refer to gpss.json for detailed information about the format of this file and the configuration properties supported.

`gpsscli` subcommands read the configuration specified in the `ListenAddress` block of the `gpsscliconfig.json` file, and ignore the `gpfdist` configuration specified in the `Gpfdist` block of the file.

--color

Enable the use of color when displaying front-end log messages. When specified, GPSS colors the log level in messages that it writes to `stdout`. Color is deactivated by default.

GPSS ignores the `--color` option if you also specify `--csv-log`.

--csv-log

Write front-end log messages in CSV format. By default, GPSS writes log messages to `stdout` using spaces between fields for a more human-readable format.

--gpss-host host

The GPSS host. The default host address is `127.0.0.1`. If specified, overrides a `ListenAddress:Host` value provided in `gpsscliconfig.json`

--gpss-port port

The GPSS port number. The default port number is `5000`. If specified, overrides a `ListenAddress:Port` value provided in `gpsscliconfig.json`

--no-check-ca

Deactivate certificate verification when SSL is enabled between the GPSS client and server. By default, GPSS checks the certificate authority (CA) each time that you invoke a `gpsscli` subcommand.

-U | --username client_auth_username

The user name with which the GPSS server instance authenticates the client.

-P | --password client_auth_passwd

The password with which the GPSS server instance authenticates the client.

-l | --log-dir directory

The directory to which GPSS writes client command log files. GPSS must have write permission to the directory. GPSS creates the log directory if it does not exist.

If you do not provide this option, GPSS writes `gpsscli` client log files to the `$HOME/gpAdminLogs` directory.

--verbose

The default behaviour of the command utility is to display information and error messages to `stdout`. When you specify the `--verbose` option, GPSS also outputs debug-level messages about the operation.

-h | --help

Show command utility help, and then exit.

Examples

Display the progress of the GPSS job identified by the name `nsync_121118`:

```
$ gpsscli progress nsync_121118
```

Display the per-partition progress of the GPSS job identified by the name `nsync_121118`:

```
$ gpsscli progress nsync_121118 --partition
```

See Also

gpss, gpsscli list, gpsscli submit

gpsscli remove

Remove one or more Greenplum Streaming Server jobs.

Synopsis

```
gpsscli remove {<job_name> | <job_id>}
     [--all]
     [-f | --force]
     [--config <gpsscliconfig.json>]
     [--gpss-host <host>] [--gpss-port <port>]
     [-U | --username <client_auth_user> -P | --password <client_auth_passwd>]
     [--no-check-ca] [-l | --log-dir <directory>] [--verbose]

gpsscli remove {-h | --help}
```

Description

The `gpsscli remove` command removes a job from the job list of a specific Greenplum Streaming Server (GPSS) instance.

Specify the `--all` flag to the command to remove all stopped and errored jobs. Specify the `-f | --force` flag to instruct GPSS to forcibly stop and remove the job(s).

When you remove a job, GPSS un-registers the job and removes all job-related resources.

If the GPSS instance to which you want to send the request is not running on the default host (`127.0.0.1`) or the default port number (`5000`), you can specify the GPSS host and/or port via command line options.

Options

job_name | job_id
: The identifier of a previously-submitted GPSS job. You can specify a job name when you run `gpsscli submit`, or the command returns a unique job identifier.

--all
: Remove all stopped and errored jobs.

-f | --force
: Forcibly stop (if running) and remove the specified job, or stop and remove all jobs when the `--all` flag is also specified.

--config gpsscliconfig.json
: The GPSS configuration file. This file includes properties that identify the `gpss` instance that services the command. When SSL encryption is enabled between the GPSS client and server, you also use this file to identify the file system location of the client SSL certificates. Refer to gpss.json for detailed information about the format of this file and the configuration properties supported.

`gpsscli` subcommands read the configuration specified in the `ListenAddress` block of the `gpsscliconfig.json` file, and ignore the `gpfdist` configuration specified in the `Gpfdist` block of the file.

--color

Enable the use of color when displaying front-end log messages. When specified, GPSS colors the log level in messages that it writes to `stdout`. Color is deactivated by default.

GPSS ignores the `--color` option if you also specify `--csv-log`.

--csv-log

Write front-end log messages in CSV format. By default, GPSS writes log messages to `stdout` using spaces between fields for a more human-readable format.

--gpss-host host

The GPSS host. The default host address is `127.0.0.1`. If specified, overrides a `ListenAddress:Host` value provided in `gpsscliconfig.json`

--gpss-port port

The GPSS port number. The default port number is `5000`. If specified, overrides a `ListenAddress:Port` value provided in `gpsscliconfig.json`

--no-check-ca

Deactivate certificate verification when SSL is enabled between the GPSS client and server. By default, GPSS checks the certificate authority (CA) each time that you invoke a `gpsscli` subcommand.

-U | --username client_auth_username

The user name with which the GPSS server instance authenticates the client.

-P | --password client_auth_passwd

The password with which the GPSS server instance authenticates the client.

-l | --log-dir directory

The directory to which GPSS writes client command log files. GPSS must have write permission to the directory. GPSS creates the log directory if it does not exist.

If you do not provide this option, GPSS writes `gpsscli` client log files to the `$HOME/gpAdminLogs` directory.

--verbose

The default behaviour of the command utility is to display information and error messages to `stdout`. When you specify the `--verbose` option, GPSS also outputs debug-level messages about the operation.

-h | --help

Show command utility help, and then exit.

Examples

Remove the GPSS job identified by the name `nsync_121118`:

```
$ gpsscli remove nsync_121118
```

See Also

gpss, gpsscli list, gpsscli stop, gpsscli submit

gpsscli shadow

Generate a shadow string for a password.

Synopsis

```
gpsscli shadow [--config <gpss.json>]

gpsscli shadow {-h | --help}
```

Description

The `gpsscli shadow` command generates a shadowed password string for the password that you input. You may generate a shadow password for the Greenplum Database user password, or for the password that a GPSS server instance uses to authenticate a client.

If you generate a Greenplum user shadow password, you provide the output generated by the command in the `PASSWORD:` property setting of your `gpsscli.yaml` load configuration file.

GPSS supports shadowing the Greenplum user password only on load jobs that you submit and manage with the `gpsscli` subcommands. GPSS does not support shadowed passwords on load jobs that you submit with `gpkafka load`.

If you generate a shadow password for GPSS client to server authentication, you provide the output generated by the command in the `Authentication:Password:` property setting of the `gpss.json` GPSS server configuration file.

GPSS uses the `Shadow:Key` property value specified in the `--config gpss.json` file, or the default key, to encode the password that you enter, and then generates and outputs the shadow password string.

`gpsscli shadow` generates a shadow password string of the following format:

```
"SHADOW:<shadow_password_string>"
```

Options

--config gpss.json
: The GPSS server configuration file. When the file includes a `Shadow:Key` setting, GPSS uses the key to encode the password input.

-h | --help
: Show command utility help, and then exit.

Examples

Generate a shadowed password string using the default key:

```
$ gpsscli shadow
please input your password
changemeCHANGEMEchangeme
"SHADOW:OH7TIH3676NIA4GQNNN3NTIEJGMGY7R2UE2A4GYUV2ED5AESDHWA"
```

Generate, without the user prompt, a shadowed password string using the key specified in the file named `gpss.json` located in the current working directory:

```
$ echo changemeCHANGEMEchangeme | gpsscli shadow --config gpss.json | tail -1
"SHADOW:ERTBKXDWLAJHUF5UOGJY34QTXIBNYP4ULTWVHIUZIF4UYFPRIJVA"
```

See Also

gpss, gpss.json, gpsscli.yaml

gpsscli start

(Re)start one or more Greenplum Streaming Server jobs.

Synopsis

```
gpsscli start {<job_name> | <job_id>}
     [--all] [--quit-at-eof]
     [{--force-reset-earliest | --force-reset-latest | --force-reset-timestamp <tstamp
>}]
     [--skip-explain]
     [--config <gpsscliconfig.json>]
     [--gpss-host <host>] [--gpss-port <port>]
     [-U | --username <client_auth_user> -P | --password <client_auth_passwd>]
     [--no-check-ca] [-l | --log-dir <directory>] [--verbose]

gpsscli start {-h | --help}
```

Description

The `gpsscli start` command (re)starts a job submitted to a specific Greenplum Streaming Server (GPSS) instance. You identify the name of the job. You can also identify the data offset from which you want the operation to begin.

Specify the `--all` flag to the command to (re)start all previously submitted jobs.

When you start a job, you initiate the data load operation. The job transitions from the *Submitted* or *Stopped* state to the *Running* state.

If the GPSS instance to which you want to send the request is not running on the default host (`127.0.0.1`) or the default port number (`5000`), you can specify the GPSS host and/or port via command line options.

Options

job_name | job_id

The identifier of a previously-submitted GPSS job. You can specify a job name when you run `gpsscli submit`, or the command returns a unique job identifier.

--all

Start all previously submitted jobs that are not currently running.

--quit-at-eof

When you specify this option, `gpsscli start` reads all pending data, stops the job, and then exits. The default behaviour of `gpsscli start` is to start the job and then exit.

`gpsscli start` ignores job retry `SCHEDULE` configuration settings when it is invoked with the `--quit-at-eof` flag.

--force-reset-earliest

`gpsscli start` returns an error if its recorded offset does not match that of the data source. Re-run `gpsscli start` and specify the `--force-reset-earliest` option to resume the load operation from the earliest available data offset known to the data source.

`gpsscli start` supports this option only when loading from a Kafka or RabbitMQ stream data source.

`--force-reset-earliest` specified on the command line takes precedence over a `FALLBACK_OFFSET/fallback_offset` set in the jobconfig.yaml when the job was submitted.

--force-reset-latest

`gpsscli start` returns an error if its recorded offset does not match that of the data source. Re-run `gpsscli start` and specify the `--force-reset-latest` option to load only new data emitted from the data source.

`gpsscli start` supports this option only when loading from a Kafka or RabbitMQ stream data source.

`--force-reset-latest` specified on the command line takes precedence over a `FALLBACK_OFFSET/fallback_offset` set in the jobconfig.yaml when the job was submitted.

--force-reset-timestamp tstamp

Specify the `--force-reset-timestamp` option to load messages published since the specified time. tstamp must specify epoch time in milliseconds, and is bounded by the earliest message time and the current time.

`gpsscli start` supports this option only when loading from a Kafka or RabbitMQ stream data source.

--skip-explain

Instructs GPSS to skip the explain SQL check step in its internal processing.

`gpsscli start` supports this option only when loading from a Kafka data source.

--config gpsscliconfig.json

The GPSS configuration file. This file includes properties that identify the `gpss` instance that services the command. When SSL encryption is enabled between the GPSS client and server, you also use this file to identify the file system location of the client SSL certificates. Refer to gpss.json for detailed information about the format of this file and the configuration properties supported.

`gpsscli` subcommands read the configuration specified in the `ListenAddress` block of the `gpsscliconfig.json` file, and ignore the `gpfdist` configuration specified in the `Gpfdist` block of the file.

--color

Enable the use of color when displaying front-end log messages. When specified, GPSS colors the log level in messages that it writes to `stdout`. Color is deactivated by default.

GPSS ignores the `--color` option if you also specify `--csv-log`.

--csv-log

Write front-end log messages in CSV format. By default, GPSS writes log messages to `stdout` using spaces between fields for a more human-readable format.

--gpss-host host

The GPSS host. The default host address is `127.0.0.1`. If specified, overrides a `ListenAddress:Host` value provided in `gpsscliconfig.json`

--gpss-port port

The GPSS port number. The default port number is `5000`. If specified, overrides a `ListenAddress:Port` value provided in `gpsscliconfig.json`

--no-check-ca

Deactivate certificate verification when SSL is enabled between the GPSS client and server. By default, GPSS checks the certificate authority (CA) each time that you invoke a `gpsscli` subcommand.

-U | --username client_auth_username

The user name with which the GPSS server instance authenticates the client.

-P | --password client_auth_passwd

The password with which the GPSS server instance authenticates the client.

-l | --log-dir directory

The directory to which GPSS writes client command log files. GPSS must have write permission to the directory. GPSS creates the log directory if it does not exist.

If you do not provide this option, GPSS writes `gpsscli` client log files to the `$HOME/gpAdminLogs` directory.

--verbose
: The default behaviour of the command utility is to display information and error messages to `stdout`. When you specify the `--verbose` option, GPSS also outputs debug-level messages about the operation.

-h | --help
: Show command utility help, and then exit.

Examples

Start the GPSS job identified by the name `nsync_121118`:

```
$ gpsscli start nsync_121118
```

See Also

gpss, gpsscli list, gpsscli submit

gpsscli status

Display the status of a Greenplum Streaming Server job.

Synopsis

```
gpsscli status {<job_name> | <job_id>}
     [--config <gpsscliconfig.json>]
     [--gpss-host <host>] [--gpss-port <port>]
     [-U | --username <client_auth_user> -P | --password <client_auth_passwd>]
     [--no-check-ca] [-l | --log-dir <directory>] [--verbose]

gpsscli status {-h | --help}
```

Description

The `gpsscli status` command displays the status of a job submitted to a specific Greenplum Streaming Server (GPSS) instance. A job's status may be one of:

- *Submitted* - The job has been submitted.
- *Running* - The job is running.
- *Stopped*- The job was stopped by a user.
- *Success* - The job completed successfully.
- *Error* - The job returned an error.

at any given point in its lifecycle.

If the GPSS instance to which you want to send the request is not running on the default host (`127.0.0.1`) or the default port number (`5000`), you can specify the GPSS host and/or port via command line options.

Options

job_name | job_id
: The identifier of a previously-submitted GPSS job. You can specify a job name when you run `gpsscli submit`, or the command returns a unique job identifier.

--config gpsscliconfig.json
: The GPSS configuration file. This file includes properties that identify the `gpss` instance that services the command. When SSL encryption is enabled between the GPSS client and server, you also use this file to identify the file system location of the client SSL certificates. Refer to gpss.json for detailed information about the format of this file and the configuration properties supported.

`gpsscli` subcommands read the configuration specified in the `ListenAddress` block of the `gpsscliconfig.json` file, and ignore the `gpfdist` configuration specified in the `Gpfdist` block of the file.

--color
: Enable the use of color when displaying front-end log messages. When specified, GPSS colors the log level in messages that it writes to `stdout`. Color is deactivated by default.

  GPSS ignores the `--color` option if you also specify `--csv-log`.

--csv-log
: Write front-end log messages in CSV format. By default, GPSS writes log messages to `stdout` using spaces between fields for a more human-readable format.

--gpss-host host
: The GPSS host. The default host address is `127.0.0.1`. If specified, overrides a `ListenAddress:Host` value provided in `gpsscliconfig.json`

--gpss-port port
: The GPSS port number. The default port number is `5000`. If specified, overrides a `ListenAddress:Port` value provided in `gpsscliconfig.json`

--no-check-ca
: Deactivate certificate verification when SSL is enabled between the GPSS client and server. By default, GPSS checks the certificate authority (CA) each time that you invoke a `gpsscli` subcommand.

-U | --username client_auth_username
: The user name with which the GPSS server instance authenticates the client.

-P | --password client_auth_passwd
: The password with which the GPSS server instance authenticates the client.

-l | --log-dir directory
: The directory to which GPSS writes client command log files. GPSS must have write permission to the directory. GPSS creates the log directory if it does not exist.

If you do not provide this option, GPSS writes `gpsscli` client log files to the `$HOME/gpAdminLogs` directory.

--verbose
: The default behaviour of the command utility is to display information and error messages to `stdout`. When you specify the `--verbose` option, GPSS also outputs debug-level messages about the operation.

-h | --help
: Show command utility help, and then exit.

Examples

Show the status of the GPSS job identified by the name `nsync_121118` to complete:

```
$ gpsscli status nsync_121118
```

See Also

gpss, gpsscli list, gpsscli submit

gpsscli stop

Stop one or more Greenplum Streaming Server jobs.

Synopsis

```
gpsscli stop {<job_name> | <job_id>}
     [--all] [--quit-at-eof]
     [--config <gpsscliconfig.json>]
     [--gpss-host <host>] [--gpss-port <port>]
     [-U | --username <client_auth_user> -P | --password <client_auth_passwd>]
     [--no-check-ca] [-l | --log-dir <directory>] [--verbose]

gpsscli stop {-h | --help}
```

Description

The `gpsscli stop` command stops a job submitted to a specific Greenplum Streaming Server (GPSS) instance.

Specify the `--all` flag to the command to stop all running jobs.

When you stop a running job, GPSS writes any batched data to Greenplum Database and stops actively reading new data from the data source. The job transitions from the *Running* to the *Stopped* state.

If the `gpss` instance servicing the job is configured to not reuse external tables (`ReuseTables: false`), `gpsscli stop` also drops the external table currently associated with the job.

If the GPSS instance to which you want to send the request is not running on the default host (`127.0.0.1`) or the default port number (`5000`), you can specify the GPSS host and/or port via command line options.

Options

job_name | job_id

The identifier of a previously-submitted GPSS job. You can specify a job name when you run `gpsscli submit`, or the command returns a unique job identifier.

--all

Stop all currently running jobs.

--quit-at-eof

When you specify this option, `gpsscli stop` reads data until it receives an EOF, then stops the job and exits. The default behaviour of `gpsscli stop` is to immediately write any unwritten batched data before stopping the job and exiting.

--config gpsscliconfig.json

The GPSS configuration file. This file includes properties that identify the `gpss` instance that services the command. When SSL encryption is enabled between the GPSS client and server, you also use this file to identify the file system location of the client SSL certificates. Refer to gpss.json for detailed information about the format of this file and the configuration properties supported.

`gpsscli` subcommands read the configuration specified in the `ListenAddress` block of the `gpsscliconfig.json` file, and ignore the `gpfdist` configuration specified in the `Gpfdist` block of the file.

--color

Enable the use of color when displaying front-end log messages. When specified, GPSS colors the log level in messages that it writes to `stdout`. Color is deactivated by default.

GPSS ignores the `--color` option if you also specify `--csv-log`.

--csv-log

Write front-end log messages in CSV format. By default, GPSS writes log messages to `stdout` using spaces between fields for a more human-readable format.

--gpss-host host

The GPSS host. The default host address is `127.0.0.1`. If specified, overrides a `ListenAddress:Host` value provided in `gpsscliconfig.json`

--gpss-port port

The GPSS port number. The default port number is `5000`. If specified, overrides a `ListenAddress:Port` value provided in `gpsscliconfig.json`

--no-check-ca

Deactivate certificate verification when SSL is enabled between the GPSS client and server. By default, GPSS checks the certificate authority (CA) each time that you invoke a `gpsscli` subcommand.

-U | --username client_auth_username

The user name with which the GPSS server instance authenticates the client.

-P | --password client_auth_passwd

The password with which the GPSS server instance authenticates the client.

-l | --log-dir directory

The directory to which GPSS writes client command log files. GPSS must have write permission to the directory. GPSS creates the log directory if it does not exist.

If you do not provide this option, GPSS writes `gpsscli` client log files to the `$HOME/gpAdminLogs` directory.

--verbose
: The default behaviour of the command utility is to display information and error messages to `stdout`. When you specify the `--verbose` option, GPSS also outputs debug-level messages about the operation.

-h | --help
: Show command utility help, and then exit.

Examples

Stop the GPSS job identified by the name `nsync_121118`:

```
$ gpsscli stop nsync_121118
```

See Also

gpss, gpsscli list, gpsscli submit

gpsscli submit

Submit one or more jobs to a Greenplum Streaming Server.

Synopsis

```
gpsscli submit <jobconfig.yaml> [...]
     [--name <job_name>]
     [-f | --force]
     [-p | --property <template_var=value>]
     [--config <gpsscliconfig.json>]
     [--gpss-host <host>] [--gpss-port <port>]
     [-U | --username <client_auth_user> -P | --password <client_auth_passwd>]
     [--no-check-ca] [-l | --log-dir <directory>] [--verbose]

gpsscli submit {-h | --help}
```

Description

The `gpsscli submit` command submits a load job to a specific Greenplum Streaming Server (GPSS) instance. When you run the command, you provide one or more YAML-formatted configuration files that define the job parameters.

When you specify a single load configuration file, you may choose a name to identify the job. If you do not provide a name, GPSS uses the base name of the load configuration file as the job identifier. For example, if you invoke this command with the load configuration file `/dir/jobconfig.yaml` and do not provide the `--name` option, GPSS assigns the job the identifier `jobconfig`.

When you submit a job, GPSS registers the job in its job list. A job is in the *Submitted* state after it is submitted.

If the GPSS instance to which you want to send the request is not running on the default host (`127.0.0.1`) or the default port number (`5000`), you can specify the GPSS host and/or port via command line options.

Options

jobconfig.yaml [...]

One or more YAML-formatted configuration files that define the parameters of the job(s). If a filename provided is not an absolute path, Greenplum Database assumes the file system location is relative to the current working directory.

GPSS uses the properties in a YAML configuration file to uniquely identify a load operation. Submit a configuration file only once. If you submit the same configuration file more than once, GPSS will create the job, but it will eventually error out.

--name job_name

Use job_name to identify the job. If you do not provide a name, the default job identifier is the base name of the load configuration file. Job names must be unique.

GPSS does not support specifying a job_name when you provide more than one jobconfig.yaml load configuration file to the command.

-f | --force

Force GPSS to reload the configuration of a job. GPSS updates the job with the configuration specified in `jobconfig.yaml`. When the configuration reload completes, the job transitions to the *Stopped* state. If you previously named the job, you must provide `--name job\_name` when you force job configuration reload with this option.

Do not attempt to update a configuration property that GPSS uses to uniquely identify a job. If you change any such configuration property, GPSS creates a new internal job and loads all available messages.

-p | --property template_var=value

Substitute value for instances of the property value template {{template_var}} referenced in the jobconfig.yaml load configuration file.

--config gpsscliconfig.json

The GPSS configuration file. This file includes properties that identify the `gpss` instance that services the command. When SSL encryption is enabled between the GPSS client and server, you

also use this file to identify the file system location of the client SSL certificates. Refer to gpss.json for detailed information about the format of this file and the configuration properties supported.

`gpsscli` subcommands read the configuration specified in the `ListenAddress` block of the `gpsscliconfig.json` file, and ignore the `gpfdist` configuration specified in the `Gpfdist` block of the file.

--color

Enable the use of color when displaying front-end log messages. When specified, GPSS colors the log level in messages that it writes to `stdout`. Color is deactivated by default.

GPSS ignores the `--color` option if you also specify `--csv-log`.

--csv-log

Write front-end log messages in CSV format. By default, GPSS writes log messages to `stdout` using spaces between fields for a more human-readable format.

--gpss-host host

The GPSS host. The default host address is `127.0.0.1`. If specified, overrides a `ListenAddress:Host` value provided in `gpsscliconfig.json`

--gpss-port port

The GPSS port number. The default port number is `5000`. If specified, overrides a `ListenAddress:Port` value provided in `gpsscliconfig.json`

--no-check-ca

Deactivate certificate verification when SSL is enabled between the GPSS client and server. By default, GPSS checks the certificate authority (CA) each time that you invoke a `gpsscli` subcommand.

-U | --username client_auth_username

The user name with which the GPSS server instance authenticates the client.

-P | --password client_auth_passwd

The password with which the GPSS server instance authenticates the client.

-l | --log-dir directory

The directory to which GPSS writes client command log files. GPSS must have write permission to the directory. GPSS creates the log directory if it does not exist.

If you do not provide this option, GPSS writes `gpsscli` client log files to the `$HOME/gpAdminLogs` directory.

--verbose

The default behaviour of the command utility is to display information and error messages to `stdout`. When you specify the `--verbose` option, GPSS also outputs debug-level messages about the operation.

-h | --help

Show command utility help, and then exit.

Examples

Submit a GPSS job named `nsync_121118` whose load parameters are defined by the configuration file named `loadcfg.yaml`:

```
$ gpsscli submit --name nsync_121118 loadcfg.yaml
```

See Also

gpss, gpsscli.yaml

gpsscli wait

Wait for a Greenplum Streaming Server job to finish.

Synopsis

```
gpsscli wait {<job_name> | <job_id>}
     [--config <gpsscliconfig.json>]
     [--gpss-host <host>] [--gpss-port <port>]
     [-U | --username <client_auth_user> -P | --password <client_auth_passwd>]
     [--no-check-ca] [-l | --log-dir <directory>] [--verbose]

gpsscli wait {-h | --help}
```

Description

The `gpsscli wait` command waits for job submitted to a specific Greenplum Streaming Server (GPSS) instance to complete. The command blocks until the job transitions to the *Success* or *Error* state.

If the GPSS instance to which you want to send the request is not running on the default host (`127.0.0.1`) or the default port number (`5000`), you can specify the GPSS host and/or port via command line options.

Options

job_name | job_id

: The identifier of a previously-submitted GPSS job. You can specify a job name when you run `gpsscli submit`, or the command returns a unique job identifier.

--config gpsscliconfig.json

: The GPSS configuration file. This file includes properties that identify the `gpss` instance that services the command. When SSL encryption is enabled between the GPSS client and server, you also use this file to identify the file system location of the client SSL certificates. Refer to gpss.json for detailed information about the format of this file and the configuration properties supported.

`gpsscli` subcommands read the configuration specified in the `ListenAddress` block of the `gpsscliconfig.json` file, and ignore the `gpfdist` configuration specified in the `Gpfdist` block of the file.

--color

Enable the use of color when displaying front-end log messages. When specified, GPSS colors the log level in messages that it writes to `stdout`. Color is deactivated by default.

GPSS ignores the `--color` option if you also specify `--csv-log`.

--csv-log
: Write front-end log messages in CSV format. By default, GPSS writes log messages to `stdout` using spaces between fields for a more human-readable format.

--gpss-host host
: The GPSS host. The default host address is `127.0.0.1`. If specified, overrides a `ListenAddress:Host` value provided in `gpsscliconfig.json`

--gpss-port port
: The GPSS port number. The default port number is `5000`. If specified, overrides a `ListenAddress:Port` value provided in `gpsscliconfig.json`

--no-check-ca
: Deactivate certificate verification when SSL is enabled between the GPSS client and server. By default, GPSS checks the certificate authority (CA) each time that you invoke a `gpsscli` subcommand.

-U | --username client_auth_username
: The user name with which the GPSS server instance authenticates the client.

-P | --password client_auth_passwd
: The password with which the GPSS server instance authenticates the client.

-l | --log-dir directory
: The directory to which GPSS writes client command log files. GPSS must have write permission to the directory. GPSS creates the log directory if it does not exist.

If you do not provide this option, GPSS writes `gpsscli` client log files to the `$HOME/gpAdminLogs` directory.

--verbose
: The default behaviour of the command utility is to display information and error messages to `stdout`. When you specify the `--verbose` option, GPSS also outputs debug-level messages about the operation.

-h | --help
: Show command utility help, and then exit.

Examples

Wait for the GPSS job identified by the name `nsync_121118` to complete:

```
$ gpsscli wait nsync_121118
```

See Also

gpss, gpsscli list, gpsscli submit

gpsscli.yaml

title: gpsscli.yaml

gpsscli configuration file.

Synopsis

```
DATABASE: <db_name>
USER: <user_name>
PASSWORD: <password>
HOST: <coordinator_host>
PORT: <greenplum_port>
VERSION: <version_number>

<DATASOURCE>
  <DATASOURCE_specific_properties>

[SCHEDULE:
   RETRY_INTERVAL: <retry_time>
   MAX_RETRIES: <num_retries>
   RUNNING_DURATION: <run_time>
   AUTO_STOP_RESTART_INTERVAL: <restart_time>
   MAX_RESTART_TIMES: <num_restarts>
   QUIT_AT_EOF_AFTER: <clock_time>]

[ALERT:
   COMMAND: <command_to_run>
   WORKDIR: <directory>
   TIMEOUT: <alert_time>]
```

Where you may specify any property value with a template variable that GPSS substitutes at runtime using the following syntax:

```
<PROPERTY:> {{<template_var>}}
```

Description

You specify the configuration parameters for a Greenplum Streaming Server (GPSS) job in a YAML-formatted configuration file that you provide to the `gpsscli submit` command. There are two types of configuration parameters in this file - Greenplum Database connection parameters, and parameters specific to the data source from which you will load data into Greenplum.

This reference page uses the name `gpsscli.yaml` to refer to this file; you may choose your own name for the file.

GPSS currently supports loading data from Kafka and file data sources. Refer to Loading Kafka Data into Greenplum and Loading File Data into Greenplum for detailed information about using GPSS to load data into Greenplum Database.

The `gpsscli` utility processes the YAML configuration file in order, using indentation (spaces) to determine the document hierarchy and the relationships between the sections. The use of white space in the file is significant, and keywords are case-sensitive.

Keywords and Values

Greenplum Database Options

DATABASE: db_name
: The name of the Greenplum database.

USER: user_name
: The name of the Greenplum Database user/role. This user_name must have permissions as described in Configuring Greenplum Database Role Privileges.

PASSWORD: password
: The password for the Greenplum Database user/role. By default, the GPSS client passes the password to the GPSS server in clear text. When the password has a `SHADOW:` prefix, it represents a shadowed password string, and GPSS uses the `Shadow:Key` specified in its gpss.json configuration file, or a default key, to decode the password.

HOST: coordinator_host
: The host name or IP address of the Greenplum Database coordinator host.

PORT: greenplum_port
: The port number of the Greenplum Database server on the coordinator host.

VERSION: version_number
: The version of the `gpsscli` configuration file. GPSS supports versions 1 and 2 of this format.

DATASOURCE: Options

DATASOURCE

: The data source. GPSS currently supports `KAFKA` and `FILE` data sources; refer to gpkafka-v2.yaml and filesource-v2.yaml for configuration file format and parameters.

  DATASOURCE_specific_parameters
  : Parameters specific to the datasource.

Job SCHEDULE: Options

SCHEDULE:

: Controls the frequency and interval of restarting jobs.

  RETRY_INTERVAL: retry_time
  : The period of time that GPSS waits before retrying a failed job. You can specify the time interval in day (`d`), hour (`h`), minute (`m`), second (`s`), or millisecond (`ms`) integer units; do not mix units. The default retry interval is `5m` (5 minutes).

  MAX_RETRIES: num_retries
  : The maximum number of times that GPSS attempts to retry a failed job. The default is 0, do not retry. If you specify a negative value, GPSS retries the job indefinitely.

  RUNNING_DURATION: run_time
  : The amount of time after which GPSS automatically stops a job. GPSS does not automatically stop a job by default.

  AUTO_STOP_RESTART_INTERVAL: restart_time
  : The amount of time after which GPSS restarts a job that it stopped due to reaching `RUNNING_DURATION`.

  MAX_RESTART_TIMES: num_restarts

The maximum number of times that GPSS restarts a job that it stopped due to reaching `RUNNING_DURATION`. The default is 0, do not restart the job.

QUIT_AT_EOF_AFTER: clock_time

The clock time after which GPSS stops a job every day when it encounters an EOF. By default, GPSS does not automatically stop a job that reaches EOF. GPSS never stops a job when the current time is before `clock_time`, even when GPSS encounters an EOF.

Job ALERT: Options

Controls notification when a job is stopped for any reason (success, completion, error, user-initiated stop).

COMMAND: command_to_run

The program that the GPSS server runs on the GPSS server host, including arguments. The command must be executable by GPSS.

command_to_run has access to job-related environment variables that GPSS sets, including: `$GPSSJOB_NAME`, `$GPSSJOB_STATUS`, and `$GPSSJOB_DETAIL`.

WORKDIR: directory

The working directory for command_to_run. The default working directory is the directory from which you started the GPSS server process. If you specify a relative path, it is relative to the directory from which you started the GPSS server process.

TIMEOUT: alert_time

The amount of time after a job stops, prompting GPSS to trigger the alert (and run command_to_run). You can specify the time interval in day (`d`), hour (`h`), minute (`m`), or second (`s`) integer units; do not mix units. The default alert timeout is `-1s` (no timeout).

Template Variables

GPSS supports using template variables to specify property values in the load configuration file.

You specify a template variable value in the load configuration file as follows:

```
<PROPERTY>: {{<template_var>}}
```

For example:

```
MAX_RETRIES: {{numretries}}
```

GPSS substitutes the template variable with a value that you specify via the `-p | --property <template_var=value>` option to the `gpsscli dryrun`, `gpsscli submit`, `gpsscli load`, or `gpkafka load` command.

For example, if the command line specifies:

```
--property numretries=10
```

GPSS substitutes occurrences of `{{numretries}}` in the load configuration file with the value `10` before submitting the job, and uses that value while the job is running.

Examples

Submit a job to load data into Greenplum Database as defined in the load configuration file named `loadit.yaml`:

```
$ gpsscli submit loadit.yaml
```

Example Greenplum Database configuration parameters in `loadit.yaml`:

```
DATABASE: ops
USER: gpadmin
PASSWORD: changeme
HOST: mdw-1
PORT: 15432
<DATASOURCE_block> ...
```

See Also

gpsscli load, gpsscli submit, gpkafka load, filesource-v2.yaml, gpkafka-v2.yaml

gpsscli-v3.yaml (Beta)

GPSS load configuration file (version 3).

Synopsis

```
version: v3
```

```
targets:
- gpdb:
    host: <host>
    port: <greenplum_port>
    user: <user_name>
    password: <password>
    database: <db_name>
    work_schema: <work_schema_name>
    error_limit: <num_errors> | <percentage_errors>
    filter_expression: <filter_string>
    tables:
      - table: <table_name>
        schema: <schema_name>
        mode:
          # specify a single mode property block (described below)
          insert: {}
          update:
            <mode_specific_property>: <value>
            ...
          merge:
            <mode_specific_property>: <value>
            ...
        mapping:
          <target_column_name> : <source_column_name> | <expression>
          ...
        filter: <output_filter_string>
      ...
```

```
sources:
- <DATASOURCE>:
    <DATASOURCE_specific_properties>
    content:
      <data_format>:
        <column_spec>
        <other_props>
```

```
option:
  schedule:
    max_retries: <num_retries>
    retry_interval: <retry_time>
    running_duration: <run_time>
    auto_stop_restart_interval: <restart_time>
    max_restart_times: <num_restarts>
    quit_at_eof_after: <clock_time>
  alert:
    command: <command_to_run>
    workdir: <directory>
    timeout: <alert_time>
```

Where the mode_specific_propertys that you can specify for `update` and `merge` mode follow:

```
update:
  match_columns: [<match_column_names>]
  order_columns: [<order_column_names>]
  update_columns: [<update_column_names>]
  update_condition: <update_condition>
```

```
merge:
  match_columns: [<match_column_names>]
  update_columns: [<update_column_names>]
  order_columns: [<order_column_names>]
  update_condition: <update_condition>
  delete_condition: <delete_condition>
```

Where data_format, column_spec, and other_props are one of the following blocks (data source-specific):

```
avro:
  source_column_name: <column_name>
  schema_url: <http://schemareg_host:schemareg_port> %, ...%
  bytes_to_base64: <boolean>
```

```
binary:
  source_column_name: <column_name>
```

```
csv:
  columns:
    - name: <column_name>
      type: <column_data_type>
    ...
  delimiter: <delim_char>
  quote: <quote_char>
  null_string: <nullstr_val>
```

```
  escape: <escape_char>
  force_not_null: <columns>
  fill_missing_fields: <boolean>
```

```
custom:
  columns:
    - name: <column_name>
      type: <column_data_type>
    ...
  name: <formatter_name>
  options:
    - <optname>=<optvalue>
    ...
```

```
delimited:
  columns:
    - name: <column_name>
      type: <column_data_type>
    ...
  delimiter: <delimiter_string>
  eol_prefix: <prefix_string>
  quote: <quote_char>
  escape: <escape_char>
```

```
json:
  column:
    name: <column_name>
    type: json | jsonb
  is_jsonl: <boolean>
  newline: <newline_str>
```

And where you may specify any property value with a template variable that GPSS substitutes at runtime using the following syntax:

```
<property:> {{<template_var>}}
```

Description

Version 3 of the GPSS load configuration file is different in both content and format than previous versions of the file. Certain symbols used in the GPSS version 1 and 2 configuration file reference page syntax have different meanings in version 3 syntax:

- Brackets `[]` are literal and are used to specify a list in version 3. They are no longer used to signify the optionality of a property.
- Curly braces `{}` are literal and are used to specify YAML mappings in version 3 syntax. They are no longer used with the pipe symbol (`|`) to identify a list of choices.

You specify the configuration properties for a Greenplum Streaming Server (GPSS) job in a YAML-formatted configuration file that you provide to the `gpsscli submit` or `gpsscli load` command. There are three

types of configuration information in this file - target Greenplum Database connection and data import properties, properties specific to the data source from which you will load data into Greenplum, and properties specific to the GPSS job.

This reference page uses the name `gpsscli-v3.yaml` to refer to this file; you may choose your own name for the file.

The `gpsscli` utility processes the YAML configuration file in order, using indentation (spaces) to determine the document hierarchy and the relationships between the sections. The use of white space in the file is significant. Keywords are not case-sensitive.

You can use the `gpsscli convert` command to convert a V2 load configuration file to V3 syntax.

Keywords and Values

version Property

version: v3
: The version of the configuration file. You must specify `version: v3`.

targets:gpdb Properties

host: host
: The host name or IP address of the Greenplum Database coordinator host.

port: greenplum_port
: The port number of the Greenplum Database server on the coordinator host.

user: user_name
: The name of the Greenplum Database user/role. This user_name must have permissions as described in the Configuring Greenplum Database Role Privileges.

password: password
: The password for the Greenplum Database user/role.

database: db_name
: The name of the Greenplum database.

work_schema: work_schema_name
: The name of the Greenplum Database schema in which GPSS creates internal tables. The default work_schema_name is `public`.

error_limit: num_errors | percentage_errors
: The error threshold, specified as either an absolute number or a percentage. GPSS stops running the job when this limit is reached.

filter_expression: filter_string
: The filter to apply to the input data before GPSS loads the data into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. If the filter evaluates to `false`, the message is dropped. filter_string must be a valid SQL conditional expression and may reference one or more source value, key, or meta column names.

tables:
: The Greenplum Database tables, and the data that GPSS will load into each.

  table: table_name
  : The name of the Greenplum Database table into which GPSS loads the data.

  schema: schema_name

The name of the Greenplum Database schema in which table_name resides. Optional, the default schema is the `public` schema.

mode:

The table load mode; `insert`, `merge`, or `update`. The default mode is `insert`.

`update` and `merge` are not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes (" ") to identify the column.

insert:

Inserts source data into Greenplum.

update:

Updates the target table columns that are listed in `update_columns` when the input columns identified in `match_columns` match the named target table columns and the optional `update_condition` is true.

merge:

Inserts new rows and updates existing rows when:

- columns are listed in `update_columns`,
- the `match_columns` target table column values are equal to the input data, and
- an optional `update_condition` is specified and met. Deletes rows when:
- the `match_columns` target table column values are equal to the input data, and
- an optional `delete_condition` is specified and met.

New rows are identified when the `match_columns` value in the source data does not have a corresponding value in the existing data of the target table. In those cases, the entire row from the source file is inserted, not only the `match_columns` and `update_columns`. If there are multiple new `match_columns` values in the input data that are the same, GPSS inserts or updates the target table using a random matching input row. When you specify `order_columns`, GPSS sorts the input data on the specified column(s) and inserts or updates from the input row with the largest value.

mode_property_name: value

The name to value mapping for a mode property. Each `mode` supports one or more of the following properties as specified in the Synopsis.

match_columns: [match_column_names]

A comma-separated list that specifies the column(s) to use as the join condition for the update. The attribute value in the specified target column(s) must be equal to that of the corresponding source data column(s) in order for the row to be updated in the target table. Required when `mode` is `merge` or `update`.

order_columns: [order_column_names]

A comma-separated list that specifies the column(s) by which GPSS sorts the rows. When multiple matching rows exist in a batch, `order_columns` is used with `match_columns` to determine the input row with the largest value; GPSS uses that row to write/update the target. Optional. May be specified in `merge mode` to sort the input data rows.

update_columns: [update_column_names]

A column-sparated list that specifies the column(s) to update for the rows that meet the `match_columns` criteria and the optional `update_condition`.

Required when `mode` is `merge` or `update`.

update_condition: update_condition

Specifies a boolean condition, similar to that which you would declare in a `WHERE` clause, that must be met in order for a row in the target table to be updated (or inserted, in the case of a `merge`). Optional.

delete_condition: delete_condition

In `merge mode`, specifies a boolean condition, similar to that which you would declare in a `WHERE` clause, that must be met for GPSS to delete rows in the target table that meet the `match_columns` criteria. Optional.

mapping:

Optional. Overrides the default source-to-target column mapping.

When you specify a `mapping`, ensure that you provide a mapping for all source data elements of interest. GPSS does not automatically match column names when you provide a `mapping` block.

```
target\_column\_name: source\_column\_name \| expression
:   target\_column\_name specifies the target Greenplum Database table column nam
e. GPSS maps this column name to the source column name specified in source\_colu
mn\_name, or to an expression. When you specify an expression, you may provide a
value expression that you would specify in the `SELECT` list of a query, such as
a constant value, a column reference, an operator invocation, a built-in or user-
defined function call, and so on.
```

filter: output_filter_string

The filter to apply to the output data before GPSS loads the data into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. If the filter evaluates to `false`, the message is dropped. output_filter_string must be a valid SQL conditional expression and may reference one or more `META` or `VALUE` column names.

sources: Properties

sources:

The data source.

DATASOURCE

GPSS currently supports `file`, `kafka`, `rabbitmq`, and `s3` data sources.

DATASOURCE_specific_properties

Configuration properties specific to the `file`, `kafka`, `rabbitmq`, or `s3` data source; refer to filesource-v3.yaml (Beta), gpkafka-v3.yaml (Beta), rabbitmq-v3.yaml (Beta), and s3source-v3.yaml (Beta) for version 3 configuration file format and properties for these sources.

option: Properties

schedule:

Controls the frequency and interval of restarting jobs.

retry_interval: retry_time
: The period of time that GPSS waits before retrying a failed job. You can specify the time interval in day (`d`), hour (`h`), minute (`m`), second (`s`), or millisecond (`ms`) integer units; do not mix units. The default retry interval is `5m` (5 minutes).

max_retries: num_retries
: The maximum number of times that GPSS attempts to retry a failed job. The default is 0, do not retry. If you specify a negative value, GPSS retries the job indefinitely.

running_duration: run_time
: The amount of time after which GPSS automatically stops a job. GPSS does not automatically stop a job by default.

auto_stop_restart_interval: restart_time
: The amount of time after which GPSS restarts a job that it stopped due to reaching `running_duration`.

max_restart_times: num_restarts
: The maximum number of times that GPSS restarts a job that it stopped due to reaching `running_duration`. The default is 0, do not restart the job.

quit_at_eof_after: clock_time
: The clock time after which GPSS stops a job every day when it encounters an EOF. By default, GPSS does not automatically stop a job that reaches EOF. GPSS never stops a job when the current time is before `clock_time`, even when GPSS encounters an EOF.

alert:

Controls notification when a job is stopped for any reason (success, completion, error, user-initiated stop).

command: command_to_run
: The program that the GPSS server runs on the GPSS server host, including arguments. The command must be executable by GPSS.
command_to_run has access to job-related environment variables that GPSS sets, including: `$GPSSJOB_NAME`, `$GPSSJOB_STATUS`, and `$GPSSJOB_DETAIL`.

workdir: directory
: The working directory for command_to_run. The default working directory is the directory from which you started the GPSS server process. If you specify a relative path, it is relative to the directory from which you started the GPSS server process.

timeout: alert_time
: The amount of time after a job stops, prompting GPSS to trigger the alert (and run command_to_run). You can specify the time interval in day (`d`), hour (`h`), minute (`m`), or second (`s`) integer units; do not mix units. The default alert timeout is `-1s` (no timeout).

Template Variables

GPSS supports using template variables to specify property values in the load configuration file.

You specify a template variable value in the load configuration file as follows:

```
<property>: {{<template_var>}}
```

For example:

```
max_retries: {{numretries}}
```

GPSS substitutes the template variable with a value that you specify via the `-p | --property <template_var=value>` option to the `gpsscli dryrun`, `gpsscli submit`, or `gpsscli load` command.

For example, if the command line specifies:

```
--property numretries=10
```

GPSS substitutes occurrences of `{{numretries}}` in the load configuration file with the value `10` before submitting the job, and uses that value while the job is running.

Notes

If you created a database object name using a double-quoted identifier (delimited identifier), you must specify the delimited name within single quotes in the load configuration file. For example, if you create a table as follows:

```
CREATE TABLE "MyTable" (c1 text);
```

Your YAML configuration file would refer to the table name as:

```
targets:
- gpdb:
    tables:
      - table: '"MyTable"'
```

You can specify backslash escape sequences in the CSV `delimiter`, `quote`, and `escape` options. GPSS supports the standard backslash escape sequences for backspace, form feed, newline, carriage return, and tab, as well as escape sequences that you specify in hexadecimal format (prefaced with `\x`). Refer to Backslash Escape Sequences in the PostgreSQL documentation for more information.

Examples

Submit a job to load data into Greenplum Database as defined in the v3 load configuration file named `loadit_v3.yaml`:

```
$ gpsscli submit loadit_v3.yaml
```

Example Greenplum Database configuration properties in `loadit_v3.yaml`:

```
version: v3
targets:
- gpdb:
    host: mdw-1
    port: 5432
    user: gpadmin
    password: changeme
    database: testdb
    work_schema: public
```

```
    error_limit: "25"
    tables:
      - table: orders
        schema: public
        mode:
          insert {}

sources:
- kafka:
    <kafka_specific_properties>
```

See Also

gpsscli convert, gpsscli submit, filesource-v3.yaml (Beta), gpkafka-v3.yaml (Beta), rabbitmq-v3.yaml (Beta), s3source-v3.yaml (Beta)

gpkafka

Command utility for loading Kafka data into Greenplum.

Synopsis

```
gpkafka <subcommand> [<options>]

gpkafka load

gpkafka {help | -h | --help}

gpkafka --version
```

Description

`gpkafka` is a wrapper around the Greenplum Streaming Server (GPSS) `gpss` and `gpsscli` utilities. If you want to use encryption, you must explicitly start a Greenplum Streaming Server instance with the `gpss` command, and use the `gpsscli` subcommands, not `gpkafka`, to submit and manage the load job.

VMware recommends that you migrate to using the GPSS utilities directly.

The Greenplum Streaming Server includes the `gpkafka` command utility. `gpkafka` provides a subcommand to load Kafka data into Greenplum Database:

- `gpkafka load` - load data from a single Kafka topic into a Greenplum Database table
- `gpkafka help` - display command help

Options

--color

Enable the use of color when displaying front-end log messages. When specified, GPSS colors the log level in messages that it writes to `stdout`. Color is deactivated by default.

GPSS ignores the `--color` option if you also specify `--csv-log`.

--csv-log

Write front-end log messages in CSV format. By default, GPSS writes log messages to `stdout` using spaces between fields for a more human-readable format.

-l | --log-dir directory

Specify the directory to which GPSS writes client command log files. `gpkafka` must have write permission to the directory. GPSS creates the log directory if it does not exist.

If you do not provide this option, GPSS writes client log files to the `$HOME/gpAdminLogs` directory.

--verbose

The default behaviour of the command utility is to display information and error messages to `stdout`. When you specify the `--verbose` option, GPSS also outputs debug-level messages about the operation.

-h | --help

Show command utility help, and then exit.

--version

Show the version of `gpkafka`, and then exit.

See Also

gpkafka load, gpss, gpsscli

gpkafka load

Load data from Kafka into Greenplum Database.

Synopsis

```
gpkafka load <jobconfig.yaml>
    [--name <job_name>]
    [-f | --force] [--quit-at-eof] [--partition]
    [{--force-reset-earliest | --force-reset-latest | --force-reset-timestamp <tstamp
>}]
    [-p | --property <template_var=value>]
    [--config <gpfdistconfig.json>]
    [--gpfdist-host <hostaddr>] [--gpfdist-port <portnum>]
    [--debug-port <portnum> ]
    [--color] [--csv-log]
    [-l | --log-dir <directory>] [--verbose]
gpkafka load {-h | --help}
```

Description

`gpkafka load` is a wrapper around the Greenplum Streaming Server (GPSS) `gpss` and `gpsscli` utilities. Starting in Greenplum Streaming Server version 1.3.2, `gpkafka load` no

longer launches a `gpss` server instance, but rather calls the backend server code directly.

When you run `gpkafka load`, the command submits, starts, and stops a GPSS job on your behalf.

VMware recommends that you migrate to using the GPSS utilities directly.

The `gpkafka load` utility loads data from a Kafka topic into a Greenplum Database table. When you run the command, you provide a YAML-formatted configuration file that defines load parameters such as the Greenplum Database connection options, the Kafka broker and topic, and the target Greenplum Database table.

`gpkafka load` uses the `gpfdist` or `gpfdists` protocol to load data into Greenplum. You can configure the protocol options by providing a JSON-formatted GPSS configuration file via the `--config gpfdistconfig.json` option to the command, or by specifying the `--gpfdist-host hostaddr` and/or `--gpfdist-port portnum` options.

By default, `gpkafka load` loads all Kafka messages published to the topic, and then waits indefinitely for new messages to load. When you provide the `--quit-at-eof` option to the command, the utility exits after it reads all published messages and writes the data to Greenplum Database.

If you provide the `--debug-port` option, `gpkafka load` displays debug information to `stdout` during the load operation and starts a debug server from which you can obtain additional debug information.

In the case of user interrupt or exit, `gpkafka load` resumes a load operation specifying the same Kafka topic and Greenplum Database table, target schema, and database names from the last recorded offset. If GPSS detects an offset mismatch, you can choose to resume a load operation from the earliest available offset for the topic. Or, you may choose to load only new messages published to the topic, or messages published since a specific time.

Options

jobconfig.yaml

The Version 1 (deprecated), Version 2, or Version 3 (Beta) YAML-formatted configuration file that defines the load operation parameters. If the filename provided is not an absolute path, Greenplum Database assumes the file system location is relative to the current working directory. Refer to gpkafka.yaml and gpkafka-v2.yaml for the format and content of the parameters that you specify in Versions 1 and 2 of this file. Refer to gpkafka-v3.yaml (Beta) for Version 3 format information.

--name job_name

Use job_name to identify the job. If you do not provide a name, the command assigns a unique identifier to the job.

-f | --force

Force GPSS to reload the configuration of a running job. GPSS stops the job, updates the job with the configuration specified in jobconfig.yaml, and then restarts the job. If you previously named the job, you must provide `--name job\_name` when you force job configuration reload with this option.

Do not attempt to update a configuration property that GPSS uses to uniquely identify a Kafka job (the Kafka topic name and the Greenplum database, schema,

and table names). If you change any such configuration property, GPSS creates a new internal job and loads all available messages.

--quit-at-eof

When you specify this option, `gpkafka load` exits after it reads all of the Kafka messages published to the topic. The default behaviour of `gpkafka load` is to wait indefinitely for, and then consume, new Kafka messages published to the topic.

`gpkafka load` ignores job retry `SCHEDULE` configuration settings when it is invoked with the `--quit-at-eof` flag.

--partition

By default, `gpkafka load` outputs the job progress by batch, and displays the start and end times, the message number and size, the number of inserted and rejected rows, and the transfer speed per batch. When you specify the `--partition` option, `gpkafka load` outputs the job progress by partition, and displays the partition identifier, the start and end times, the beginning and ending offsets, the message size, and the transfer speed per partition.

--force-reset-earliest

`gpkafka load` returns an error if its recorded offset does not match the Kafka message offset for the topic. Re-run `gpkafka load` and specify the `--force-reset-earliest` option to resume the load operation from the earliest available message published to the Kafka topic.

`--force-reset-earliest` specified on the command line takes precedence over a `FALLBACK_OFFSET/fallback_offset` set in the jobconfig.yaml.

--force-reset-latest

`gpkafka load` returns an error if its recorded offset does not match the Kafka message offset for the topic. Re-run `gpkafka load` and specify the `--force-reset-latest` option to load only new data messages published to the Kafka topic.

`--force-reset-latest` specified on the command line takes precedence over a `FALLBACK_OFFSET/fallback_offset` set in the jobconfig.yaml.

--force-reset-timestamp tstamp

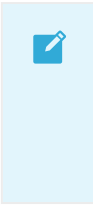
Specify the `--force-reset-timestamp` option to load Kafka messages published to the topic from the offset associated with the specified time. tstamp must specify epoch time in milliseconds, and is bounded by the earliest message time and the current time.

-p | --property template_var=value

Substitute value for instances of the property value template {{template_var}} referenced in the jobconfig.yaml load configuration file.

--config gpfdistconfig.json

The GPSS configuration file. This file includes properties that configure the `gpfdist/s` protocol used for the load request. Refer to gpss.json for detailed information about the format of this file and the configuration properties supported.



`gpkafka load` reads the configuration specified in the `Gpfdist` protocol block of the `gpfdistconfig.json` file; it ignores the GPSS configuration specified in the `ListenAddress` block of the file.

--gpfdist-host hostaddr
: The `gpfdist` service host name or IP address that GPSS sets in the external table `LOCATION` clause. If specified, overrides a `Gpfdist:Host` value provided in `gpfdistconfig.json`.

--gpfdist-port portnum
: The `gpfdist` service port number. If specified, overrides a `Gpfdist:Port` value provided in `gpfdistconfig.json`.

--debug-port portnum
: When you specify this option, `gpkafka load` starts a debug server at the port identified by portnum; additional debug information including the call stack and performance statistics is available via `curl http://gpkafkahost:portnum/debug/pprof/`.

--color
: Enable the use of color when displaying front-end log messages. When specified, GPSS colors the log level in messages that it writes to `stdout`. Color is deactivated by default.

  GPSS ignores the `--color` option if you also specify `--csv-log`.

--csv-log
: Write front-end log messages in CSV format. By default, GPSS writes log messages to `stdout` using spaces between fields for a more human-readable format.

-l | --log-dir directory
: Specify the directory to which GPSS writes client command log files. GPSS must have write permission to the directory. GPSS creates the log directory if it does not exist.

  If you do not provide this option, GPSS writes client log files to the `$HOME/gpAdminLogs` directory.

--verbose
: The default behaviour of the command utility is to display information and error messages to `stdout`. When you specify the `--verbose` option, GPSS also outputs debug-level messages about the operation.

-h | --help
: Show command utility help, and then exit.

Examples

Stream Kafka data into Greenplum Database using the load parameters defined in a configuration file named `loadcfg.yaml` located in the current directory:

```
gpkafka load loadcfg.yaml
```

Load Kafka data into Greenplum Database using a configuration file located in the current directory named `loadcfg.yaml`; exit the load operation after reading all Kafka messages published to the topic:

```
gpkafka load --quit-at-eof loadcfg.yaml
```

See Also

gpkafka.yaml, gpkafka-v2.yaml, gpss, gpss.json, gpsscli

gpkafka-v3.yaml (Beta)

GPSS load configuration file for a Kafka data source (version 3).

Synopsis

```
version: v3
```

```
targets:
- gpdb:
    host: <host>
    port: <greenplum_port>
    user: <user_name>
    password: <password>
    database: <db_name>
    work_schema: <work_schema_name>
    error_limit: <num_errors> | <percentage_errors>
    filter_expression: <filter_string>
    tables:
      - table: <table_name>
        schema: <schema_name>
        mode:
          # specify a single mode property block (described below)
          insert: {}
          update:
            <mode_specific_property>: <value>
            ...
          merge:
            <mode_specific_property>: <value>
            ...
        transformer:
          transform: <udf_transform_udf_name>
          properties:
            <udf_transform_property_name>: <property_value>
            ...
          columns:
            - <udf_transform_column_name>
            ...
        mapping:
          <target_column_name> : <source_column_name> | <expression>
          ...
        filter: <output_filter_string>
      ...
```

```
sources:
- kafka:
    topic: <kafka_topic>
    brokers: <kafka_broker_host:broker_port> %, ...%
    partitions: (<partition_numbers>)
    key_content:
      <data_format>:
```

```
        <column_spec>
        <other_props>
    value_content:
      <data_format>:
        <column_spec>
        <other_props>
    meta:
      json:
        column:
          name: meta
          type: json
    encoding: <char_set>
    transformer:
      path: <path_to_plugin_transform_library>
      on_init: <plugin_transform_init_name>
      transform: <plugin_transform_name>
      properties:
        <plugin_transform_property_name>: <property_value>
        ...
    rdkafka_prop:
      <kafka_property_name>: <kafka_property_value>
      ...
    task:
      batch_size:
        max_count: <number_of_rows>
        interval_ms: <wait_time>
        idle_duration_ms: <idle_time>
      window_size: <num_batches>
      window_statement: <udf_or_sql_to_run>
      prepare_statement: <udf_or_sql_to_run>
      teardown_statement: <udf_or_sql_to_run>
    save_failing_batch: <boolean>
    recover_failing_batch: <boolean> (Beta)
    consistency: strong | at-least | at-most | none
    fallback_offset: earliest | latest
```

```
option:
  schedule:
    max_retries: <num_retries>
    retry_interval: <retry_time>
    running_duration: <run_time>
    auto_stop_restart_interval: <restart_time>
    max_restart_times: <num_restarts>
    quit_at_eof_after: <clock_time>
  alert:
    command: <command_to_run>
    workdir: <directory>
    timeout: <alert_time>
```

Where the mode_specific_propertys that you can specify for `update` and `merge` mode follow:

```
update:
  match_columns: [<match_column_names>]
  order_columns: [<order_column_names>]
  update_columns: [<update_column_names>]
  update_condition: <update_condition>
```

```
merge:
  match_columns: [<match_column_names>]
  update_columns: [<update_column_names>]
  order_columns: [<order_column_names>]
  update_condition: <update_condition>
  delete_condition: <delete_condition>
```

Where data_format, column_spec, and other_props are one of the following blocks

```
avro:
  source_column_name: <column_name>
  schema_url: <http://schemareg_host:schemareg_port> %, ...%
  bytes_to_base64: <boolean>
```

```
  schema_ca_on_gpdb: <sr_ca_file_path>
  schema_cert_on_gpdb: <sr_cert_file_path>
  schema_key_on_gpdb: <sr_key_file_path>
  schema_min_tls_version: <minimum_version>
  schema_path_on_gpdb: <path_to_file>
```

```
binary:
  source_column_name: <column_name>
```

```
csv:
  columns:
    - name: <column_name>
      type: <column_data_type>
    ...
  delimiter: <delim_char>
  quote: <quote_char>
  null_string: <nullstr_val>
  escape: <escape_char>
  force_not_null: <columns>
  fill_missing_fields: <boolean>
```

```
custom:
  columns:
    - name: <column_name>
      type: <column_data_type>
    ...
  name: <formatter_name>
  options:
    - <optname>=<optvalue>
    ...
```

```
delimited:
  columns:
    - name: <column_name>
      type: <column_data_type>
    ...
  delimiter: <delimiter_string>
  eol_prefix: <prefix_string>
  quote: <quote_char>
  escape: <escape_char>
```

```
json:
  column:
    name: <column_name>
    type: json | jsonb
  is_jsonl: <boolean>
  newline: <newline_str>
```

And where you may specify any property value with a template variable that GPSS substitutes at runtime using the following syntax:

```
<property:> {{<template_var>}}
```

Description

Version 3 of the GPSS load configuration file is different in both content and format than previous versions of the file. Certain symbols used in the GPSS version 1 and 2 configuration file reference page syntax have different meanings in version 3 syntax:

- Brackets `[]` are literal and are used to specify a list in version 3. They are no longer used to signify the optionality of a property.
- Curly braces `{}` are literal and are used to specify YAML mappings in version 3 syntax. They are no longer used with the pipe symbol (`|`) to identify a list of choices.

You specify load configuration properties for a Greenplum Streaming Server (GPSS) Kafka load job in a YAML-formatted configuration file. (This reference page uses the name `gpkafka-v3.yaml` when referring to this file; you may choose your own name for the file.) Load properties include Greenplum Database connection and data import properties, Kafka broker, topic, and message format information, and properies specific to the GPSS job.

The `gpsscli` and `gpkafka load` utilities processes the YAML configuration file in order, using indentation (spaces) to determine the document hierarchy and the relationships between the sections. The use of white space in the file is significant. Keywords are not case-sensitive.

Keywords and Values

version Property

version: v3
: The version of the configuration file. You must specify `version: v3`.

targets:gpdb Properties

host: host
: The host name or IP address of the Greenplum Database coordinator host.

port: greenplum_port
: The port number of the Greenplum Database server on the coordinator host.

user: user_name

The name of the Greenplum Database user/role. This user_name must have permissions as described in the Configuring Greenplum Database Role Privileges.

password: password

The password for the Greenplum Database user/role.

database: db_name

The name of the Greenplum database.

work_schema: work_schema_name

The name of the Greenplum Database schema in which GPSS creates internal tables. The default work_schema_name is `public`.

error_limit: num_errors | percentage_errors

The error threshold, specified as either an absolute number or a percentage. GPSS stops running the job when this limit is reached.

filter_expression: filter_string

The filter to apply to the input data before GPSS loads the data into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. If the filter evaluates to `false`, the message is dropped. filter_string must be a valid SQL conditional expression and may reference one or more source value, key, or meta column names.

tables:

The Greenplum Database tables, and the data that GPSS will load into each.

table: table_name

The name of the Greenplum Database table into which GPSS loads the data.

schema: schema_name

The name of the Greenplum Database schema in which table_name resides. Optional, the default schema is the `public` schema.

mode:

The table load mode; `insert`, `merge`, or `update`. The default mode is `insert`.

`update` and `merge` are not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes (" ") to identify the column.

insert:

Inserts source data into Greenplum.

update:

Updates the target table columns that are listed in `update_columns` when the input columns identified in `match_columns` match the named target table columns and the optional `update_condition` is true.

merge:

Inserts new rows and updates existing rows when:

- columns are listed in `update_columns`,
- the `match_columns` target table column values are equal to the input data, and
- an optional `update_condition` is specified and met.

Deletes rows when:

- the `match_columns` target table column values are equal to the input data, and
- an optional `delete_condition` is specified and met.

New rows are identified when the `match_columns` value in the source data does not have a corresponding value in the existing data of the target table. In those cases, the entire row from the source file is inserted, not only the `match_columns` and `update_columns`. If there are multiple new `match_columns` values in the input data that are the same, GPSS inserts or updates the target table using a random matching input row. When you specify `order_columns`, GPSS sorts the input data on the specified column(s) and inserts or updates from the input row with the largest value.

mode_property_name: value
: The name to value mapping for a mode property. Each `mode` supports one or more of the following properties as specified in the Synopsis.

match_columns: [match_column_names]
: A comma-separated list that specifies the column(s) to use as the join condition for the update. The attribute value in the specified target column(s) must be equal to that of the corresponding source data column(s) in order for the row to be updated in the target table. Required when `mode` is `merge` or `update`.

order_columns: [order_column_names]
: A comma-separated list that specifies the column(s) by which GPSS sorts the rows. When multiple matching rows exist in a batch, `order_columns` is used with `match_columns` to determine the input row with the largest value; GPSS uses that row to write/update the target. Optional. May be specified in `merge mode` to sort the input data rows.

update_columns: [update_column_names]
: A column-sparated list that specifies the column(s) to update for the rows that meet the `match_columns` criteria and the optional `update_condition`.
Required when `mode` is `merge` or `update`.

update_condition: update_condition
: Specifies a boolean condition, similar to that which you would declare in a `WHERE` clause, that must be met in order for a row in the target table to be updated (or inserted, in the case of a `merge`). Optional.

delete_condition: delete_condition
: In `merge mode`, specifies a boolean condition, similar to that which you would declare in a `WHERE` clause, that must be met for GPSS to delete rows in the target table that meet the `match_columns` criteria. Optional.

transformer:

: Optional. Output data transform block. An output data transformer is a user-defined function (UDF) that transforms the data before it is loaded into Greenplum Database. The semantics of the UDF are transform-specific.

GPSS currently supports specifying only one of the `mapping` or (UDF) `transformer` blocks in the load configuration file, not both.

transform: udf_transform_udf_name

The name of the output transform UDF. GPSS invokes this function for every batch of data it writes to Greenplum Database.

properties: udf_transform_property_name: property_value

One or more property name and value pairs that GPSS passes to udf_transform_udf_name.

columns: udf_transform_column_name

The name of one or more columns involved in the transform.

mapping:

Optional. Overrides the default source-to-target column mapping.

GPSS currently supports specifying only one of the `mapping` or (UDF) `transformer` blocks in the load configuration file, not both.

When you specify a `mapping`, ensure that you provide a mapping for all source data elements of interest. GPSS does not automatically match column names when you provide a `mapping` block.

target_column_name: source_column_name | expression

target_column_name specifies the target Greenplum Database table column name. GPSS maps this column name to the source column name specified in source_column_name, or to an expression. When you specify an expression, you may provide a value expression that you would specify in the `SELECT` list of a query, such as a constant value, a column reference, an operator invocation, a built-in or user-defined function call, and so on.

filter: output_filter_string

The filter to apply to the output data before GPSS loads the data into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. If the filter evaluates to `false`, the message is dropped. output_filter_string must be a valid SQL conditional expression and may reference one or more `META` or `VALUE` column names.

sources:kafka: Options

topic: kafka_topic

The name of the Kafka topic from which to load data. The topic must exist.

brokers: kafka_broker_host:broker_port

A host and port number for each of one or more Kafka brokers.

partitions: (partition_numbers)

A single, a comma-separated list, and/or a range of partition numbers from which GPSS reads messages from the Kafka topic. A range that you specify with the `M...N` syntax includes both the range start and end values. By default, GPSS reads messages from all partitions of the Kafka topic.

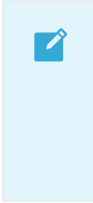
Ensure that you do not configure multiple jobs that specify overlapping partition numbers in the same topic; GPSS behavior is undefined.

key_content:

The Kafka message data type, field names, and type-specific properties. You must specify all Kafka key elements in the order in which they appear in the Kafka message. Optional when you specify a `value_content` block; GPSS ignores the Kafka message key in this circumstance.

value_content:

The Kafka message value data type, field names, and type-specific properties. You must specify all Kafka data elements in the order in which they appear in the Kafka message. Optional when you specify a `key_content` block; GPSS ignores the Kafka message value in this circumstance.



You must not provide a `value_content` block when you specify `csv` format for the `key_content` block. Similarly, you must not provide a `key_content` block when you specify `csv` format for a `value_content` block.

column_spec

The source to Greenplum column mapping. The supported column specification differs for different data formats as described below.

The default source-to-target data mapping behaviour of GPSS is to match a column name as defined in `source_column_name`, `column:name`, or `columns:name` with a column name in the target Greenplum Database `table`. You can override the default mapping by specifying a `mapping:` block.

data_format

The format of the key or value data. You may specify a data_format of `avro`, `binary`, `csv`, `custom`, `delimited`, or `json` for the key and value, with some restrictions.

avro

When you specify the `avro` data format for a key or value, GPSS reads the data into a single `json`-type column. You may specify a schema registery location and optional SSL certificates and keys, and whether or not you want GPSS to convert `bytes` fields into base64-encoded strings.

source_column_name: column_name

The name of the single `json`-type column into which GPSS reads the key or value data.

schema_url: schemareg_host:schemareg_port

When you specify the `avro` format and the Avro schema of the JSON data that you want to load is registered in the Confluent Schema Registry, you must identify the host name and port number of each Confluent Schema Registry server in your Kafka cluster. You may specify more than one address, and at least one of the addresses must be legal.

bytes_to_base64: boolean

When `true`, GPSS converts Avro `bytes` fields into base64-encoded strings. The default value is `false`, GPSS does not perform the conversion.

schema_ca_on_gpdb: sr_ca_file_path

The file system path to the CA certificate that GPSS uses to verify the peer. This file must reside in sr_ca_file_path on all Greenplum Database segment hosts.

schema_cert_on_gpdb: sr_cert_file_path

The file system path to the client certificate that GPSS uses to connect to the HTTPS schema registry. This file must reside in sr_cert_file_path on all Greenplum Database segment hosts.

schema_key_on_gpdb: sr_key_file_path

The file system path to the private key file that GPSS uses to connect to the HTTPS schema registry. This file must reside in sr_key_file_path on all Greenplum Database segment hosts.

schema_min_tls_version: minimum_version
: The minimum transport layer security (TLS) version that GPSS requests on the connection to the schema registry. Supported versions are `1.0`, `1.1`, `1.2`, or `1.3`. The default minimum TLS version is `1.0`.

schema_path_on_gpdb: path_to_file
: When you specify the `avro` format and the Avro schema of the JSON key or value data that you want to load is specified in a separate `.avsc` file, you must identify the file system location in path_to_file, and the file must reside in this location on every Greenplum Database segment host.

GPSS does not cache the schema. GPSS must reload the schema for every batch of Kafka data. Also, GPSS supports providing the schema for either the key or the value, but not both.

binary
: When you specify the `binary` data format, GPSS reads the data into a single `bytea`-type column.

source_column_name: column_name
: The name of the single `bytea`-type column into which GPSS reads the key or value data.

csv
: When you specify the `csv` data format, GPSS reads the data into the list of columns that you specify. The message content cannot contain line ending characters (CR and LF).

columns:
: A set of column name/type mappings. The value `[]` specifies all columns.

name: column_name
: The name of a key or value column. column_name must match the column name of the target Greenplum Database table.

type: column_data_type
: The data type of the column. You must specify an equivalent data type for each data element and the associated Greenplum Database table column.

delimiter: delim_char
: Specifies a single ASCII character that separates columns within each message or row of data. The default delimiter is a comma (`,`).

quote: quote_char
: Specifies the quotation character. Because GPSS does not provide a default value for this property, you must specify a value.

null_string: nullstr_val
: Specifies the string that represents the null value. Because GPSS does not provide a default value for this property, you must specify a value.

escape: escape_char
: Specifies the single character that is used for escaping data characters in the content that might otherwise be interpreted as row or column delimiters. Make sure to choose an escape character that is not used anywhere in your actual column data. Because GPSS does not provide a default value for this property, you must specify a value.

force_not_null: columns
: Specifies a comma-separated list of column names to process as though each column were quoted and hence not a NULL value. For the default `null_string` (nothing between two delimiters), missing values are evaluated as zero-length strings.

fill_missing_fields: boolean
: Specifies the action of GPSS when it reads a row of data that has missing trailing field values (the row has missing data fields at the end of a line or row). The default value is `false`, GPSS returns an error when it encounters a row with missing trailing field values.

If set to `true`, GPSS sets missing trailing field values to `NULL`. Blank rows, fields with a `NOT NULL` constraint, and trailing delimiters on a line will still generate an error.

custom
: When you specify the `custom` data format, GPSS uses the custom formatter that you specify to process the input data before writing it to Greenplum Database.

columns:
: A set of column name/type mappings. The value `[]` specifies all columns.

name: column_name
: The name of a key or value column. column_name must match the column name of the target Greenplum Database table.

type: column_data_type
: The data type of the column. You must specify an equivalent data type for each data element and the associated Greenplum Database table column.

name: formatter_name
: When you specify the `custom` data format, formatter_name is required and must identify the name of the formatter user-defined function that GPSS should use when loading the data.

options:
: A set of function argument name=value pairs.

optname=optvalue
: The name and value of the set of arguments to pass into the formatter_name UDF.

delimited
: When you specify the `delimited` data format, GPSS reads the data into the list of columns that you specify. You must specify the data `delimiter`.

columns:
: A set of column name/type mappings. The value `[]` specifies all columns.

name: column_name
: The name of a key or value column. column_name must match the column name of the target Greenplum Database table.

type: column_data_type
: The data type of the column. You must specify an equivalent data type for each data element and the associated Greenplum Database table column.

delimiter: delimiter_string
: When you specify the `delimited` data format, delimiter_string is required and must identify the data element delimiter. delimiter_string may be a multi-byte value, and up to 32 bytes in length. It may not contain quote and escape characters.

eol_prefix: prefix_string
: Specifies the prefix before the end of line character (`\n`) that indicates the end of a row. The default prefix is empty.

quote: quote_char
: Specifies the single ASCII quotation character. The default quote character is empty.

If you do not specify a quotation character, GPSS assumes that all columns are unquoted. If you do not specify a quotation character and do specify an escape character, GPSS assumes that all columns are unquoted and escapes the delimiter, end-of-line prefix, and escape itself.
When you specify a quotation character, you must specify an escape character. GPSS reads any content between quote characters as-is, except for escaped characters.

escape: escape_char

Specifies the single ASCII character used to escape special characters (for example, the `delimiter`, `eol_prefix`, `quote`, or `escape` itself). Therdefault escape character is empty.
When you specify an escape character and do not specify a quotation character, GPSS escapes only the delimiter, end-of-line prefix, and escape itself.
When you specify both an escape character and a quotation character, GPSS escapes only these characters.

json

When you specify the `json` data format, GPSS can read the data as a single JSON object or as a single JSON record per line.

column:

A single column name/type mapping.

name: column_name

The name of the key or value column. column_name must match the column name of the target Greenplum Database table.

type: json | jsonb | gp_jsonb (Beta) | gp_json (Beta)

The data type of the column.

is_jsonl: boolean

Identifies whether or not GPSS reads the JSON data as a single object or single-record-per-line. The default is `false`, GPSS reads the JSON data as a single object.

newline: newline_str

A string that specifies the new line character(s) that end each JSON record. The default newline is `"\n"`.

meta:

The data type and field name of the Kafka meta data. `meta:` must specify the `json` or `jsonb` (Greenplum 6 only) data format, and a single `json`-type column. The available Kafka meta data properties include:

- `topic` (text) - the Kafka topic name
- `partition` (int) - the partition number
- `offset` (bigint) - the record location within the partition
- `timestamp` (bigint) - the time that the message was appended to the Kafka log

You can load any of these properties into the target table with a `mapping`, or use a property in the update or merge criteria for a load operation.

encoding: char_set

The source data encoding. You can specify an encoding character set when the source data is of the `csv`, `custom`, `delimited`, or `json` format. GPSS supports the character sets identified in Character Set Support in the Tanzu Greenplum documentation.

transformer:

Input data transform block. An input data transformer is a plugin, set of `go` functions that transform the data after it is read from the source. The semantics of the transform are function-specific. You specify the library and function names in this block, as well as the properties that GPSS passes to these functions:

path: path_to_plugin_transform_library
: The file system location of the plugin transformer library on the Greenplum Streaming Server server host.

on_init: plugin_transform_init_name
: The name of an initialization function that GPSS calls when it loads the transform library.

transform: plugin_transform_name
: The name of the transform function. GPSS invokes this function for every message it reads.

properties: plugin_transform_property_name: property_value
: One or more property name and value pairs that GPSS passes to plugin_transform_init_name and plugin_transform_name.

rdkafka_prop:

Kafka consumer configuration property names and values.

kafka_property_name
: The name of a Kafka property.

kafka_property_value
: The Kafka property value.

task:

The batch size and commit window.

batch_size:
: Controls how GPSS commits data to Greenplum Database. You may specify both `max_count` and `interval_ms` as long as both values are not zero (`0`). Try setting and tuning `interval_ms` to your environment; introduce a `max_count` setting only if you encounter high memory usage associated with message buffering.

max_count: number_of_rows
: The number of rows to batch before triggering an `INSERT` operation on the Greenplum Database table. The default value of `max_count` is `0`, which instructs GPSS to ignore this commit trigger condition.

interval_ms: wait_time
: The minimum amount of time to wait (milliseconds) between each `INSERT` operation on the table. The default value is `5000`.

idle_duration_ms: idle_time
: The maximum amount of time to wait (milliseconds) for the first batch of Kafka data. When you use this property to enable lazy load, GPSS waits until Kafka data is available before locking the target Greenplum table. You can specify:

- `0` (lazy load is deactivated)
- `-1` (lazy load is activated, the job never stops), or

- a positive value (lazy load is activated, the job stops after idle_time duration of no data in the Kafka topic)

The default value is `0`.

window_size: num_batches
: The number of batches to read before running `window_statement`. The default batch interval is 0.

window_statement: udf_or_sql_to_run
: A user-defined function or SQL command(s) that you want to run after GPSS reads `window_size` number of batches. The default is null, no command to run.

prepare_statement: udf_or_sql_to_run
: A user-defined function or SQL command(s) that you want GPSS to run before it executes the job. The default is null, no command to run.

teardown_statement: udf_or_sql_to_run
: A user-defined function or SQL command(s) that you want GPSS to run after the job stops. GPSS runs the function or command(s) on job success and job failure. The default is null, no command to run.

save_failing_batch: boolean
: Determines whether or not GPSS saves data into a backup table before it writes the data to Greenplum Database. Saving the data in this manner aids recovery when GPSS encounters errors during the evaluation of expressions. The default is `false`; GPSS does not use a backup table, and returns immediately when it encounters an expression error. When you set this property to `true`, GPSS writes both the good and the bad data in the batch to a backup table named `gpssbackup_<jobhash>`, and continues to process incoming data. You must then manually load the good data from the backup table into Greenplum **or** set `recover_failing_batch` (Beta) to `true` to have GPSS automatically reload the good data.

Using a backup table to hedge against mapping errors may impact performance, especially when the data that you are loading has not been cleaned.

recover_failing_batch: boolean (Beta)
: When set to `true` and `save_failing_batch` is also `true`, GPSS automatically reloads the good data in the batch and retains only the error data in the backup table. The default value is `false`; GPSS does not process the backup table.

Enabling this property requires that GPSS has the Greenplum Database privileges to create a function.

consistency: strong | at-least | at-most | none
: Specify how GPSS should manage message offsets when it acts as a high-level Kafka consumer. Valid values are `strong`, `at-least`, `at-most`, and `none`. The default value is `strong`. Refer to Understanding Kafka Message Offset Management for more detailed information.

fallback_offset: earliest | latest

Specifies the behaviour of GPSS when it detects a Kafka message offset gap. When set to `earliest`, GPSS automatically resumes a load operation from the earliest available published message. When set to `latest`, GPSS loads only new messages to the Kafka topic. If this property is not set, GPSS returns an error.

option: Properties

schedule:

Controls the frequency and interval of restarting jobs.

retry_interval: retry_time
: The period of time that GPSS waits before retrying a failed job. You can specify the time interval in day (`d`), hour (`h`), minute (`m`), second (`s`), or millisecond (`ms`) integer units; do not mix units. The default retry interval is `5m` (5 minutes).

max_retries: num_retries
: The maximum number of times that GPSS attempts to retry a failed job. The default is 0, do not retry. If you specify a negative value, GPSS retries the job indefinitely.

running_duration: run_time
: The amount of time after which GPSS automatically stops a job. GPSS does not automatically stop a job by default.

auto_stop_restart_interval: restart_time
: The amount of time after which GPSS restarts a job that it stopped due to reaching `running_duration`.

max_restart_times: num_restarts
: The maximum number of times that GPSS restarts a job that it stopped due to reaching `running_duration`. The default is 0, do not restart the job.

quit_at_eof_after: clock_time
: The clock time after which GPSS stops a job every day when it encounters an EOF. By default, GPSS does not automatically stop a job that reaches EOF. GPSS never stops a job when the current time is before `clock_time`, even when GPSS encounters an EOF.

alert:

Controls notification when a job is stopped for any reason (success, completion, error, user-initiated stop).

command: command_to_run
: The program that the GPSS server runs on the GPSS server host, including arguments. The command must be executable by GPSS.
command_to_run has access to job-related environment variables that GPSS sets, including: `$GPSSJOB_NAME`, `$GPSSJOB_STATUS`, and `$GPSSJOB_DETAIL`.

workdir: directory
: The working directory for command_to_run. The default working directory is the directory from which you started the GPSS server process. If you specify a relative path, it is relative to the directory from which you started the GPSS server process.

timeout: alert_time
: The amount of time after a job stops, prompting GPSS to trigger the alert (and run command_to_run). You can specify the time interval in day (`d`), hour (`h`), minute (`m`), or second (`s`) integer units; do not mix units. The default alert timeout is `-1s` (no timeout).

Template Variables

GPSS supports using template variables to specify property values in the load configuration file.

You specify a template variable value in the load configuration file as follows:

```
<property>: {{<template_var>}}
```

For example:

```
max_retries: {{numretries}}
```

GPSS substitutes the template variable with a value that you specify via the `-p | --property <template_var=value>` option to the `gpsscli dryrun`, `gpsscli submit`, `gpsscli load`, or `gpkafka load` command.

For example, if the command line specifies:

```
--property numretries=10
```

GPSS substitutes occurrences of `{{numretries}}` in the load configuration file with the value `10` before submitting the job, and uses that value while the job is running.

Notes

If you created a database object name using a double-quoted identifier (delimited identifier), you must specify the delimited name within single quotes in the load configuration file. For example, if you create a table as follows:

```
CREATE TABLE "MyTable" (c1 text);
```

Your YAML configuration file would refer to the table name as:

```
targets:
- gpdb:
    tables:
      - table: '"MyTable"'
```

You can specify backslash escape sequences in the CSV `delimiter`, `quote`, and `escape` options. GPSS supports the standard backslash escape sequences for backspace, form feed, newline, carriage return, and tab, as well as escape sequences that you specify in hexadecimal format (prefaced with `\x`). Refer to Backslash Escape Sequences in the PostgreSQL documentation for more information.

Kafka Properties

GPSS requires Kafka version 0.11 or newer for exactly-once delivery assurance. You can run with an older version of Kafka (but lose the exactly-once guarantee) by adding the following `rdkafka_prop` block to your `gpkafka-v3.yaml` load configuration file:

```
rdkafka_prop:
      api.version.request: false
```

```
      broker.version.fallback: 0.8.2.1
```

Examples

Load data from Kafka as defined in the Version 3 configuration file named `loadfromkafka_v3.yaml`:

```
gpkafka load loadfromkafka_v3.yaml
```

Example `loadfromkafka_v3.yaml` configuration file:

```
version: v3
targets:
- gpdb:
    host: mdw-1
    port: 15432
    user: gpadmin
    password: changeme
    database: testdb
    work_schema: public
    error_limit: 25
    tables:
      - table: tbl_order_merge
        schema: public
        mode:
          insert {}
        mapping:
          data: (value->>'data')::text
          o: (meta->>'offset')::bigint
          p: (meta->>'partition')::int
          pk: (value->>'pk')::int
          ts: (meta->>'timestamp')::bigint

sources:
- kafka:
    topic: daily_orders
    brokers: localhost:9092
    key_content:
      binary:
        source_column_name: key
      value_content:
        json:
          column:
            name: value
            type: JSON
      meta:
        json:
          column:
            name: meta
            type: JSON
      task:
        batch_size:
          interval_ms: 5000
          max_count: 1
        window_size: 5
option:
  schedule:
    running_duration: 2s
```

```
    auto_stop_restart_interval : 2s
    max_restart_times : 1
```

See Also

gpkafka load, gpsscli submit, gpsscli load

gpkafka-v2.yaml

gpkafka configuration file (version 2).

Synopsis

```
DATABASE: <db_name>
USER: <user_name>
PASSWORD: <password>
HOST: <host>
PORT: <greenplum_port>
VERSION: 2
KAFKA:
   INPUT:
      SOURCE:
        BROKERS: <kafka_broker_host:broker_port> [, ... ]
        TOPIC: <kafka_topic>
        [PARTITIONS: (<partition_numbers>)]
        [FALLBACK_OFFSET: { earliest | latest }]
      [VALUE:
        COLUMNS:
           - NAME: { <column_name> | __IGNORED__ }
             TYPE: <column_data_type>
           [ ... ]
         FORMAT: <value_data_format>
         [[DELIMITED_OPTION:
            DELIMITER: <delimiter_string>
            [EOL_PREFIX: <prefix_string>]
            [QUOTE: <quote_char>]
            [ESCAPE: <escape_char>] ] |
         [AVRO_OPTION:
            [SCHEMA_REGISTRY_ADDR: <http://schemareg_host:schemareg_port> [, ... ]]
            [SCHEMA_CA_ON_GPDB: <sr_ca_file_path>]
            [SCHEMA_CERT_ON_GPDB: <sr_cert_file_path>]
            [SCHEMA_KEY_ON_GPDB: <sr_key_file_path>]
            [SCHEMA_MIN_TLS_VERSION: <minimum_version>]
            [SCHEMA_PATH_ON_GPDB: <path_to_file>]r
            [BYTES_TO_BASE64: <boolean>]] |
         [CSV_OPTION:
            [DELIMITER: <delim_char>]
            [QUOTE: <quote_char>]
            [NULL_STRING: <nullstr_val>]
            [ESCAPE: <escape_char>]
            [FORCE_NOT_NULL: <columns>]
            [FILL_MISSING_FIELDS: <boolean>]] |
         [JSONL_OPTION:
            [NEWLINE: <newline_str>]] |
         [CUSTOM_OPTION:
```

```
            NAME: <udf_name>
            PARAMSTR: <udf_parameter_string>]]
      [KEY:
        COLUMNS:
           - NAME: { <column_name> | __IGNORED__ }
             TYPE: <column_data_type>
           [ ... ]
         FORMAT: <key_data_format>
         [[DELIMITED_OPTION:
            DELIMITER: <delimiter_string> |
            [EOL_PREFIX: <prefix_string>]
            [QUOTE: <quote_char>]
            [ESCAPE: <escape_char>] ] |
         [AVRO_OPTION:
            [SCHEMA_REGISTRY_ADDR: <http://schemareg_host:schemareg_port> [, ... ]]
            [SCHEMA_CA_ON_GPDB: <sr_ca_file_path>]
            [SCHEMA_CERT_ON_GPDB: <sr_cert_file_path>]
            [SCHEMA_KEY_ON_GPDB: <sr_key_file_path>]
            [SCHEMA_MIN_TLS_VERSION: <minimum_version>]
            [SCHEMA_PATH_ON_GPDB: <path_to_file>]
            [BYTES_TO_BASE64: <boolean>]] |
         [CSV_OPTION:
            [DELIMITER: <delim_char>]
            [QUOTE: <quote_char>]
            [NULL_STRING: <nullstr_val>]
            [ESCAPE: <escape_char>]
            [FORCE_NOT_NULL: <columns>]
            [FILL_MISSING_FIELDS: <boolean>] |
         [CUSTOM_OPTION:
            NAME: <udf_name>
            PARAMSTR: <udf_parameter_string>]]
      [META:
         COLUMNS:
            - NAME: <meta_column_name>
              TYPE: { json | jsonb }
         FORMAT: json]
      [TRANSFORMER:
         PATH: <path_to_plugin_transform_library>
         ON_INIT: <plugin_transform_init_name>
         TRANSFORM: <plugin_transform_name>
         PROPERTIES:
           <plugin_transform_property_name>: <property_value>
           [ ... ] ]
      [FILTER: <filter_string>]
      [ENCODING: <char_set>]
      [ERROR_LIMIT: { <num_errors> | <percentage_errors> }]
   { OUTPUT:
      [SCHEMA: <output_schema_name>]
      TABLE: <table_name>
      [FILTER: <output_filter_string>]
      [MODE: <mode>]
      [MATCH_COLUMNS:
         - <match_column_name>
         [ ... ]]
      [ORDER_COLUMNS:
         - <order_column_name>
         [ ... ]]
      [UPDATE_COLUMNS:
         - <update_column_name>
```

```
      [ ... ]]
   [UPDATE_CONDITION: <update_condition>]
   [DELETE_CONDITION: <delete_condition>]
   [TRANSFORMER:
      TRANSFORM: <udf_transform_udf_name>
      PROPERTIES:
        <udf_transform_property_name>: <property_value>
        [ ... ]
      COLUMNS:
        - <udf_transform_column_name>
        [ ... ] ]
   [MAPPING:
      - NAME: <target_column_name>
        EXPRESSION: { <source_column_name> | <expression> }
      [ ... ]
        |
      <target_column_name> : { <source_column_name> | <expression> }
      [ ... ] ] |
OUTPUTS:
   - TABLE: <table_name>
     [MODE: <mode>]
     [MATCH_COLUMNS:
        - <match_column_name>
        [ ... ]]
     [ORDER_COLUMNS:
        - <order_column_name>
        [ ... ]]
     [UPDATE_COLUMNS:
        - <update_column_name>
        [ ... ]]
     [UPDATE_CONDITION: <update_condition>]
     [DELETE_CONDITION: <delete_condition>]
     [TRANSFORMER:
        TRANSFORM: <udf_transform_udf_name>
        PROPERTIES:
           <udf_transform_property_name>: <property_value>
           [ ... ]
        COLUMNS:
           - <udf_transform_column_name>
           [ ... ] ]
     [MAPPING:
        - NAME: <target_column_name>
          EXPRESSION: { <source_column_name> | <expression> }
        [ ... ]
          |
        <target_column_name> : { <source_column_name> | <expression> }
        [ ... ] ]
   [...] }
[METADATA:
   [SCHEMA: <metadata_schema_name>]]
COMMIT:
   SAVE_FAILING_BATCH: <boolean>
   RECOVER_FAILING_BATCH: <boolean> (Beta)
   MAX_ROW: <num_rows>
   MINIMAL_INTERVAL: <wait_time>
   CONSISTENCY: { strong | at-least | at-most | none }
   IDLE_DURATION: <idle_time>
[POLL:
   BATCHSIZE: <num_records>
```

```
      TIMEOUT: <poll_time>]
   [TASK:
      POST_BATCH_SQL: <udf_or_sql_to_run>
      BATCH_INTERVAL: <num_batches>
      PREPARE_SQL: <udf_or_sql_to_run>
      TEARDOWN_SQL: <udf_or_sql_to_run> ]
   [PROPERTIES:
      <kafka_property_name>: <kafka_property_value>
      [ ... ]]
[SCHEDULE:
   RETRY_INTERVAL: <retry_time>
   MAX_RETRIES: <num_retries>
   RUNNING_DURATION: <run_time>
   AUTO_STOP_RESTART_INTERVAL: <restart_time>
   MAX_RESTART_TIMES: <num_restarts>
   QUIT_AT_EOF_AFTER: <clock_time>]
[ALERT:
   COMMAND: <command_to_run>
   WORKDIR: <directory>
   TIMEOUT: <alert_time>]
```

Where you may specify any property value with a template variable that GPSS substitutes at runtime using the following syntax:

```
<PROPERTY:> {{<template_var>}}
```

Description

You specify load configuration parameters for the `gpsscli` and `gpkafka` utilities in a YAML-formatted configuration file. (This reference page uses the name `gpkafka.yaml` when referring to this file; you may choose your own name for the file.) Load parameters include Greenplum Database connection and target table information, Kafka broker and topic information, and error and commit thresholds.

Version 2 of the `gpkafka.yaml` configuration file syntax supports `KEY` and `VALUE` blocks. Version 1 does not.

The `gpsscli` and `gpkafka` utilities process the YAML configuration file in order, using indentation (spaces) to determine the document hierarchy and the relationships between the sections. The use of white space in the file is significant, and keywords are case-sensitive.

Keywords and Values

Greenplum Database Options

DATABASE: db_name
: The name of the Greenplum database.

USER: user_name
: The name of the Greenplum Database user/role. This user_name must have permissions as described in the Greenplum Streaming Server documentation.

PASSWORD: password

The password for the Greenplum Database user/role.

HOST: host

The host name or IP address of the Greenplum Database coordinator host.

PORT: greenplum_port

The port number of the Greenplum Database server on the coordinator host.

VERSION: 2

The version of the configuration file. You must specify `VERSION: 2` when you configure `VALUE` and/or `KEY` blocks in the file.

KAFKA:INPUT: Options

SOURCE

Kafka input configuration parameters.

BROKERS: kafka_broker_host:broker_port

The host and port identifying the Kafka broker.

TOPIC: kafka_topic

The name of the Kafka topic from which to load data. The topic must exist.

PARTITIONS: (partition_numbers)

A single, a comma-separated list, and/or a range of partition numbers from which GPSS reads messages from the Kafka topic. A range that you specify with the `M...N` syntax includes both the range start and end values. By default, GPSS reads messages from all partitions of the Kafka topic.

Ensure that you do not configure multiple jobs that specify overlapping partition numbers in the same topic; GPSS behavior is undefined.

FALLBACK_OFFSET: { earliest | latest }

Specifies the behaviour of GPSS when it detects a Kafka message offset gap. When set to `earliest`, GPSS automatically resumes a load operation from the earliest available published message. When set to `latest`, GPSS loads only new messages to the Kafka topic. If this property is not set, GPSS returns an error.

VALUE:

The Kafka message value field names, data types, and format. You must specify all Kafka data elements in the order in which they appear in the Kafka message. Optional when you specify a `KEY` block; GPSS ignores the Kafka message value in this circumstance.

KEY:

The Kafka message key field names, data types, and format. You must specify all Kafka key elements in the order in which they appear in the Kafka message. Optional when you specify a `VALUE` block; GPSS ignores the Kafka message key in this circumstance.

COLUMNS:NAME: column_name

The name of a key or value column. column_name must match the column name of the target Greenplum Database table. Specify `__IGNORED__` to omit this Kafka message data element from the load operation.

The default source-to-target data mapping behaviour of GPSS is to match a column name as defined in `COLUMNS:NAME` with a column name in the target Greenplum Database `TABLE`. You can override the default mapping by specifying a `MAPPING` block.

COLUMNS:TYPE: data_type

The data type of the column. You must specify an equivalent data type for each non-ignored Kafka message data element and the associated Greenplum Database table column.

FORMAT: data_format

The format of the Kafka message key or value data. You may specify a `FORMAT` of `avro`, `binary`, `csv`, `custom`, `delimited`, `json`, or `jsonl` for the key and value, with some restrictions.

avro

When you specify the `avro` data format, you must define only a single `json` type column in `COLUMNS`. If the Kafka message key or value schema is registered in a Confluent Schema Registry, you must also provide the `AVRO_OPTION`.

binary

When you specify the `binary` data format, you must define only a single `bytea` type column in `COLUMNS`.

csv

When you specify the `csv` data format, the message content cannot contain line ending characters (CR and LF).

You must not provide a `VALUE` block when you specify `csv` format for a `KEY` block. Similarly, you must not provide a `KEY` block when you specify `csv` format for a `VALUE` block.

custom

When you specify the `custom` data format, you must provide a `CUSTOM_OPTION`.

delimited

When you specify the `delimited` data format, you must provide a `DELIMITED_OPTION`.

json

When you specify the `json` data format, you must define only a single `json` type column in `COLUMNS`.

jsonl

When you specify the `jsonl` data format, you may provide a `JSONL_OPTION` to define a newline character.

AVRO_OPTION

Optional. When you specify `avro` as the `FORMAT`, you may provide `AVRO_OPTION`s that identify a schema registry location and optional SSL certificates and keys, and whether or not you want GPSS to convert Avro `bytes` fields into base64-encoded strings.

SCHEMA_REGISTRY_ADDR: schemareg_host:schemareg_port

When you specify `FORMAT: avro` and the Avro schema of the JSON data you want to load is registered in the Confluent Schema Registry, you must identify the host name and port number of each Confluent Schema Registry server in your Kafka cluster. You may specify more than one address, and at least one of the addresses must be legal.

SCHEMA_CA_ON_GPDB: sr_ca_file_path

The file system path to the CA certificate that GPSS uses to verify the peer. This file must reside in sr_ca_file_path on all Greenplum Database segment hosts.

SCHEMA_CERT_ON_GPDB: sr_cert_file_path

The file system path to the client certificate that GPSS uses to connect to the HTTPS schema registry. This file must reside in sr_cert_file_path on all Greenplum Database segment hosts.

SCHEMA_KEY_ON_GPDB: sr_key_file_path

The file system path to the private key file that GPSS uses to connect to the HTTPS schema registry. This file must reside in sr_key_file_path on all Greenplum Database segment hosts.

SCHEMA_MIN_TLS_VERSION: minimum_version

The minimum transport layer security (TLS) version that GPSS requests on the connection to the schema registry. Supported versions are `1.0`, `1.1`, `1.2`, or `1.3`. The default minimum TLS version is `1.0`.

SCHEMA_PATH_ON_GPDB: path_to_file

When you specify the `avro` format and the Avro schema of the JSON key or value data that you want to load is specified in a separate `.avsc` file, you must identify the file system location in path_to_file, and the file must reside in this location on every Greenplum Database segment host.

GPSS does not cache the schema. GPSS must reload the schema for every batch of Kafka data. Also, GPSS supports providing the schema for either the key or the value, but not both.

BYTES_TO_BASE64: boolean

When `true`, GPSS converts Avro `bytes` fields into base64-encoded strings. The default value is `false`, GPSS does not perform the conversion.

CSV_OPTION

When you specify `FORMAT: csv`, you may provide the following options:

DELIMITER: delim_char

Specifies a single ASCII character that separates columns within each message or row of data. The default delimiter is a comma (`,`).

QUOTE: quote_char

Specifies the quotation character. Because GPSS does not provide a default value for this property, you must specify a value.

NULL_STRING: nullstr_val

Specifies the string that represents the null value. Because GPSS does not specify a default value for this property, you must specify a value.

ESCAPE: escape_char

Specifies the single character that is used for escaping data characters in the content that might otherwise be interpreted as row or column delimiters. Make sure to choose an escape character that is not used anywhere in your actual column data. Because GPSS does not provide a default value for this property. you must specify a value.

FORCE_NOT_NULL: columns

Specifies a comma-separated list of column names to process as though each column were quoted and hence not a NULL value. For the default `null_string` (nothing between two

delimiters), missing values are evaluated as zero-length strings.

FILL_MISSING_FIELDS: boolean

Specifies the action of GPSS when it reads a row of data that has missing trailing field values (the row has missing data fields at the end of a line or row). The default value is `false`, GPSS returns an error when it encounters a row with missing trailing field values.

If set to `true`, GPSS sets missing trailing field values to `NULL`. Blank rows, fields with a `NOT NULL` constraint, and trailing delimiters on a line will still generate an error.

JSONL_OPTION

Optional. When you specify `FORMAT: jsonl`, you may choose to provide the `JSONL_OPTION` properties.

NEWLINE: newline_str

A string that specifies the new line character(s) that end each JSON record. The default newline is `"\n"`.

CUSTOM_OPTION

Optional. When you specify `FORMAT: custom`, you are required to provide the `CUSTOM_OPTION` properties. This block identifies the name and the arguments of a custom formatter user-defined function.

NAME: udf_name

The name of the custom formatter user-defined function.

PARAMSTR: udf_parameter_string

A string specifying the comma-separated list of arguments to pass to the custom formatter user-defined function.

DELIMITED_OPTION

Optional. When you specify `FORMAT: delimited`, you may choose to provide the `DELIMITER_OPTION` properties.

DELIMITER: delimiter_string

When you specify the `delimited` format, delimiter_string is required and must identify the data element delimiter. delimiter_string may be a multi-byte value, and up to 32 bytes in length. It may not contain quote and escape characters.

EOL_PREFIX: prefix_string

Specifies the prefix before the end of line character (`\n`) that indicates the end of a row. The default prefix is empty.

QUOTE: quote_char

Specifies the single ASCII quotation character. The default quote character is empty.

If you do not specify a quotation character, GPSS assumes that all columns are unquoted. If you do not specify a quotation character and do specify an escape character, GPSS assumes that all columns are unquoted and escapes the delimiter, end-of-line prefix, and escape itself.

When you specify a quotation character, you must specify an escape character. GPSS reads any content between quote characters as-is, except for escaped characters.

ESCAPE: escape_char

Specifies the single ASCII character used to escape special characters (for example, the delimiter, end-of-line prefix, quote, or escape itself). Therdefault escape character is empty.

When you specify an escape character and do not specify a quotation character, GPSS escapes only the delimiter, end-of-line prefix, and escape itself.
When you specify both an escape character and a quotation character, GPSS escapes only these characters.

META:

The field name, type, and format of the Kafka meta data. `META` must specify a single `json` or `jsonb` (Greenplum 6 only) type column and `FORMAT: json`. The available Kafka meta data properties include:

- `topic` (text) - the Kafka topic name
- `partition` (int) - the partition number
- `offset` (bigint) - the record location within the partition
- `timestamp` (bigint) - the time that the message was appended to the Kafka log

You can load any of these properties into the target table with a `MAPPING`, or use a property in the update or merge criteria for a load operation.

TRANSFORMER:

Input data transform block. An input data transformer is a plugin, a set of `go` functions that transform the data after it is read from the source. The semantics of the transform are function-specific. You specify the library and function names in this block, as well as the properties that GPSS passes to these functions:

PATH: path_to_plugin_transform_library
The file system location of the plugin transformer library on the Greenplum Streaming Server server host.
ON_INIT: plugin_transform_init_name
The name of an initialization function that GPSS calls when it loads the transform library.
TRANSFORM: plugin_transform_name
The name of the transform function. GPSS invokes this function for every message it reads.
PROPERTIES: plugin_transform_property_name: property_value
One or more property name and value pairs that GPSS passes to plugin_transform_init_name and plugin_transform_name.

FILTER: filter_string
The filter to apply to the Kafka input messages before GPSS loads the data into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. If the filter evaluates to `false`, the message is dropped. filter_string must be a valid SQL conditional expression and may reference one or more `KEY`, `VALUE`, or `META` column names.
ENCODING: char_set
The source data encoding. You can specify an encoding character set when the source data is of the `csv`, `custom`, `delimited`, or `json` format. GPSS supports the character sets identified in Character Set Support in the Tanzu Greenplum documentation.
ERROR_LIMIT: { num_errors | percentage_errors }
The error threshold, specified as either an absolute number or a percentage. `gpkafka load` exits when this limit is reached. The default `ERROR_LIMIT` is zero; GPSS deactivates error logging and

stops the load operation when it encounters the first error. Due to a limitation of the Greenplum Database external table framework, GPSS does not accept `ERROR_LIMIT: 1`.

KAFKA:OUTPUT: Options

You must specify only one of the `OUTPUT` or `OUTPUTS` blocks. You cannot specify both.

SCHEMA: output_schema_name

The name of the Greenplum Database schema in which table_name resides. Optional, the default schema is the `public` schema.

TABLE: table_name

The name of the Greenplum Database table into which GPSS loads the Kafka data.

FILTER: output_filter_string

The filter to apply to the output data before GPSS loads the data into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. If the filter evaluates to `false`, the message is dropped. output_filter_string must be a valid SQL conditional expression and may reference one or more `META` or `VALUE` column names.

MODE: mode

The table load mode. Valid mode values are `INSERT`, `MERGE`, or `UPDATE`. The default value is `INSERT`.

`UPDATE` - Updates the target table columns that are listed in `UPDATE_COLUMNS` when the input columns identified in `MATCH_COLUMNS` match the named target table columns and the optional `UPDATE_CONDITION` is true.

`UPDATE` is not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes (" ") to identify the column.

`MERGE` - Inserts new rows and updates existing rows when:

- columns are listed in `UPDATE_COLUMNS`,
- the `MATCH_COLUMNS` target table column values are equal to the input data, and
- an optional `UPDATE_CONDITION` is specified and met.

Deletes rows when:

- the `MATCH_COLUMNS` target table column values are equal to the input data, and
- an optional `DELETE_CONDITION` is specified and met.

New rows are identified when the `MATCH_COLUMNS` value in the source data does not have a corresponding value in the existing data of the target table. In those cases, the entire row from the source file is inserted, not only the `MATCH_COLUMNS` and `UPDATE_COLUMNS`. If there are multiple new `MATCH_COLUMNS` values in the input data that are the same, GPSS inserts or updates the target table using a random matching input row. When you specify `ORDER_COLUMNS`, GPSS sorts the input data on the specified column(s) and inserts or updates from the input row with the largest value.

`MERGE` is not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes (" ") to identify the column.

MATCH_COLUMNS:

Required if `MODE` is `MERGE` or `UPDATE`.

match_column_name
: Specifies the column(s) to use as the join condition for the update. The attribute value in the specified target column(s) must be equal to that of the corresponding source data column(s) in order for the row to be updated in the target table.

ORDER_COLUMNS:

Optional. May be specified in `MERGE MODE` to sort the input data rows.

order_column_name
: Specify the column(s) by which GPSS sorts the rows. When multiple matching rows exist in a batch, `ORDER_COLUMNS` is used with `MATCH_COLUMNS` to determine the input row with the largest value; GPSS uses that row to write/update the target.

UPDATE_COLUMNS:

Required if `MODE` is `MERGE` or `UPDATE`.

update_column_name
: Specifies the column(s) to update for the rows that meet the `MATCH_COLUMNS` criteria and the optional `UPDATE_CONDITION`.

UPDATE_CONDITION: update_condition
: Optional. Specifies a boolean condition, similar to that which you would declare in a `WHERE` clause, that must be met in order for a row in the target table to be updated (or inserted, in the case of a `MERGE`).

DELETE_CONDITION: delete_condition
: Optional. In `MERGE MODE`, specifies a boolean condition, similar to that which you would declare in a `WHERE` clause, that must be met for GPSS to delete rows in the target table that meet the `MATCH_COLUMNS` criteria.

TRANSFORMER:

Optional. Output data transform block. An output data transformer is a user-defined function (UDF) that transforms the data before it is loaded into Greenplum Database. The semantics of the UDF are transform-specific.

GPSS currently supports specifying only one of the `MAPPING` or (UDF) `TRANSFORMER` blocks in the load configuration file, not both.

TRANSFORM: udf_transform_udf_name
: The name of the output transform UDF. GPSS invokes this function for every batch of data it writes to Greenplum Database.

PROPERTIES: udf_transform_property_name: property_value
: One or more property name and value pairs that GPSS passes to udf_transform_udf_name.

COLUMNS: udf_transform_column_name
: The name of one or more columns involved in the transform.

MAPPING:

Optional. Overrides the default source-to-target column mapping. GPSS supports two mapping syntaxes.

GPSS currently supports specifying only one of the `MAPPING` or (UDF) `TRANSFORMER` blocks in the load configuration file, not both.

When you specify a `MAPPING`, ensure that you provide a mapping for all Kafka message key and value elements of interest. GPSS does not automatically match column names when you provide a `MAPPING`.

NAME: target_column_name
: Specifies the target Greenplum Database table column name.

EXPRESSION: { source_column_name | expression }
: Specifies a Kafka `COLUMNS:NAME` (source_column_name) or an expression. When you specify an expression, you may provide a value expression that you would specify in the `SELECT` list of a query, such as a constant value, a column reference, an operator invocation, a built-in or user-defined function call, and so on.

target_column_name: { source_column_name | expression }
: When you use this `MAPPING` syntax, specify the target_column_name and {source_column_name | expression} as described above.

KAFKA:OUTPUTS: Options

You must specify only one of the `OUTPUT` or `OUTPUTS` blocks. You cannot specify both.

TABLE: table_name
: The name of a Greenplum Database table into which GPSS loads the Kafka data.

other options
: As specified in the **KAFKA:OUTPUT: Options** section.

KAFKA:METADATA: Options

SCHEMA: metadata_schema_name
: The name of the Greenplum Database schema in which GPSS creates external and history tables. The default metadata_schema_name is `KAFKA:OUTPUT:SCHEMA`.

Greenplum Database COMMIT: Options

COMMIT:

: Controls how `gpkafka load` commits a batch of data to Greenplum Database. You may specify both `MAX_ROW` and `MINIMAL_INTERVAL` as long as both values are not zero (`0`). Try setting and tuning `MINIMAL_INTERVAL` to your environment; introduce a `MAX_ROW` setting only if you encounter high memory usage associated with message buffering.

SAVE_FAILING_BATCH: boolean

Determines whether or not GPSS saves data into a backup table before it writes the data to Greenplum Database. Saving the data in this manner aids recovery when GPSS encounters errors during the evaluation of expressions. The default is `false`; GPSS does not use a backup table, and returns immediately when it encounters an expression error. When you set this property to `true`, GPSS writes both the good and the bad data in the batch to a backup table named `gpssbackup_<jobhash>`, and continues to process incoming Kafka messages. You must then manually load the good data from the backup table into Greenplum **or** set `RECOVER_FAILING_BATCH` (Beta) to `true` to have GPSS automatically reload the good data.

Using a backup table to hedge against mapping errors may impact performance, especially when the data that you are loading has not been cleaned.

RECOVER_FAILING_BATCH: boolean (Beta)

When set to `true` and `SAVE_FAILING_BATCH` is also `true`, GPSS automatically reloads the good data in the batch and retains only the error data in the backup table. The default value is `false`; GPSS does not process the backup table.

Enabling this property requires that GPSS has the Greenplum Database privileges to create a function.

MAX_ROW: number_of_rows

The number of rows to batch before triggering an `INSERT` operation on the Greenplum Database table. The default value of `MAX_ROW` is `0`, which instructs GPSS to ignore this commit trigger condition.

MINIMAL_INTERVAL: wait_time

The minimum amount of time to wait (milliseconds) between each `INSERT` operation on the table. The default value is `5000`.

CONSISTENCY: { strong | at-least | at-most | none }

Specify how GPSS should manage message offsets when it acts as a high-level consumer. Valid values are `strong`, `at-least`, `at-most`, and `none`. The default value is `strong`. Refer to Understanding Kafka Message Offset Management for more detailed information.

IDLE_DURATION: idle_time

The maximum amount of time to wait (milliseconds) for the first batch of Kafka data. When you use this property to enable lazy load, GPSS waits until Kafka data is available before locking the target Greenplum table. You can specify:

- `0` (lazy load is deactivates)
- `-1` (lazy load is activated, the job never stops), or
- a positive value (lazy load is activated, the job stops after idle_time duration of no data in the Kafka topic) The default value is `0`.

Kafka POLL: Options

The `POLL` properties are deprecated and ignored by GPSS.

POLL:

Controls the polling time period and batch size when reading Kafka data.

BATCHSIZE: num_records
: The number of Kafka records in a batch. `BATCHSIZE` should be smaller than `COMMIT:MAX_ROW`. The default batch size is 200.

TIMEOUT: poll_time
: The maximum time, in milliseconds, to wait in a polling cycle if Kafka data is not available. You must specify a `TIMEOUT` greater than `100` milliseconds and less than `COMMIT:MINIMAL_INTERVAL`. The default poll timeout is 1000 milliseconds.

Greenplum Database TASK: Options

TASK:

Controls the execution and scheduling of a periodic (maintenance) task.

POST_BATCH_SQL: udf_or_sql_to_run
: The user-defined function or SQL command(s) that you want to run after the specified number of batches are read from Kafka. The default is null.

BATCH_INTERVAL: num_batches
: The number of batches to read before running udf_or_sql_to_run. The default batch interval is 0.

PREPARE_SQL: udf_or_sql_to_run
: The user-defined function or SQL command(s) that you want GPSS to run before it executes the job. The default is null, no command to run.

TEARDOWN_SQL: udf_or_sql_to_run
: The user-defined function or SQL command(s) that you want GPSS to run after the job stops. GPSS runs the function or command(s) on job success and job failure. The default is null, no command to run.

Kafka PROPERTIES: Options

PROPERTIES:

Kafka consumer configuration property names and values.

kafka_property_name
: The name of a Kafka property.

kafka_property_value
: The Kafka property value.

Job SCHEDULE: Options

SCHEDULE:

Controls the frequency and interval of restarting jobs.

RETRY_INTERVAL: retry_time
: The period of time that GPSS waits before retrying a failed job. You can specify the time interval in day (`d`), hour (`h`), minute (`m`), second (`s`), or millisecond (`ms`) integer units; do not mix units. The default retry interval is `5m` (5 minutes).

MAX_RETRIES: num_retries
: The maximum number of times GPSS attempts to retry a failed job. The default is 0, do not retry. If you specify a negative value, GPSS retries the job indefinitely.

RUNNING_DURATION: run_time
: The amount of time after which GPSS automatically stops a job. GPSS does not automatically stop a job by default.

AUTO_STOP_RESTART_INTERVAL: restart_time
: The amount of time after which GPSS restarts a job that it stopped due to reaching `RUNNING_DURATION`.

MAX_RESTART_TIMES: num_restarts
: The maximum number of times that GPSS restarts a job that it stopped due to reaching `RUNNING_DURATION`. The default is 0, do not restart the job.

QUIT_AT_EOF_AFTER: clock_time
: The clock time after which GPSS stops a job every day when it encounters an EOF. By default, GPSS does not automatically stop a job that reaches EOF. GPSS never stops a job when the current time is before `clock_time`, even when GPSS encounters an EOF.

Job ALERT: Options

Controls notification when a job is stopped for any reason (success, completion, error, user-initiated stop).

COMMAND: command_to_run
: The program that the GPSS server runs on the GPSS server host, including arguments. The command must be executable by GPSS.
command_to_run has access to job-related environment variables that GPSS sets, including: `$GPSSJOB_NAME`, `$GPSSJOB_STATUS`, and `$GPSSJOB_DETAIL`.

WORKDIR: directory
: The working directory for command_to_run. The default working directory is the directory from which you started the GPSS server process. If you specify a relative path, it is relative to the directory from which you started the GPSS server process.

TIMEOUT: alert_time
: The amount of time after a job stops, prompting GPSS to trigger the alert (and run command_to_run). You can specify the time interval in day (`d`), hour (`h`), minute (`m`), or second (`s`) integer units; do not mix units. The default alert timeout is `-1s` (no timeout).

Template Variables

GPSS supports using template variables to specify property values in the load configuration file.

You specify a template variable value in the load configuration file as follows:

```
<PROPERTY>: {{<template_var>}}
```

For example:

```
MAX_RETRIES: {{numretries}}
```

GPSS substitutes the template variable with a value that you specify via the `-p | --property <template_var=value>` option to the `gpsscli dryrun`, `gpsscli submit`, `gpsscli load`, or `gpkafka load` command.

For example, if the command line specifies:

```
--property numretries=10
```

GPSS substitutes occurrences of `{{numretries}}` in the load configuration file with the value `10` before submitting the job, and uses that value while the job is running.

Notes

If you created a database object name using a double-quoted identifier (delimited identifier), you must specify the delimited name within single quotes in the `gpkafka.yaml` configuration file. For example, if you create a table as follows:

```
CREATE TABLE "MyTable" ("MyColumn" text);
```

Your `gpkafka.yaml` YAML configuration file would refer to the above table and column names as:

```
  COLUMNS:
     - name: '"MyColumn"'
       type: text
OUTPUT:
   TABLE: '"MyTable"'
```

GPSS requires Kafka version 0.11 or newer for exactly-once delivery assurance. You can run with an older version of Kafka (but lose the exactly-once guarantee) by adding the following `PROPERTIES` block to your `gpkafka-v2.yaml` load configuration file:

```
PROPERTIES:
      api.version.request: false
      broker.version.fallback: 0.8.2.1
```

You can specify backslash escape sequences in the CSV `DELIMITER`, `QUOTE`, and `ESCAPE` options. GPSS supports the standard backslash escape sequences for backspace, form feed, newline, carriage return, and tab, as well as escape sequences that you specify in hexadecimal format (prefaced with `\x`). Refer to Backslash Escape Sequences in the PostgreSQL documentation for more information.

Examples

Load data from Kafka as defined in the Version 2 configuration file named `kafka2greenplumv2.yaml`:

```
gpkafka load kafka2greenplumv2.yaml
```

Example `kafka2greenplumv2.yaml` configuration file:

```
DATABASE: ops
USER: gpadmin
HOST: mdw-1
PORT: 5432
VERSION: 2
KAFKA:
   INPUT:
      SOURCE:
         BROKERS: kbrokerhost1:9092
         TOPIC: customer_expenses2
         PARTITIONS: (2, 5...7, 13)
      VALUE:
         COLUMNS:
           - NAME: c1
             TYPE: json
         FORMAT: avro
         AVRO_OPTION:
           SCHEMA_REGISTRY_ADDR: http://localhost:8081
      KEY:
         COLUMNS:
           - NAME: key
             TYPE: json
         FORMAT: avro
         AVRO_OPTION:
           SCHEMA_REGISTRY_ADDR: http://localhost:8081
      META:
         COLUMNS:
           - NAME: meta
             TYPE: json
         FORMAT: json
      ERROR_LIMIT: 25
   OUTPUT:
      SCHEMA: payables
      TABLE: expenses2
      MAPPING:
        - NAME: customer_id
          EXPRESSION: (c1->>'cust_id')::int
        - NAME: newcust
          EXPRESSION: ((c1->>'cust_id')::int > 5000000)::boolean
        - NAME: expenses
          EXPRESSION: (c1->>'expenses')::decimal
        - NAME: tax_due
          EXPRESSION: ((c1->>'expenses')::decimal * .075)::decimal
        - NAME: t
          EXPRESSION: (meta->>'topic')::text
   METADATA:
      SCHEMA: gpkafka_internal
   COMMIT:
      MINIMAL_INTERVAL: 2000
```

See Also

Loading Avro Data from Kafka, gpkafka.yaml, gpkafka load, gpsscli load, gpsscli submit

gpkafka.yaml

gpkafka configuration file (version 1).

Synopsis

```
DATABASE: <db_name>
USER: <user_name>
PASSWORD: <password>
HOST: <host>
PORT: <greenplum_port>
[VERSION: 1]
KAFKA:
   INPUT:
      SOURCE:
        BROKERS: <kafka_broker_host:broker_port> [, ... ]
        TOPIC: <kafka_topic>
      [COLUMNS:
         - NAME: { <column_name> | __IGNORED__ }
           TYPE: <column_data_type>
         [ ... ]]
      FORMAT: <data_format>
      [[DELIMITED_OPTION:
         DELIMITER: <delimiter_string>] |
      [AVRO_OPTION:
         [SCHEMA_REGISTRY_ADDR: <http://schemareg_host:schemareg_port> [, ... ]]
         [BYTES_TO_BASE64: <boolean>]] |
      [CUSTOM_OPTION:
         NAME: <udf_name>
         PARAMSTR: <udf_parameter_string>]]
      [FILTER: <filter_string>]
      [ERROR_LIMIT: { <num_errors> | <percentage_errors> }]
   OUTPUT:
      [SCHEMA: <output_schema_name>]
      TABLE: <table_name>
      [MODE: <mode>]
      [MATCH_COLUMNS:
         - <match_column_name>
         [ ... ]]
      [ORDER_COLUMNS:
         - <order_column_name>
         [ ... ]]
      [UPDATE_COLUMNS:
         - <update_column_name>
         [ ... ]]
      [UPDATE_CONDITION: <update_condition>]
      [DELETE_CONDITION: <delete_condition>]
      [MAPPING:
         - NAME: <target_column_name>
           EXPRESSION: { <source_column_name> | <expression> }
         [ ... ]
           |
         <target_column_name> : { <source_column_name> | <expression> }
         [ ... ] ]
   [METADATA:
      [SCHEMA: <metadata_schema_name>]]
   COMMIT:
      MAX_ROW: <num_rows>
      MINIMAL_INTERVAL: <wait_time>
   [POLL:
```

```
      BATCHSIZE: <num_records>
      TIMEOUT: <poll_time>]
   [TASK:
      POST_BATCH_SQL: <udf_or_sql_to_run>
      BATCH_INTERVAL: <num_batches>]
    [PROPERTIES:
      <kafka_property_name>: <kafka_property_value>
      [ ... ]]
```

Description

The `gpkafka.yaml` Version 1 configuration file format is deprecated and may be removed in a future release. Use the version 2 or version 3 (Beta) configuration file format to configure a Kafka load job.

You specify load configuration parameters for the `gpkafka` utilities in a YAML-formatted configuration file. (This reference page uses the name `gpkafka.yaml` when referring to this file; you may choose your own name for the file.) Load parameters include Greenplum Database connection and target table information, Kafka broker and topic information, and error and commit thresholds.

Version 1 of the `gpkafka.yaml` configuration file syntax does not support `KEY` and `VALUE` blocks.

The `gpkafka` utility processes the YAML configuration file in order, using indentation (spaces) to determine the document hierarchy and the relationships between the sections. The use of white space in the file is significant, and keywords are case-sensitive.

Keywords and Values

Greenplum Database Connection Options

DATABASE: db_name
: The name of the Greenplum database.

USER: user_name
: The name of the Greenplum Database user/role. This user_name must have permissions as described in the Greenplum Streaming Server documentation.

PASSWORD: password
: The password for the Greenplum Database user/role.

HOST: host
: The host name or IP address of the Greenplum Database coordinator host.

PORT: greenplum_port
: The port number of the Greenplum Database server on the coordinator host.

VERSION: 1
: Optional. The version of the load configuration file. The default version is Version 1.

KAFKA:INPUT: Options

SOURCE

Kafka input configuration parameters.

BROKERS: kafka_broker_host:broker_port
: The host and port identifying the Kafka broker.

TOPIC: kafka_topic
: The name of the Kafka topic from which to load data. The topic must exist.

COLUMNS:

The column names and data types. You must specify all Kafka data elements in the order in which they appear in the Kafka message. Optional when the column names and types match the target Greenplum Database table definition.

The default source-to-target data mapping behaviour of GPSS is to match a column name as defined in `COLUMNS:NAME` with a column name in the target Greenplum Database `TABLE`. You can override the default mapping by specifying a `MAPPING` block.

NAME: column_name
: The name of a column. column_name must match the column name of the target Greenplum Database table. Specify `__IGNORED__` to omit this Kafka message data element from the load operation.

TYPE: data_type
: The data type of the column. You must specify an equivalent data type for each non-ignored Kafka message data element and the associated Greenplum Database table column.

FORMAT: data_format

The format of the Kafka message value data. You may specify a `FORMAT` of `avro`, `binary`, `csv`, `custom`, `delimited`, or `json`.

avro
: When you specify the `avro` data format, you must define only a single `json` type column in `COLUMNS`. If the Kafka message value schema is registered in a Confluent Schema Registry, you must also provide the `AVRO_OPTION`.

binary
: When you specify the `binary` data format, you must define only a single `bytea` type column in `COLUMNS`.

csv
: When you specify the `csv` data format, the message content cannot contain line ending characters (CR and LF).

custom
: When you specify the `custom` data format, you must provide a `CUSTOM_OPTION`.

delimited
: When you specify the `delimited` data format, you must provide a `DELIMITED_OPTION`.

json
: When you specify the `json` data format, you must define only a single `json` type column in `COLUMNS`.

AVRO_OPTION

Optional. When you specify `avro` as the `FORMAT`, you may provide `AVRO_OPTION`s that identify a schema registry location and whether or not you want GPSS to convert Avro `bytes` fields into base64-encoded strings.

SCHEMA_REGISTRY_ADDR: http://schemareg_host:schemareg_port
: Optional. When you specify `avro` as the `FORMAT` and the Avro schema of the JSON data you want to load is registered in the Confluent Schema Registry, you must identify the host name and port number of each Confluent Schema Registry server in your Kafka cluster. You may specify more than one address, and at least one of the addresses must be legal.

BYTES_TO_BASE64: boolean
: When `true`, GPSS converts Avro `bytes` fields into base64-encoded strings. The default value is `false`, GPSS does not perform the conversion.

CUSTOM_OPTION

Optional. When you specify `custom` as the `FORMAT`, `CUSTOM_OPTION` is required. This block identifies the name and the arguments of a custom formatter user-defined function.

NAME: udf_name
: The name of the custom formatter user-defined function.

PARAMSTR: udf_parameter_string
: A string specifying the comma-separated list of arguments to pass to the custom formatter user-defined function.

DELIMITED_OPTION:DELIMITER: delimiter_string
: Optional. When you specify `delimited` as the `FORMAT`, delimiter_string is required and must identify the Kafka message data element delimiter. delimiter_string may be a multi-byte value, and up to 32 bytes in length. It may not contain quote and escape characters.

FILTER: filter_string
: The filter to apply to the Kafka input messages before GPSS loads the data into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. If the filter evaluates to `false`, the message is dropped. filter_string must be a valid SQL conditional expression and may reference one or more `COLUMNS` names.

ERROR_LIMIT: { num_errors | percentage_errors }
: The error threshold, specified as either an absolute number or a percentage. `gpkafka load` exits when this limit is reached. The default `ERROR_LIMIT` is zero; GPSS deactivates error logging, and stops the load operation when it encounters the first error. Due to a limitation of the Greenplum Database external table framework, GPSS does not accept `ERROR_LIMIT: 1`.

KAFKA:OUTPUT: Options

SCHEMA: output_schema_name
: The name of the Greenplum Database schema in which table_name resides. Optional, the default schema is the `public` schema.

TABLE: table_name
: The name of the Greenplum Database table into which GPSS loads the Kafka data.

MODE: mode

The table load mode. Valid mode values are `INSERT`, `MERGE`, or `UPDATE`. The default value is `INSERT`.

`UPDATE` - Updates the target table columns that are listed in `UPDATE_COLUMNS` when the input columns identified in `MATCH_COLUMNS` match the named target table columns and the optional `UPDATE_CONDITION` is true.

`UPDATE` is not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes (" ") to identify the column.

`MERGE` - Inserts new rows and updates existing rows when:

- columns are listed in `UPDATE_COLUMNS`,
- the `MATCH_COLUMNS` target table column values are equal to the input data, and
- an optional `UPDATE_CONDITION` is specified and met.

Deletes rows when:

- the `MATCH_COLUMNS` target table column values are equal to the input data, and
- an optional `DELETE_CONDITION` is specified and met.

New rows are identified when the `MATCH_COLUMNS` value in the source data does not have a corresponding value in the existing data of the target table. In those cases, the entire row from the source file is inserted, not only the `MATCH_COLUMNS` and `UPDATE_COLUMNS`. If there are multiple new `MATCH_COLUMNS` values in the input data that are the same, GPSS inserts or updates the target table using a random matching input row. When you specify `ORDER_COLUMNS`, GPSS sorts the input data on the specified column(s) and inserts or updates from the input row with the largest value.

`MERGE` is not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes (" ") to identify the column.

MATCH_COLUMNS:

Required if `MODE` is `MERGE` or `UPDATE`.

match_column_name
: Specifies the column(s) to use as the join condition for the update. The attribute value in the specified target column(s) must be equal to that of the corresponding source data column(s) in order for the row to be updated in the target table.

ORDER_COLUMNS:

Optional. May be specified in `MERGE MODE` to sort the input data rows.

order_column_name
: Specify the column(s) by which GPSS sorts the rows. When multiple matching rows exist in a batch, `ORDER_COLUMNS` is used with `MATCH_COLUMNS` to determine the input row with the largest value; GPSS uses that row to write/update the target.

UPDATE_COLUMNS:

Required if `MODE` is `MERGE` or `UPDATE`.

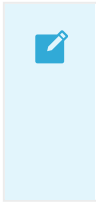
update_column_name
: Specifies the column(s) to update for the rows that meet the `MATCH_COLUMNS` criteria and the optional `UPDATE_CONDITION`.

UPDATE_CONDITION: update_condition
: Optional. Specifies a boolean condition, similar to that which you would declare in a `WHERE` clause, that must be met in order for a row in the target table to be updated (or inserted, in the case of a `MERGE`).

DELETE_CONDITION: delete_condition
: Optional. In `MERGE MODE`, specifies a boolean condition, similar to that which you would declare in a `WHERE` clause, that must be met for GPSS to delete rows in the target table that meet the `MATCH_COLUMNS` criteria.

MAPPING:

Optional. Overrides the default source-to-target column mapping. GPSS supports two mapping syntaxes.



When you specify a `MAPPING`, ensure that you provide a mapping for all Kafka data elements of interest. GPSS does not automatically match column names when you provide a `MAPPING`.

NAME: target_column_name
: Specifies the target Greenplum Database table column name.

EXPRESSION: { source_column_name | expression }
: Specifies a Kafka `COLUMNS:NAME` (source_column_name) or an expression. When you specify an expression, you may provide a value expression that you would specify in the `SELECT` list of a query, such as a constant value, a column reference, an operator invocation, a built-in or user-defined function call, and so on.

target_column_name: { source_column_name | expression }
: When you use this `MAPPING` syntax, specify the target_column_name and {source_column_name | expression} as described above.

KAFKA:METADATA: Options

SCHEMA: metadata_schema_name
: The name of the Greenplum Database schema in which GPSS creates external and history tables. The default metadata_schema_name is `KAFKA:OUTPUT:SCHEMA`.

Greenplum Database COMMIT: Options

COMMIT:

Controls how `gpkafka load` commits data to Greenplum Database. You must specify one of `MAX_ROW` or `MINIMAL_INTERVAL`. You may specify both configuration parameters as long as both values are not zero (`0`). Try setting and tuning `MINIMAL_INTERVAL` to your environment; introduce a `MAX_ROW` setting only if you encounter high memory usage associated with message buffering.

MAX_ROW: number_of_rows
: The number of rows to batch before triggering an `INSERT` operation on the Greenplum Database table. The default value of `MAX_ROW` is `0`, which instructs GPSS to ignore this commit trigger condition.

MINIMAL_INTERVAL: wait_time

The minimum amount of time to wait (milliseconds) between each `INSERT` operation on the table. The default value is `0`, wait forever.

Kafka POLL: Options

The `POLL` properties are deprecated and ignored by GPSS.

POLL:

Controls the polling time period and batch size when reading Kafka data.

BATCHSIZE: num_records
: The number of Kafka records in a batch. `BATCHSIZE` must be smaller than `COMMIT:MAX_ROW`. The default batch size is 200.

TIMEOUT: poll_time
: The maximum time, in milliseconds, to wait in a polling cycle if Kafka data is not available. You must specify a `TIMEOUT` greater than `100` milliseconds and less than `COMMIT:MINIMAL_INTERVAL`. The default poll timeout is 1000 milliseconds.

Greenplum Database TASK: Options

TASK:

Controls the execution and scheduling of a periodic (maintenance) task.

POST_BATCH_SQL: udf_or_sql_to_run
: The user-defined function or SQL command(s) that you want to run after the specified number of batches are read from Kafka. The default is null.

BATCH_INTERVAL: num_batches
: The number of batches to read before running udf_or_sql_to_run. The default batch interval is 0.

Kafka PROPERTIES: Options

PROPERTIES:

Kafka consumer configuration property names and values.

kafka_property_name
: The name of a Kafka property.

kafka_property_value
: The Kafka property value.

Notes

If you created a database object name using a double-quoted identifier (delimited identifier), you must specify the delimited name within single quotes in the `gpkafka.yaml` configuration file. For example, if you create a table as follows:

```
CREATE TABLE "MyTable" ("MyColumn" text);
```

Your `gpkafka.yaml` YAML configuration file would refer to the above table and column names as:

```
  COLUMNS:
     - name: '"MyColumn"'
       type: text
OUTPUT:
   TABLE: '"MyTable"'
```

GPSS requires Kafka version 0.11 or newer for exactly-once delivery assurance. You can run with an older version of Kafka (but lose the exactly-once guarantee) by adding the following `PROPERTIES` block to your `gpkafka.yaml` load configuration file:

```
PROPERTIES:
      api.version.request: false
      broker.version.fallback: 0.8.2.1
```

Examples

Load data from Kafka as defined in the Version 1 configuration file named `kafka2greenplum.yaml`:

```
gpkafka load kafka2greenplum.yaml
```

Example `kafka2greenplum.yaml` configuration file:

```
DATABASE: ops
USER: gpadmin
HOST: mdw-1
PORT: 5432
KAFKA:
   INPUT:
      SOURCE:
         BROKERS: kbrokerhost1:9092
         TOPIC: customer_expenses
      COLUMNS:
         - NAME: cust_id
           TYPE: int
         - NAME: month
           TYPE: int
         - NAME: expenses
           TYPE: decimal(9,2)
      FORMAT: delimited
      DELIMITED_OPTION:
         DELIMITER: '|'
      ERROR_LIMIT: 25
   OUTPUT:
      SCHEMA: payables
      TABLE: expenses
      MAPPING:
        - NAME: customer_id
          EXPRESSION: cust_id
        - NAME: newcust
          EXPRESSION: cust_id > 5000000
        - NAME: expenses
          EXPRESSION: expenses
        - NAME: tax_due
```

```
         EXPRESSION: expenses * .0725
   METADATA:
      SCHEMA: gpkafka_internal
   COMMIT:
      MINIMAL_INTERVAL: 2000
```

See Also

gpkafka-v2.yaml, gpkafka load, gpss, gpss.json

filesource-v3.yaml (Beta)

GPSS load configuration file for a File data source (version 3).

Synopsis

```
version: v3
```

```
targets:
- gpdb:
    host: <host>
    port: <greenplum_port>
    user: <user_name>
    password: <password>
    database: <db_name>
    work_schema: <work_schema_name>
    error_limit: <num_errors> | <percentage_errors>
    filter_expression: <filter_string>
    tables:
      - table: <table_name>
        schema: <schema_name>
        mode:
          # specify a single mode property block (described below)
          insert: {}
          update:
            <mode_specific_property>: <value>
            ...
          merge:
            <mode_specific_property>: <value>
            ...
        mapping:
          <target_column_name> : <source_column_name> | <expression>
          ...
        filter: <output_filter_string>
      ...
```

```
sources:
- file:
    uri:
      - <file_path>
      ...
    exec:
      command: <command_to_run>
      workdir: <directory>
```

```
      stderr_as_fail: <boolean>
    content:
      <data_format>:
        <column_spec>
        <other_props>
    encoding: <char_set>
    task:
      prepare_statement: <udf_or_sql_to_run>
      teardown_statement: <udf_or_sql_to_run>
    meta:
      json:
        column:
          name: meta
          type: json
```

```
option:
  schedule:
    max_retries: <num_retries>
    retry_interval: <retry_time>
    running_duration: <run_time>
    auto_stop_restart_interval: <restart_time>
    max_restart_times: <num_restarts>
    quit_at_eof_after: <clock_time>
  alert:
    command: <command_to_run>
    workdir: <directory>
    timeout: <alert_time>
```

Where the mode_specific_propertys that you can specify for `update` and `merge` mode follow:

```
update:
  match_columns: [<match_column_names>]
  order_columns: [<order_column_names>]
  update_columns: [<update_column_names>]
  update_condition: <update_condition>
```

```
merge:
  match_columns: [<match_column_names>]
  update_columns: [<update_column_names>]
  order_columns: [<order_column_names>]
  update_condition: <update_condition>
  delete_condition: <delete_condition>
```

Where data_format, column_spec, and other_props are one of the following blocks (data source-specific):

```
avro:
  source_column_name: <column_name>
  schema_url: <http://schemareg_host:schemareg_port> %, ...%
  bytes_to_base64: <boolean>
```

```
binary:
  source_column_name: <column_name>
```

```
csv:
  columns:
```

```
    - name: <column_name>
      type: <column_data_type>
    ...
  delimiter: <delim_char>
  quote: <quote_char>
  null_string: <nullstr_val>
  escape: <escape_char>
  force_not_null: <columns>
  fill_missing_fields: <boolean>
```

```
custom:
  columns:
    - name: <column_name>
      type: <column_data_type>
    ...
  name: <formatter_name>
  options:
    - <optname>=<optvalue>
    ...
```

```
delimited:
  columns:
    - name: <column_name>
      type: <column_data_type>
    ...
  delimiter: <delimiter_string>
  eol_prefix: <prefix_string>
  quote: <quote_char>
  escape: <escape_char>
```

```
json:
  column:
    name: <column_name>
    type: json | jsonb
  is_jsonl: <boolean>
  newline: <newline_str>
```

And where you may specify any property value with a template variable that GPSS substitutes at runtime using the following syntax:

```
<property:> {{<template_var>}}
```

Description

Version 3 of the GPSS load configuration file is different in both content and format than previous versions of the file. Certain symbols used in the GPSS version 1 and 2 configuration file reference page syntax have different meanings in version 3 syntax:

- Brackets `[]` are literal and are used to specify a list in version 3. They are no longer used to signify the optionality of a property.
- Curly braces `{}` are literal and are used to specify YAML mappings in version 3 syntax. They are no longer used with the pipe symbol (`|`) to identify a list of

choices.

You specify the configuration properties for a Greenplum Streaming Server (GPSS) file load job in a YAML-formatted configuration file that you provide to the `gpsscli submit` or `gpsscli load` commands. There are three types of configuration properties in this file - those that identify the Greenplum Database connection and target table, properties specific to the file data source that you will load into Greenplum, and job-related properties.

This reference page uses the name `filesource-v3.yaml` to refer to this file; you may choose your own name for the file.

The `gpsscli` utility processes the YAML configuration file keywords in order, using indentation (spaces) to determine the document hierarchy and the relationships between the sections. The use of white space in the file is significant. Keywords are not case-sensitive.

Keywords and Values

version Property

version: v3
: The version of the configuration file. You must specify `version: v3`.

targets:gpdb Properties

host: host
: The host name or IP address of the Greenplum Database coordinator host.

port: greenplum_port
: The port number of the Greenplum Database server on the coordinator host.

user: user_name
: The name of the Greenplum Database user/role. This user_name must have permissions as described in the Configuring Greenplum Database Role Privileges.

password: password
: The password for the Greenplum Database user/role.

database: db_name
: The name of the Greenplum database.

work_schema: work_schema_name
: The name of the Greenplum Database schema in which GPSS creates internal tables. The default work_schema_name is `public`.

error_limit: num_errors | percentage_errors
: The error threshold, specified as either an absolute number or a percentage. GPSS stops running the job when this limit is reached.

filter_expression: filter_string
: The filter to apply to the input data before GPSS loads the data into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. If the filter evaluates to `false`, the message is dropped. filter_string must be a valid SQL conditional expression and may reference one or more source value, key, or meta column names.

tables:
: The Greenplum Database tables, and the data that GPSS will load into each.

table: table_name
: The name of the Greenplum Database table into which GPSS loads the data.

schema: schema_name
: The name of the Greenplum Database schema in which table_name resides. Optional, the default schema is the `public` schema.

mode:
: The table load mode; `insert`, `merge`, or `update`. The default mode is `insert`.

`update` and `merge` are not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes (" ") to identify the column.

insert:
: Inserts source data into Greenplum.

update:
: Updates the target table columns that are listed in `update_columns` when the input columns identified in `match_columns` match the named target table columns and the optional `update_condition` is true.

merge:
: Inserts new rows and updates existing rows when:

- columns are listed in `update_columns`,
- the `match_columns` target table column values are equal to the input data, and
- an optional `update_condition` is specified and met.

Deletes rows when:

- the `match_columns` target table column values are equal to the input data, and
- an optional `delete_condition` is specified and met.

New rows are identified when the `match_columns` value in the source data does not have a corresponding value in the existing data of the target table. In those cases, the entire row from the source file is inserted, not only the `match_columns` and `update_columns`. If there are multiple new `match_columns` values in the input data that are the same, GPSS inserts or updates the target table using a random matching input row. When you specify `order_columns`, GPSS sorts the input data on the specified column(s) and inserts or updates from the input row with the largest value.

mode_property_name: value
: The name to value mapping for a mode property. Each `mode` supports one or more of the following properties as specified in the Synopsis.

match_columns: [match_column_names]
: A comma-separated list that specifies the column(s) to use as the join condition for the update. The attribute value in the specified target column(s) must be equal to that of the corresponding source data column(s) in order for the row to be updated in the target table.

Required when `mode` is `merge` or `update`.

order_columns: [order_column_names]

A comma-separated list that specifies the column(s) by which GPSS sorts the rows. When multiple matching rows exist in a batch, `order_columns` is used with `match_columns` to determine the input row with the largest value; GPSS uses that row to write/update the target. Optional. May be specified in `merge mode` to sort the input data rows.

update_columns: [update_column_names]

A column-sparated list that specifies the column(s) to update for the rows that meet the `match_columns` criteria and the optional `update_condition`.

Required when `mode` is `merge` or `update`.

update_condition: update_condition

Specifies a boolean condition, similar to that which you would declare in a `WHERE` clause, that must be met in order for a row in the target table to be updated (or inserted, in the case of a `merge`). Optional.

delete_condition: delete_condition

In `merge mode`, specifies a boolean condition, similar to that which you would declare in a `WHERE` clause, that must be met for GPSS to delete rows in the target table that meet the `match_columns` criteria. Optional.

mapping:

Optional. Overrides the default source-to-target column mapping.

When you specify a `mapping`, ensure that you provide a mapping for all source data elements of interest. GPSS does not automatically match column names when you provide a `mapping` block.

```
target\_column\_name: source\_column\_name \| expression
:   target\_column\_name specifies the target Greenplum Database table column nam
e. GPSS maps this column name to the source column name specified in source\_colu
mn\_name, or to an expression. When you specify an expression, you may provide a
value expression that you would specify in the `SELECT` list of a query, such as
a constant value, a column reference, an operator invocation, a built-in or user-
defined function call, and so on.
```

filter: output_filter_string

The filter to apply to the output data before GPSS loads the data into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. If the filter evaluates to `false`, the message is dropped. output_filter_string must be a valid SQL conditional expression and may reference one or more `META` or `VALUE` column names.

sources:file: Options

The file input configuration parameters. You must provide exactly one of `uri` or an `exec` block.

uri

The path to the file.

file_path

A URL identifying a file or files to be loaded. You can specify wildcards in any element of the path. To load all files in a directory, specify `dirname/*`.

exec

The execution options for the command whose `stdout` GPSS loads into Greenplum Database.

command: command_to_run

The program that the GPSS server runs on the local host, including the arguments. The command must be executable by GPSS, and can include pipe and quote characters.

workdir: directory

The working directory for the child process. The default working directory is the directory from which you started the GPSS server process. If you specify a relative path, it is relative to the directory from which you started the GPSS server process.

stderr_as_fail: boolean

Specifies whether data written to `stderr` constitutes failure of the command, regardless of the command return value. The default value is `false`; GPSS does not consider writing to `stderr` a failure, and will write a message to the GPSS log file. When true, GPSS treats any output to `stderr` as a failure, and rolls back the operation.

content:

The file type, field names, and type-specific properties of the file data. You must specify all data elements in the order in which they appear in the file.

column_spec

The source to Greenplum column mapping. The supported column specification differs for different data formats as described below.

The default source-to-target data mapping behaviour of GPSS is to match a column name as defined in `source_column_name`, `column:name`, or `columns:name` with a column name in the target Greenplum Database `table`. You can override the default mapping by specifying a `mapping:` block.

data_format

The format of the key or value data. You may specify a data_format of `avro`, `binary`, `csv`, `custom`, `delimited`, or `json` for the key and value, with some restrictions.

avro

When you specify the `avro` data format for a key or value, GPSS reads the data into a single `json`-type column. You may specify a schema registery location and whether or not you want GPSS to convert `bytes` fields into base64-encoded strings.

source_column_name: column_name

The name of the single `json`-type column into which GPSS reads the key or value data.

schema_url: schemareg_host:schemareg_port

When you specify the `avro` format and the Avro schema of the JSON data that you want to load is registered in the Confluent Schema Registry, you must identify the host name and port number of each Confluent Schema Registry server in your Kafka cluster. You may specify more than one address, and at least one of the addresses must be legal.

bytes_to_base64: boolean

When `true`, GPSS converts Avro `bytes` fields into base64-encoded strings. The default value is `false`, GPSS does not perform the conversion.

binary

When you specify the `binary` data format, GPSS reads the data into a single `bytea`-type column.

source_column_name: column_name
: The name of the single `bytea`-type column into which GPSS reads the key or value data.

csv
: When you specify the `csv` data format, GPSS reads the data into the list of columns that you specify. The file content cannot contain line ending characters (CR and LF).

columns:
: A set of column name/type mappings. The value `[]` specifies all columns.

name: column_name
: The name of a key or value column. column_name must match the column name of the target Greenplum Database table.

type: column_data_type
: The data type of the column. You must specify an equivalent data type for each data element and the associated Greenplum Database table column.

delimiter: delim_char
: Specifies a single ASCII character that separates columns within each message or row of data. The default delimiter is a comma (`,`).

quote: quote_char
: Specifies the quotation character. Because GPSS does not provide a default value for this property, you must specify a value.

null_string: nullstr_val
: Specifies the string that represents the null value. Because GPSS does not provide a default value for this property, you must specify a value.

force_not_null: columns
: Specifies a comma-separated list of column names to process as though each column were quoted and hence not a NULL value. For the default `null_string` (nothing between two delimiters), missing values are evaluated as zero-length strings.

fill_missing_fields: boolean
: Specifies the action of GPSS when it reads a row of data that has missing trailing field values (the row has missing data fields at the end of a line or row). The default value is `false`, GPSS returns an error when it encounters a row with missing trailing field values.

  If set to `true`, GPSS sets missing trailing field values to `NULL`. Blank rows, fields with a `NOT NULL` constraint, and trailing delimiters on a line will still generate an error.

newline: newline_str
: Specifies the string that represents a new line. + GPSS does not specify a default value.

custom
: When you specify the `custom` data format, GPSS uses the custom formatter that you specify to process the input data before writing it to Greenplum Database.

columns:
: A set of column name/type mappings. The value `[]` specifies all columns.

name: column_name
: The name of a key or value column. column_name must match the column name of the target Greenplum Database table.

type: column_data_type
: The data type of the column. You must specify an equivalent data type for each data element and the associated Greenplum Database table column.

name: formatter_name
: When you specify the `custom` data format, formatter_name is required and must identify the name of the formatter user-defined function that GPSS should use when loading the data.

options:
: A set of function argument name=value pairs.

optname=optvalue
: The name and value of the set of arguments to pass into the formatter_name UDF.

delimited
: When you specify the `delimited` data format, GPSS reads the data into the list of columns that you specify. You must specify the data `delimiter`.

columns:
: A set of column name/type mappings. The value `[]` specifies all columns.

name: column_name
: The name of a key or value column. column_name must match the column name of the target Greenplum Database table.

type: column_data_type
: The data type of the column. You must specify an equivalent data type for each data element and the associated Greenplum Database table column.

delimiter: delimiter_string
: When you specify the `delimited` data format, delimiter_string is required and must identify the data element delimiter. delimiter_string may be a multi-byte value, and up to 32 bytes in length. It may not contain quote and escape characters.

eol_prefix: prefix_string
: Specifies the prefix before the end of line character (`\n`) that indicates the end of a row. The default prefix is empty.

quote: quote_char
: Specifies the single ASCII quotation character. The default quote character is empty.
: If you do not specify a quotation character, GPSS assumes that all columns are unquoted. If you do not specify a quotation character and do specify an escape character, GPSS assumes that all columns are unquoted and escapes the delimiter, end-of-line prefix, and escape itself.
: When you specify a quotation character, you must specify an escape character. GPSS reads any content between quote characters as-is, except for escaped characters.

escape: escape_char
: Specifies the single ASCII character used to escape special characters (for example, the delimiter, end-of-line prefix, quote, or escape itself). Therdefault escape character is empty.
: When you specify an escape character and do not specify a quotation character, GPSS escapes only the delimiter, end-of-line prefix, and escape itself.
: When you specify both an escape character and a quotation character, GPSS escapes only these characters.

json
: When you specify the `json` data format, GPSS can read the data as a single JSON object or as a single JSON record per line.

column:
: A single column name/type mapping.

name: column_name
: The name of the key or value column. column_name must match the column name of the target Greenplum Database table.

type: json | jsonb | gp_jsonb (Beta) | gp_json (Beta)
: The data type of the column.

is_jsonl: boolean
: Identifies whether or not GPSS reads the JSON data as a single object or single-record-per-line. The default is `false`, GPSS reads the JSON data as a single object.

newline: newline_str
: A string that specifies the new line character(s) that end each JSON record. The default newline is `"\n"`.

encoding: char_set
: The source data encoding. You can specify an encoding character set when the source data is of the `csv`, `custom`, `delimited`, or `json` format. GPSS supports the character sets identified in Character Set Support in the Greenplum Database documentation.

task:
: File data source task properties.

prepare_statement: udf_or_sql_to_run
: A user-defined function or SQL command(s) that you want GPSS to run before it executes the job. The default is null, no command to run.

teardown_statement: udf_or_sql_to_run
: A user-defined function or SQL command(s) that you want GPSS to run after the job stops. GPSS runs the function or command(s) on job success and job failure. The default is null, no command to run.

meta:
: The data type and field name of the file meta data. `meta:` must specify the `json` or `jsonb` (Greenplum 6 only) data format, and a single json-type column.

You can load this property into the target table with a `mapping`, or use the property in the update or merge criteria for a load operation

option: Properties

schedule:
: Controls the frequency and interval of restarting jobs.

retry_interval: retry_time
: The period of time that GPSS waits before retrying a failed job. You can specify the time interval in day (`d`), hour (`h`), minute (`m`), second (`s`), or millisecond (`ms`) integer units; do not mix units. The default retry interval is `5m` (5 minutes).

max_retries: num_retries
: The maximum number of times that GPSS attempts to retry a failed job. The default is 0, do not retry. If you specify a negative value, GPSS retries the job indefinitely.

running_duration: run_time
: The amount of time after which GPSS automatically stops a job. GPSS does not automatically stop a job by default.

auto_stop_restart_interval: restart_time
: The amount of time after which GPSS restarts a job that it stopped due to reaching `running_duration`.

max_restart_times: num_restarts
: The maximum number of times that GPSS restarts a job that it stopped due to reaching `running_duration`. The default is 0, do not restart the job.

quit_at_eof_after: clock_time
: The clock time after which GPSS stops a job every day when it encounters an EOF. By default, GPSS does not automatically stop a job that reaches EOF. GPSS never stops a job when the current time is before `clock_time`, even when GPSS encounters an EOF.

alert:

Controls notification when a job is stopped for any reason (success, completion, error, user-initiated stop).

command: command_to_run
: The program that the GPSS server runs on the GPSS server host, including arguments. The command must be executable by GPSS.
command_to_run has access to job-related environment variables that GPSS sets, including: `$GPSSJOB_NAME`, `$GPSSJOB_STATUS`, and `$GPSSJOB_DETAIL`.

workdir: directory
: The working directory for command_to_run. The default working directory is the directory from which you started the GPSS server process. If you specify a relative path, it is relative to the directory from which you started the GPSS server process.

timeout: alert_time
: The amount of time after a job stops, prompting GPSS to trigger the alert (and run command_to_run). You can specify the time interval in day (`d`), hour (`h`), minute (`m`), or second (`s`) integer units; do not mix units. The default alert timeout is `-1s` (no timeout).

Template Variables

GPSS supports using template variables to specify property values in the load configuration file.

You specify a template variable value in the load configuration file as follows:

```
<property>: {{<template_var>}}
```

For example:

```
max_retries: {{numretries}}
```

GPSS substitutes the template variable with a value that you specify via the `-p | --property <template_var=value>` option to the `gpsscli dryrun`, `gpsscli submit`, or `gpsscli load` command.

For example, if the command line specifies:

```
--property numretries=10
```

GPSS substitutes occurrences of `{{numretries}}` in the load configuration file with the value `10` before submitting the job, and uses that value while the job is running.

Notes

If you created a database object name using a double-quoted identifier (delimited identifier), you must specify the delimited name within single quotes in the load configuration file. For example, if you create a table as follows:

```
CREATE TABLE "MyTable" (c1 text);
```

Your YAML configuration file would refer to the table name as:

```
targets:
- gpdb:
    tables:
      - table: '"MyTable"'
```

You can specify backslash escape sequences in the CSV `delimiter`, `quote`, and `escape` options. GPSS supports the standard backslash escape sequences for backspace, form feed, newline, carriage return, and tab, as well as escape sequences that you specify in hexadecimal format (prefaced with `\x`). Refer to Backslash Escape Sequences in the PostgreSQL documentation for more information.

Examples

Submit a job to load data from an Avro file as defined in the version 3 load configuration file named `loadfromfile_v3.yaml`:

```
$ gpsscli submit loadfromfile_v3.yaml
```

Example `loadfromfile_v3.yaml` configuration file:

```
version: v3
targets:
- gpdb:
    host: mdw-1
    port: 15432
    user: gpadmin
    password: changeme
    database: testdb
    work_schema: public
    error_limit: "25"
    tables:
      - table: orders
        schema: public
        mode:
          merge:
            match_columns: [pk]
            order_columns: [seq]
            delete_condition: flag = 0
        mapping:
          data: data
          pk: pk
          seq: seq
sources:
- file:
    uri:
      - file:///tmp/data.csv
    content:
```

```
      csv:
        columns:
          - name: pk
            type: int
          - name: seq
            type: int
          - name: data
            type: text
          - name: flag
            type: int
option:
  schedule: {}
```

See Also

gpsscli load, gpsscli submit

filesource-v2.yaml

Load configuration file for a GPSS file data source.

Synopsis

```
DATABASE: <db_name>
USER: <user_name>
PASSWORD: <password>
HOST: <coordinator_host>
PORT: <greenplum_port>
VERSION: 2
FILE:
    INPUT:
       SOURCE:
         { URL: <file_path> |
         EXEC:
           COMMAND: <command_to_run>
           WORKDIR: <directory>
           STDERR_AS_FAIL: <boolean> }
       VALUE:
         [COLUMNS:
            - NAME: <column_name>
              TYPE: <column_data_type>
            [ ... ]]
         FORMAT: <value_data_format>
         [AVRO_OPTION:
            BYTES_TO_BASE64: <boolean>]
         [CSV_OPTION:
            [DELIMITER: <delim_char>]
            [QUOTE: <quote_char>]
            [NULL_STRING: <nullstr_val>]
            [ESCAPE: <escape_char>]
            [FORCE_NOT_NULL: <columns>]
            [FILL_MISSING_FIELDS: <boolean>]]
            [NEWLINE: <newline_str>]]
         [DELIMITED_OPTION:
            [DELIMITER: <delimiter_string>]
```

```
            [EOL_PREFIX: <prefix_string>]
            [QUOTE: <quote_char>]
            [ESCAPE: <escape_char>]]
         [JSONL_OPTION:
            [NEWLINE: <newline_str>]]
       [META:
         COLUMNS:
            - NAME: <meta_column_name>
              TYPE: { json | jsonb }
         FORMAT: json
       [FILTER: <filter_string>]
       [ENCODING: <char_set>]
       [ERROR_LIMIT: { <num_errors> | <percentage_errors> }]
    OUTPUT:
       [SCHEMA: <output_schema_name>]
       TABLE: <table_name>
       [FILTER: <output_filter_string>]
       [MODE: <mode>]
       [MATCH_COLUMNS:
          - <match_column_name>
          [ ... ]]
       [ORDER_COLUMNS:
          - <order_column_name>
          [ ... ]]
       [UPDATE_COLUMNS:
          - <update_column_name>
          [ ... ]]
       [UPDATE_CONDITION: <update_condition>]
       [DELETE_CONDITION: <delete_condition>]
    TASK:
       PREPARE_SQL: <udf_or_sql_command_to_run>
       TEARDOWN_SQL: <udf_or_sql_command_to_run>
[SCHEDULE:
   RETRY_INTERVAL: <retry_time>
   MAX_RETRIES: <num_retries>
   RUNNING_DURATION: <run_time>
   AUTO_STOP_RESTART_INTERVAL: <restart_time>
   MAX_RESTART_TIMES: <num_restarts>
   QUIT_AT_EOF_AFTER: <clock_time>]
[ALERT:
   COMMAND: <command_to_run>
   WORKDIR: <directory>
   TIMEOUT: <alert_time>]
```

Where you may specify any property value with a template variable that GPSS substitutes at runtime using the following syntax:

```
<PROPERTY:> {{<template_var>}}
```

Description

You specify the configuration parameters for a Greenplum Streaming Server (GPSS) file load job in a YAML-formatted configuration file that you provide to the `gpsscli submit` or `gpsscli load` commands. There are two types of configuration parameters in this file - those that identify the Greenplum Database connection and target table, and parameters specific to the file data source that you will load into Greenplum.

This reference page uses the name `filesource.yaml` to refer to this file; you may choose your own name for the file.

The `gpsscli` utility processes the YAML configuration file keywords in order, using indentation (spaces) to determine the document hierarchy and the relationships between the sections. The use of white space in the file is significant, and keywords are case-sensitive.

Keywords and Values

Greenplum Database Options

DATABASE: db_name
: The name of the Greenplum database.

USER: user_name
: The name of the Greenplum Database user/role. This user_name must have permissions as described in Configuring Greenplum Database Role Privileges.

PASSWORD: password
: The password for the Greenplum Database user/role.

HOST: coordinator_host
: The host name or IP address of the Greenplum Database coordinator host.

PORT: greenplum_port
: The port number of the Greenplum Database server on the coordinator host.

VERSION: 2
: The version of the GPSS load configuration file. GPSS supports version 2 of this format for a file data source.

FILE:INPUT: Options

SOURCE:

: The file input configuration parameters. You must provide exactly one of `URL` or an `EXEC` block.

  URL: file_path
  : The URL identifying the file or files to be loaded. You can specify wildcards in any element of the path. To load all files in a directory, specify `dirname/*`.

  EXEC:
  : The execution options for the command whose `stdout` GPSS loads into Greenplum Database.

```
COMMAND: command_to_run
:   The program that the GPSS server runs on the local host, including the argume
nts. The command must be executable by GPSS, and can include pipe and quote chara
cters.
```

```
WORKDIR: directory
:   The working directory for the child process. The default working directory is
the directory from which you started the GPSS server process. If you specify a re
lative path, it is relative to the directory from which you started the GPSS serv
er process.
```

```
STDERR_AS_FAIL: boolean
:   Specifies whether data written to `stderr` constitutes failure of the comman
d, regardless of the command return value. The default value is `false`; GPSS doe
s not consider writing to `stderr` a failure, and will write a message to the GPS
S log file. When true, GPSS treats any output to `stderr` as a failure, and rolls
back the operation.
```

VALUE:

The field names, types, and format of the file data. You must specify all data elements in the order in which they appear in the file.

COLUMNS:NAME: column_name
: The name of a data value column. column_name must match the column name of the target Greenplum Database table.

: The default source-to-target data mapping behaviour of GPSS is to match a column name as defined in `COLUMNS:NAME` with a column name in the target Greenplum Database `TABLE`. You can override the default mapping by specifying a `MAPPING` block.

COLUMNS:TYPE: data_type
: The data type of the column. You must specify a compatible data type for each data element and the associated Greenplum Database table column.

FORMAT: data_format
: The format of the value data. You may specify a `FORMAT` of `avro`, `binary`, `csv`, `json`, or `jsonl` for the value data, with some restrictions.

avro
: When you specify the `avro` data format, you must define only a single `json` type column in `COLUMNS`. If the schema is registered in a Confluent Schema Registry, you must also provide the `AVRO_OPTION`.

binary
: When you specify the `binary` data format, you must define only a single `bytea` type column in `COLUMNS`.

csv
: When you specify the `csv` data format, the message content cannot contain line ending characters (CR and LF). You may also choose to provide `CSV_OPTION`s.
When you specify `FORMAT: csv`, you must not provide a `META` block.

delimited
: When you specify the `delimited` data format, the delimited message content may contain a multi-byte delimiter. You must provide `DELIMITED_OPTION`s.

json
: When you specify the `json` data format, you must define only a single `json` type column in `COLUMNS`.

jsonl
: When you specify the `jsonl` data format, you may provide a `JSONL_OPTION` to define a newline character.

AVRO_OPTION:BYTES_TO_BASE64: boolean
: When `true`, GPSS converts Avro `bytes` fields into base64-encoded strings. The default value is `false`, GPSS does not perfrom the conversion.

CSV_OPTION
: When you specify `FORMAT: csv`, you may also provide the following options:

DELIMITER: delim_char
: Specifies a single ASCII character that separates columns within each message or row of data. The default delimiter is a comma (`,`).

QUOTE: quote_char
: Specifies the quotation character. Because GPSS does not provide a default value for this property, you must specify a value.

NULL_STRING: nullstr_val
: Specifies the string that represents the null value. Because GPSS does not provide a default value for this property, you must specify a value.

ESCAPE: escape_char
: Specifies the single character that is used for escaping data characters in the content that might otherwise be interpreted as row or column delimiters. Make sure to choose an escape character that is not used anywhere in your actual column data. Because GPSS does not provide a default value for this property, you must specify a value.

FORCE_NOT_NULL: columns
: Specifies a comma-separated list of column names to process as though each column were quoted and hence not a NULL value. For the default `null_string` (nothing between two delimiters), missing values are evaluated as zero-length strings.

FILL_MISSING_FIELDS: boolean
: Specifies the action of GPSS when it reads a row of data that has missing trailing field values (the row has missing data fields at the end of a line or row). The default value is `false`, GPSS returns an error when it encounters a row with missing trailing field values.
: If set to `true`, GPSS sets missing trailing field values to `NULL`. Blank rows, fields with a `NOT NULL` constraint, and trailing delimiters on a line will still generate an error.

NEWLINE: newline_str
: Specifies the string that represents a new line. GPSS does not specify a default value.

DELIMITED_OPTION
: When you specify `FORMAT: delimited`, you may also provide the following options:

DELIMITER: delimiter_string
: When you specify the `delimited` data format, delimiter_string is required and must identify the data element delimiter. delimiter_string may be a multi-byte value, and up to 32 bytes in length. It may not contain quote and escape characters.

EOL_PREFIX: prefix_string
: Specifies the prefix before the end of line character (`\n`) that indicates the end of a row. The default prefix is empty.

QUOTE: quote_char
: Specifies the single ASCII quotation character. The default quote character is empty.
: If you do not specify a quotation character, GPSS assumes that all columns are unquoted. If you do not specify a quotation character and do specify an escape character, GPSS assumes that all columns are unquoted and escapes the delimiter, end-of-line prefix, and escape itself.
: When you specify a quotation character, you must specify an escape character. GPSS reads any content between quote characters as-is, except for escaped characters.

ESCAPE: escape_char
: Specifies the single ASCII character used to escape special characters (for example, the delimiter, end-of-line prefix, quote, or escape itself). Therdefault escape character is empty.

When you specify an escape character and do not specify a quotation character, GPSS escapes only the delimiter, end-of-line prefix, and escape itself.
When you specify both an escape character and a quotation character, GPSS escapes only these characters.

JSONL_OPTION
Optional. When you specify `FORMAT: jsonl`, you may choose to provide the `JSONL_OPTION` properties.

NEWLINE: newline_str
A string that specifies the new line character(s) that end each JSON record. The default newline is `"\n"`.

META:
The field name, type, and format of the file meta data. `META` must specify a single `json` or `jsonb` (Greenplum 6 only) type column and `FORMAT: json`. The available meta data for a file is a single `text`-type property named `filename`. You can load this property into the target table with a `MAPPING`, or use the property in the update or merge criteria for a load operation.

FILTER: filter_string
The filter to apply to the input data before GPSS loads the data into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. If the filter evaluates to `false`, the message is dropped. filter_string must be a valid SQL conditional expression and may reference one or more `META` or `VALUE` column names.

ENCODING: char_set
The source data encoding. You can specify an encoding character set when the source data is of the `csv`, `custom`, `delimited`, or `json` format. GPSS supports the character sets identified in Character Set Support in the Greenplum Database documentation.

ERROR_LIMIT: { num_errors | percentage_errors }
The error threshold, specified as either an absolute number or a percentage. GPSS stops the load operation when this limit is reached. The default `ERROR_LIMIT` is zero; GPSS deactivates error logging and stops the load operation when it encounters the first error. Due to a limitation of the Greenplum Database external table framework, GPSS does not accept `ERROR_LIMIT: 1`.

FILE:OUTPUT: Options

SCHEMA: output_schema_name
The name of the Greenplum Database schema in which table_name resides. Optional, the default schema is the `public` schema.

TABLE: table_name
The name of the Greenplum Database table into which GPSS loads the file data.

FILTER: output_filter_string
The filter to apply to the output data before GPSS loads the data into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. If the filter evaluates to `false`, the message is dropped. output_filter_string must be a valid SQL conditional expression and may reference one or more `META` or `VALUE` column names.

MODE: mode

The table load mode. Valid mode values are `INSERT`, `MERGE`, or `UPDATE`. The default value is `INSERT`.

`UPDATE` - Updates the target table columns that are listed in `UPDATE_COLUMNS` when the input columns identified in `MATCH_COLUMNS` match the named target table columns and the optional

UPDATE_CONDITION is true.

UPDATE is not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes (" ") to identify the column.

MERGE - Inserts new rows and updates existing rows when:

- columns are listed in UPDATE_COLUMNS,
- the MATCH_COLUMNS target table column values are equal to the input data, and
- an optional UPDATE_CONDITION is specified and met.

Deletes rows when:

- the MATCH_COLUMNS target table column values are equal to the input data, and
- an optional DELETE_CONDITION is specified and met.

New rows are identified when the MATCH_COLUMNS value in the source data does not have a corresponding value in the existing data of the target table. In those cases, the entire row from the source file is inserted, not only the MATCH_COLUMNS and UPDATE_COLUMNS. If there are multiple new MATCH_COLUMNS values in the input data that are the same, GPSS inserts or updates the target table using a random matching input row. When you specify ORDER_COLUMNS, GPSS sorts the input data on the specified column(s) and inserts or updates from the input row with the largest value.

MERGE is not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes (" ") to identify the column.

MATCH_COLUMNS:

Required if MODE is MERGE or UPDATE.

match_column_name
: Specifies the column(s) to use as the join condition for the update. The attribute value in the specified target column(s) must be equal to that of the corresponding source data column(s) in order for the row to be updated in the target table.

ORDER_COLUMNS:

Optional. May be specified in MERGE MODE to sort the input data rows.

order_column_name
: Specify the column(s) by which GPSS sorts the rows. When multiple matching rows exist in a batch, ORDER_COLUMNS is used with MATCH_COLUMNS to determine the input row with the largest value; GPSS uses that row to write/update the target.

UPDATE_COLUMNS:

Required if MODE is MERGE or UPDATE.

update_column_name
: Specifies the column(s) to update for the rows that meet the MATCH_COLUMNS criteria and the optional UPDATE_CONDITION.

UPDATE_CONDITION: update_condition

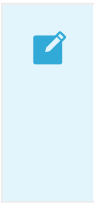
Optional. Specifies a boolean condition, similar to that which you would declare in a `WHERE` clause, that must be met in order for a row in the target table to be updated (or inserted, in the case of a `MERGE`).

DELETE_CONDITION: delete_condition

Optional. In `MERGE MODE`, specifies a boolean condition, similar to that which you would declare in a `WHERE` clause, that must be met for GPSS to delete rows in the target table that meet the `MATCH_COLUMNS` criteria.

MAPPING:

Optional. Overrides the default source-to-target column mapping. GPSS supports two mapping syntaxes.



When you specify a `MAPPING`, ensure that you provide a mapping for all data value elements of interest. GPSS does not automatically match column names when you provide a `MAPPING`.

NAME: target_column_name

Specifies the target Greenplum Database table column name.

EXPRESSION: { source_column_name | expression }

Specifies a value or meta `COLUMNS:NAME` (source_column_name) or an expression. When you specify an expression, you may provide a value expression that you would specify in the `SELECT` list of a query, such as a constant value, a column reference, an operator invocation, a built-in or user-defined function call, and so on.

target_column_name: { source_column_name | expression }

When you use this `MAPPING` syntax, specify the target_column_name and {source_column_name | expression} as described above.

FILE:TASK: Options

PREPARE_SQL: udf_or_sql_to_run

The user-defined function or SQL command(s) that you want GPSS to run before it executes the job. The default is null, no command to run.

TEARDOWN_SQL: udf_or_sql_to_run

The user-defined function or SQL command(s) that you want GPSS to run after the job stops. GPSS runs the function or command(s) on job success and job failure. The default is null, no command to run.

Job SCHEDULE: Options

SCHEDULE:

Controls the frequency and interval of restarting jobs.

RETRY_INTERVAL: retry_time

The period of time that GPSS waits before retrying a failed job. You can specify the time interval in day (`d`), hour (`h`), minute (`m`), second (`s`), or millisecond (`ms`) integer units; do not mix units. The default retry interval is `5m` (5 minutes).

MAX_RETRIES: num_retries

The maximum number of times that GPSS attempts to retry a failed job. The default is 0, do not retry. If you specify a negative value, GPSS retries the job indefinitely.

RUNNING_DURATION: run_time

The amount of time after which GPSS automatically stops a job. GPSS does not automatically stop a job by default.

AUTO_STOP_RESTART_INTERVAL: restart_time

The amount of time after which GPSS restarts a job that it stopped due to reaching `RUNNING_DURATION`.

MAX_RESTART_TIMES: num_restarts

The maximum number of times that GPSS restarts a job that it stopped due to reaching `RUNNING_DURATION`. The default is 0, do not restart the job.

QUIT_AT_EOF_AFTER: clock_time

The clock time after which GPSS stops a job every day when it encounters an EOF. By default, GPSS does not automatically stop a job that reaches EOF. GPSS never stops a job when the current time is before `clock_time`, even when GPSS encounters an EOF.

Job ALERT: Options

Controls notification when a job is stopped for any reason (success, completion, error, user-initiated stop).

COMMAND: command_to_run

The program that the GPSS server runs on the GPSS server host, including arguments. The command must be executable by GPSS.

command_to_run has access to job-related environment variables that GPSS sets, including: `$GPSSJOB_NAME`, `$GPSSJOB_STATUS`, and `$GPSSJOB_DETAIL`.

WORKDIR: directory

The working directory for command_to_run. The default working directory is the directory from which you started the GPSS server process. If you specify a relative path, it is relative to the directory from which you started the GPSS server process.

TIMEOUT: alert_time

The amount of time after a job stops, prompting GPSS to trigger the alert (and run command_to_run). You can specify the time interval in day (`d`), hour (`h`), minute (`m`), or second (`s`) integer units; do not mix units. The default alert timeout is `-1s` (no timeout).

Template Variables

GPSS supports using template variables to specify property values in the load configuration file.

You specify a template variable value in the load configuration file as follows:

```
<PROPERTY>: {{<template_var>}}
```

For example:

```
MAX_RETRIES: {{numretries}}
```

GPSS substitutes the template variable with a value that you specify via the `-p | --property <template_var=value>` option to the `gpsscli dryrun`, `gpsscli submit`, or `gpsscli load` command.

For example, if the command line specifies:

```
--property numretries=10
```

GPSS substitutes occurrences of `{{numretries}}` in the load configuration file with the value `10` before submitting the job, and uses that value while the job is running.

Notes

If you created a database object name using a double-quoted identifier (delimited identifier), you must specify the delimited name within single quotes in the `filesource.yaml` configuration file. For example, if you create a table as follows:

```
CREATE TABLE "MyTable" ("MyColumn" text);
```

Your `filesource.yaml` YAML configuration file would refer to the above table and column names as:

```
  COLUMNS:
     - name: '"MyColumn"'
       type: text

OUTPUT:
   TABLE: '"MyTable"'
```

You can specify backslash escape sequences in the CSV `DELIMITER`, `QUOTE`, and `ESCAPE` options. GPSS supports the standard backslash escape sequences for backspace, form feed, newline, carriage return, and tab, as well as escape sequences that you specify in hexadecimal format (prefaced with `\x`). Refer to Backslash Escape Sequences in the PostgreSQL documentation for more information.

Examples

Submit a job to load data from an Avro file as defined in the version 2 load configuration file named `loadfromfile.yaml`:

```
$ gpsscli submit loadfromfile.yaml
```

Example `loadfromfile.yaml` configuration file:

```
DATABASE: ops
USER: gpadmin
PASSWORD: changeme
HOST: mdw-1
PORT: 15432
VERSION: 2
FILE:
   INPUT:
      SOURCE:
         URL: file:///tmp/file.avro
      VALUE:
         COLUMNS:
           - NAME: value
             TYPE: json
         FORMAT: avro
      META:
```

```
         COLUMNS:
           - NAME: meta
             TYPE: json
         FORMAT: json
      FILTER: (value->>'x')::int < 10
      ERROR_LIMIT: 25
   OUTPUT:
      SCHEMA: gpschema
      TABLE: gptable
      MODE: INSERT
      MAPPING:
        - NAME: a
          EXPRESSION: (value->>'x')::int
        - NAME: b
          EXPRESSION: (value->>'y')::text
        - NAME: c
          EXPRESSION: (meta->>'filename')::text
SCHEDULE:
  RETRY_INTERVAL: 500ms
  MAX_RETRIES: 2
```

See Also

gpsscli load, gpsscli submit

rabbitmq-v3.yaml (Beta)

GPSS load configuration file for a RabbitMQ data source (version 3).

Synopsis

```
version: v3
```

```
targets:
- gpdb:
    host: <host>
    port: <greenplum_port>
    user: <user_name>
    password: <password>
    database: <db_name>
    work_schema: <work_schema_name>
    error_limit: <num_errors> | <percentage_errors>
    filter_expression: <filter_string>
    tables:
      - table: <table_name>
        schema: <schema_name>
        mode:
          # specify a single mode property block (described below)
          insert: {}
          update:
            <mode_specific_property>: <value>
            ...
          merge:
            <mode_specific_property>: <value>
            ...
```

```
        transformer:
          transform: <udf_transform_udf_name>
          properties:
            <udf_transform_property_name>: <property_value>
            ...
          columns:
            - <udf_transform_column_name>
            ...
        mapping:
          <target_column_name> : <source_column_name> | <expression>
          ...
        filter: <output_filter_string>
```

```
sources:
- rabbitmq:
    server: <rmq_user>:<rmq_password>@<rmq_host>:<rmq_port>
    vhost: <gpss_vhost>
    stream: <name> | queue: <name>
    consistency: strong | at-least | at-most | none
    fallback_offset: earliest | latest
    save_failing_batch: <boolean>
    recover_failing_batch: <boolean> (Beta)
    data_content:
      <data_format>:
        <column_spec>
        <other_props>
    meta:
      json:
        column:
          name: meta
          type: json
    encoding: <char_set>
    transformer:
      path: <path_to_plugin_transform_library>
      on_init: <plugin_transform_init_name>
      transform: <plugin_transform_name>
      properties:
        <plugin_transform_property_name>: <property_value>
        ...
    properties:
      <rmq_property_name>: <rmq_property_value>
      ...
    task:
      batch_size:
        max_count: <number_of_rows>
        interval_ms: <wait_time>
        idle_duration_ms: <idle_time>
      window_size: <num_batches>
      window_statement: <udf_or_sql_to_run>
      prepare_statement: <udf_or_sql_to_run>
      teardown_statement: <udf_or_sql_to_run>

option:
  schedule:
    max_retries: <num_retries>
    retry_interval: <retry_time>
    running_duration: <run_time>
    auto_stop_restart_interval: <restart_time>
```

```
    max_restart_times: <num_restarts>
    quit_at_eof_after: <clock_time>
  alert:
    command: <command_to_run>
    workdir: <directory>
    timeout: <alert_time>
```

Where the mode_specific_propertys that you can specify for `update` and `merge` mode follow:

```
update:
  match_columns: [<match_column_names>]
  order_columns: [<order_column_names>]
  update_columns: [<update_column_names>]
  update_condition: <update_condition>
```

```
merge:
  match_columns: [<match_column_names>]
  update_columns: [<update_column_names>]
  order_columns: [<order_column_names>]
  update_condition: <update_condition>
  delete_condition: <delete_condition>
```

Where data_format, column_spec, and other_props are one of the following blocks

```
binary:
  source_column_name: <column_name>
```

```
csv:
  columns:
    - name: <column_name>
      type: <column_data_type>
    ...
  delimiter: <delim_char>
  quote: <quote_char>
  null_string: <nullstr_val>
  escape: <escape_char>
  force_not_null: <columns>
  fill_missing_fields: <boolean>
```

```
custom:
  columns:
    - name: <column_name>
      type: <column_data_type>
    ...
  name: <formatter_name>
  options:
    - <optname>=<optvalue>
    ...
```

```
delimited:
  columns:
    - name: <column_name>
      type: <column_data_type>
    ...
  delimiter: <delimiter_string>
```

```
  eol_prefix: <prefix_string>
  quote: <quote_char>
  escape: <escape_char>
```

```
json:
  column:
    name: <column_name>
    type: json | jsonb
  is_jsonl: <boolean>
  newline: <newline_str>
```

And where you may specify any property value with a template variable that GPSS substitutes at runtime using the following syntax:

```
<property:> {{<template_var>}}
```

Description

Version 3 of the GPSS load configuration file is different in both content and format than previous versions of the file. Certain symbols used in the GPSS version 1 and 2 configuration file reference page syntax have different meanings in version 3 syntax:

- Brackets `[]` are literal and are used to specify a list in version 3. They are no longer used to signify the optionality of a property.
- Curly braces `{}` are literal and are used to specify YAML mappings in version 3 syntax. They are no longer used with the pipe symbol (`|`) to identify a list of choices.

You specify load configuration properties for a Greenplum Streaming Server (GPSS) RabbitMQ load job in a YAML-formatted configuration file. (This reference page uses the name `rabbitmq-v3.yaml` when referring to this file; you may choose your own name for the file.) Load properties include Greenplum Database connection and data import properties, RabbitMQ data source information, and properies specific to the GPSS job.

The `gpsscli` utilities process the YAML configuration file in order, using indentation (spaces) to determine the document hierarchy and the relationships between the sections. The use of white space in the file is significant. Keywords are not case-sensitive.

Keywords and Values

version Property

version: v3
: The version of the configuration file. You must specify `version: v3`.

targets:gpdb Properties

host: host

The host name or IP address of the Greenplum Database coordinator host.

port: greenplum_port

The port number of the Greenplum Database server on the coordinator host.

user: user_name

The name of the Greenplum Database user/role. This user_name must have permissions as described in the Configuring Greenplum Database Role Privileges.

password: password

The password for the Greenplum Database user/role.

database: db_name

The name of the Greenplum database.

work_schema: work_schema_name

The name of the Greenplum Database schema in which GPSS creates internal tables. The default work_schema_name is `public`.

error_limit: num_errors | percentage_errors

The error threshold, specified as either an absolute number or a percentage. GPSS stops running the job when this limit is reached.

filter_expression: filter_string

The filter to apply to the input data before GPSS loads the data into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. If the filter evaluates to `false`, the message is dropped. filter_string must be a valid SQL conditional expression and may reference one or more source value or meta column names.

tables:

The Greenplum Database tables, and the data that GPSS will load into each.

GPSS supports loading from a RabbitMQ data source into a single Greenplum Database table only.

table: table_name

The name of the Greenplum Database table into which GPSS loads the data.

schema: schema_name

The name of the Greenplum Database schema in which table_name resides. Optional, the default schema is the `public` schema.

mode:

The table load mode; `insert`, `merge`, or `update`. The default mode is `insert`.

`update` and `merge` are not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes (" ") to identify the column.

insert:

Inserts source data into Greenplum.

update:

Updates the target table columns that are listed in `update_columns` when the input columns identified in `match_columns` match the named target table columns and the optional

`update_condition` is true.

merge:

Inserts new rows and updates existing rows when:

- columns are listed in `update_columns`,
- the `match_columns` target table column values are equal to the input data, and
- an optional `update_condition` is specified and met.

Deletes rows when:

- the `match_columns` target table column values are equal to the input data, and
- an optional `delete_condition` is specified and met.

New rows are identified when the `match_columns` value in the source data does not have a corresponding value in the existing data of the target table. In those cases, the entire row from the source file is inserted, not only the `match_columns` and `update_columns`. If there are multiple new `match_columns` values in the input data that are the same, GPSS inserts or updates the target table using a random matching input row. When you specify `order_columns`, GPSS sorts the input data on the specified column(s) and inserts or updates from the input row with the largest value.

mode_property_name: value

The name to value mapping for a mode property. Each `mode` supports one or more of the following properties as specified in the Synopsis.

match_columns: [match_column_names]

A comma-separated list that specifies the column(s) to use as the join condition for the update. The attribute value in the specified target column(s) must be equal to that of the corresponding source data column(s) in order for the row to be updated in the target table. Required when `mode` is `merge` or `update`.

order_columns: [order_column_names]

A comma-separated list that specifies the column(s) by which GPSS sorts the rows. When multiple matching rows exist in a batch, `order_columns` is used with `match_columns` to determine the input row with the largest value; GPSS uses that row to write/update the target. Optional. May be specified in `merge mode` to sort the input data rows.

update_columns: [update_column_names]

A column-sparated list that specifies the column(s) to update for the rows that meet the `match_columns` criteria and the optional `update_condition`.

Required when `mode` is `merge` or `update`.

update_condition: update_condition

Specifies a boolean condition, similar to that which you would declare in a `WHERE` clause, that must be met in order for a row in the target table to be updated (or inserted, in the case of a `merge`). Optional.

delete_condition: delete_condition

In `merge mode`, specifies a boolean condition, similar to that which you would declare in a `WHERE` clause, that must be met for GPSS to delete rows in the target table that meet the `match_columns` criteria. Optional.

transformer:

Optional. Output data transform block. An output data transformer is a user-defined function (UDF) that transforms the data before it is loaded into Greenplum Database. The semantics of the UDF are transform-specific.

> GPSS currently supports specifying only one of the `mapping` or (UDF) `transformer` blocks in the load configuration file, not both.

transform: udf_transform_udf_name
: The name of the output transform UDF. GPSS invokes this function for every batch of data it writes to Greenplum Database.

properties: udf_transform_property_name: property_value
: One or more property name and value pairs that GPSS passes to udf_transform_udf_name.

columns: output_transform_column_name
: The name of one or more columns involved in the transform.

mapping:

Optional. Overrides the default source-to-target column mapping.

> GPSS currently supports specifying only one of the `mapping` or (UDF) `transformer` blocks in the load configuration file, not both.

> When you specify a `mapping`, ensure that you provide a mapping for all source data elements of interest. GPSS does not automatically match column names when you provide a `mapping` block.

target_column_name: source_column_name | expression
: target_column_name specifies the target Greenplum Database table column name. GPSS maps this column name to the source column name specified in source_column_name, or to an expression. When you specify an expression, you may provide a value expression that you would specify in the `SELECT` list of a query, such as a constant value, a column reference, an operator invocation, a built-in or user-defined function call, and so on.

filter: output_filter_string
: The filter to apply to the output data before GPSS loads the data into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. If the filter evaluates to `false`, the message is dropped. output_filter_string must be a valid SQL conditional expression and may reference one or more `META` or `VALUE` column names.

sources:rabbitmq: Options

server: rmq_user:rmq_password@rmq_host:rmq_port
: The RabbitMQ server connection string; includes the user name with which RabbitMQ logs in to the broker, the password for rmq_user, the hostname or IP address of the RabbitMQ server, and the port number on which the RabbitMQ server is listening. rmq_user and rmq_password are optional, and must be omitted when loading from a RabbitMQ queue over a TLS-encrypted connection.

vhost: gpss_vhost
: The RabbitMQ virtual host that represents the GPSS server.

stream: name
: The name of the RabbitMQ stream from which to read the data. You may specify only one of `stream` or `queue`.

queue: name
: The name of the RabbitMQ queue from which to read the data. You may specify only one of `stream` or `queue`.

consistency: strong | at-least | at-most | none
: Specify how GPSS should manage message offsets when it acts as a consumer of a RabbitMQ queue or stream. Valid values are `at-least` (GPSS stores the offsets before commit), `at-most` (GPSS stores the offsets after commit), and `none`. For streams, GPSS also supports `strong` consistency. The default value is `at-least`. Refer to Understanding RabbitMQ Message Offset Management for more detailed information.

fallback_offset: earliest | latest
: When reading from a RabbitMQ *stream*, specifies the behaviour of GPSS when it detects a message offset gap. When set to `earliest`, GPSS automatically resumes a load operation from the earliest available published message. When set to `latest`, GPSS loads only new messages to the RabbitMQ stream.

save_failing_batch: boolean
: Determines whether or not GPSS saves data into a backup table before it writes the data to Greenplum Database. Saving the data in this manner aids recovery when GPSS encounters errors during the evaluation of expressions. The default is `false`; GPSS does not use a backup table, and returns immediately when it encounters an expression error. When you set this property to `true`, GPSS writes both the good and the bad data in the batch to a backup table named `gpssbackup_<jobhash>`, and continues to process incoming data. You must then manually load the good data from the backup table into Greenplum **or** set `recover_failing_batch` (Beta) to `true` to have GPSS automatically reload the good data.

Using a backup table to hedge against mapping errors may impact performance, especially when the data that you are loading has not been cleaned.

recover_failing_batch: boolean (Beta)
: When set to `true` and `save_failing_batch` is also `true`, GPSS automatically reloads the good data in the batch and retains only the error data in the backup table. The default value is `false`; GPSS does not process the backup table.

Enabling this property requires that GPSS has the Greenplum Database privileges to create a function.

data_content:
: The RabbitMQ message value data type, field names, and type-specific properties. You must specify all RabbitMQ data elements in the order in which they appear in the RabbitMQ message.

column_spec

The source to Greenplum column mapping. The supported column specification differs for different data formats as described below.

The default source-to-target data mapping behaviour of GPSS is to match a column name as defined in `source_column_name`, `column:name`, or `columns:name` with a column name in the target Greenplum Database `table`. You can override the default mapping by specifying a `mapping:` block.

data_format
: The format of the value data. You may specify a data_format of `binary`, `csv`, `custom`, `delimited`, or `json` for the value, with some restrictions.

binary
: When you specify the `binary` data format, GPSS reads the data into a single `bytea`-type column.

source_column_name: column_name
: The name of the single `bytea`-type column into which GPSS reads the value data.

csv
: When you specify the `csv` data format, GPSS reads the data into the list of columns that you specify. The message content cannot contain line ending characters (CR and LF).

columns:
: A set of column name/type mappings. The value `[]` specifies all columns.

name: column_name
: The name of a value column. column_name must match the column name of the target Greenplum Database table.

type: column_data_type
: The data type of the column. You must specify an equivalent data type for each data element and the associated Greenplum Database table column.

delimiter: delim_char
: Specifies a single ASCII character that separates columns within each message or row of data. The default delimiter is a comma (`,`).

quote: quote_char
: Specifies the quotation character. Because GPSS does not provide a default value for this property, you must specify a value.

null_string: nullstr_val
: Specifies the string that represents the null value. Because GPSS does not provide a default value for this property, you must specify a value.

escape: escape_char
: Specifies the single character that is used for escaping data characters in the content that might otherwise be interpreted as row or column delimiters. Make sure to choose an escape character that is not used anywhere in your actual column data. Because GPSS does not provide a default value for this property, you must specify a value.

force_not_null: columns
: Specifies a comma-separated list of column names to process as though each column were quoted and hence not a NULL value. For the default `null_string` (nothing between two delimiters), missing values are evaluated as zero-length strings.

fill_missing_fields: boolean
: Specifies the action of GPSS when it reads a row of data that has missing trailing field values (the row has missing data fields at the end of a line or row). The default value is `false`, GPSS returns an error when it encounters a row with missing trailing field values.

If set to `true`, GPSS sets missing trailing field values to `NULL`. Blank rows, fields with a `NOT NULL` constraint, and trailing delimiters on a line will still generate an error.

custom
: When you specify the `custom` data format, GPSS uses the custom formatter that you specify to process the input data before writing it to Greenplum Database.

columns:
: A set of column name/type mappings. The value `[]` specifies all columns.

name: column_name
: The name of a key or value column. column_name must match the column name of the target Greenplum Database table.

type: column_data_type
: The data type of the column. You must specify an equivalent data type for each data element and the associated Greenplum Database table column.

name: formatter_name
: When you specify the `custom` data format, formatter_name is required and must identify the name of the formatter user-defined function that GPSS should use when loading the data.

options:
: A set of function argument name=value pairs.

optname=optvalue
: The name and value of the set of arguments to pass into the formatter_name UDF.

delimited
: When you specify the `delimited` data format, GPSS reads the data into the list of columns that you specify. You must specify the data `delimiter`.

columns:
: A set of column name/type mappings. The value `[]` specifies all columns.

name: column_name
: The name of a value column. column_name must match the column name of the target Greenplum Database table.

type: column_data_type
: The data type of the column. You must specify an equivalent data type for each data element and the associated Greenplum Database table column.

delimiter: delimiter_string
: When you specify the `delimited` data format, delimiter_string is required and must identify the data element delimiter. delimiter_string may be a multi-byte value, and up to 32 bytes in length. It may not contain quote and escape characters.

eol_prefix: prefix_string
: Specifies the prefix before the end of line character (`\n`) that indicates the end of a row. The default prefix is empty.

quote: quote_char
: Specifies the single ASCII quotation character. The default quote character is empty.
: If you do not specify a quotation character, GPSS assumes that all columns are unquoted. If you do not specify a quotation character and do specify an escape character, GPSS assumes that all columns are unquoted and escapes the delimiter, end-of-line prefix, and escape itself.
: When you specify a quotation character, you must specify an escape character. GPSS reads any content between quote characters as-is, except for escaped characters.

escape: escape_char

Specifies the single ASCII character used to escape special characters (for example, the `delimiter`, `eol_prefix`, `quote`, or `escape` itself). Therdefault escape character is empty.
When you specify an escape character and do not specify a quotation character, GPSS escapes only the delimiter, end-of-line prefix, and escape itself.
When you specify both an escape character and a quotation character, GPSS escapes only these characters.

json
When you specify the `json` data format, GPSS can read the data as a single JSON object or as a single JSON record per line.

column:
A single column name/type mapping.

name: column_name
The name of the key or value column. column_name must match the column name of the target Greenplum Database table.

type: json | jsonb | gp_jsonb (Beta) | gp_json (Beta)
The data type of the column.

is_jsonl: boolean
Identifies whether or not GPSS reads the JSON data as a single object or single-record-per-line. The default is `false`, GPSS reads the JSON data as a single object.

newline: newline_str
A string that specifies the new line character(s) that end each JSON record. The default newline is `"\n"`.

meta:
The data type and field name of the RabbitMQ meta data. `meta:` must specify the `json` or `jsonb` (Greenplum 6 only) data format, and a single `json`-type column.

The available RabbitMQ meta data properties for a streaming source include:

- `stream` (text) - the RabbitMQ stream name
- `offset` (bigint) - the message offset

The available RabbitMQ meta data properties for a queue source include:

- `queue` (text) - the RabbitMQ queue name
- `messageId` (text) - the message identifier
- `correlationId` (text) - the correlation identifier
- `timestamp` (bitint) - the time that the message was added to the RabbitMQ queue

You can load any of these properties into the target table with a `mapping`, or use a property in the update or merge criteria for a load operation.

encoding: char_set
The source data encoding. You can specify an encoding character set when the source data is of the `csv`, `custom`, `delimited`, or `json` format. GPSS supports the character sets identified in Character Set Support in the Tanzu Greenplum documentation.

transformer:

Input data transform block. An input data transformer is a plugin, a set of `go` functions that transform the data after it is read from the source. The semantics of the transform are function-specific. You

specify the library and function names in this block, as well as the properties that GPSS passes to these functions:

path: path_to_plugin_transform_library
: The file system location of the plugin transformer library on the Greenplum Streaming Server server host.

on_init: plugin_transform_init_name
: The name of an initialization function that GPSS calls when it loads the transform library.

transform: plugin_transform_name
: The name of the transform function. GPSS invokes this function for every message it reads.

properties: plugin_transform_property_name: property_value
: One or more property name and value pairs that GPSS passes to plugin_transform_init_name and plugin_transform_name.

properties:

RabbitMQ configuration property names and values.

rmq_property_name
: The name of a RabbitMQ property.

rmq_property_value
: The RabbitMQ property value.

task:

The batch size and commit window.

batch_size:
: Controls how GPSS commits data to Greenplum Database. You may specify both `max_count` and `interval_ms` as long as both values are not zero (`0`). Try setting and tuning `interval_ms` to your environment; introduce a `max_count` setting only if you encounter high memory usage associated with message buffering.

max_count: number_of_rows
: The number of rows to batch before triggering an `INSERT` operation on the Greenplum Database table. The default value of `max_count` is `0`, which instructs GPSS to ignore this commit trigger condition.

interval_ms: wait_time
: The minimum amount of time to wait (milliseconds) between each `INSERT` operation on the table. The default value is `5000`.

idle_duration_ms: idle_time
: The maximum amount of time to wait (milliseconds) for the first batch of data. When you use this property to enable lazy load, GPSS waits until RabbitMQ data is available before locking the target Greenplum table. You can specify:

  - `0` (lazy load is deactivated)
  - `-1` (lazy load is activated, the job never stops), or
  - a positive value (lazy load is activated, the job stops after idle_time duration of no data in the RabbitMQ queue or stream)

  The default value is `0`.

window_size: num_batches
: The number of batches to read before running `window_statement`. The default batch interval is 0.

window_statement: udf_or_sql_to_run
: A user-defined function or SQL command(s) that you want to run after GPSS reads `window_size` number of batches. The default is null, no command to run.

prepare_statement: udf_or_sql_to_run
: A user-defined function or SQL command(s) that you want GPSS to run before it executes the job. The default is null, no command to run.

teardown_statement: udf_or_sql_to_run
: A user-defined function or SQL command(s) that you want GPSS to run after the job stops. GPSS runs the function or command(s) on job success and job failure. The default is null, no command to run.

option: Properties

schedule:

Controls the frequency and interval of restarting jobs.

retry_interval: retry_time
: The period of time that GPSS waits before retrying a failed job. You can specify the time interval in day (`d`), hour (`h`), minute (`m`), second (`s`), or millisecond (`ms`) integer units; do not mix units. The default retry interval is `5m` (5 minutes).

max_retries: num_retries
: The maximum number of times that GPSS attempts to retry a failed job. The default is 0, do not retry. If you specify a negative value, GPSS retries the job indefinitely.

running_duration: run_time
: The amount of time after which GPSS automatically stops a job. GPSS does not automatically stop a job by default.

auto_stop_restart_interval: restart_time
: The amount of time after which GPSS restarts a job that it stopped due to reaching `running_duration`.

max_restart_times: num_restarts
: The maximum number of times that GPSS restarts a job that it stopped due to reaching `running_duration`. The default is 0, do not restart the job.

quit_at_eof_after: clock_time
: The clock time after which GPSS stops a job every day when it encounters an EOF. By default, GPSS does not automatically stop a job that reaches EOF. GPSS never stops a job when the current time is before `clock_time`, even when GPSS encounters an EOF.

alert:

Controls notification when a job is stopped for any reason (success, completion, error, user-initiated stop).

command: command_to_run
: The program that the GPSS server runs on the GPSS server host, including arguments. The command must be executable by GPSS.

command_to_run has access to job-related environment variables that GPSS sets, including: `$GPSSJOB_NAME`, `$GPSSJOB_STATUS`, and `$GPSSJOB_DETAIL`.

workdir: directory

The working directory for command_to_run. The default working directory is the directory from which you started the GPSS server process. If you specify a relative path, it is relative to the directory from which you started the GPSS server process.

timeout: alert_time

The amount of time after a job stops, prompting GPSS to trigger the alert (and run command_to_run). You can specify the time interval in day (`d`), hour (`h`), minute (`m`), or second (`s`) integer units; do not mix units. The default alert timeout is `-1s` (no timeout).

Template Variables

GPSS supports using template variables to specify property values in the load configuration file.

You specify a template variable value in the load configuration file as follows:

```
<property>: {{<template_var>}}
```

For example:

```
max_retries: {{numretries}}
```

GPSS substitutes the template variable with a value that you specify via the `-p | --property <template_var=value>` option to the `gpsscli dryrun`, `gpsscli submit`, or `gpsscli load` command.

For example, if the command line specifies:

```
--property numretries=10
```

GPSS substitutes occurrences of `{{numretries}}` in the load configuration file with the value `10` before submitting the job, and uses that value while the job is running.

Notes

If you created a database object name using a double-quoted identifier (delimited identifier), you must specify the delimited name within single quotes in the load configuration file. For example, if you create a table as follows:

```
CREATE TABLE "MyTable" (c1 text);
```

Your YAML configuration file would refer to the table name as:

```
targets:
- gpdb:
    tables:
      - table: '"MyTable"'
```

You can specify backslash escape sequences in the CSV `delimiter`, `quote`, and `escape` options. GPSS supports the standard backslash escape sequences for backspace, form feed, newline, carriage return, and

tab, as well as escape sequences that you specify in hexadecimal format (prefaced with `\x`). Refer to Backslash Escape Sequences in the PostgreSQL documentation for more information.

Examples

Load data from RabbitMQ as defined in the Version 3 configuration file named `loadfromrmq_v3.yaml`:

```
gpsscli load loadfromrmq_v3.yaml
```

Example `loadfromrmq_v3.yaml` configuration file:

```
version: v3
targets:
- gpdb:
    host: mdw-1
    port: 15432
    user: gpadmin
    password: changeme
    database: testdb
    work_schema: public
    error_limit: 25
    tables:
      - table: tbl1
        schema: public
        mode:
          insert {}

sources:
- rabbitmq:
    server: gpadmin:changeme@mdw-1:5672
    queue: test
    vhost: gpadmin
    data_content:
      csv:
        columns: []
        delimiter: ","
        quote: "'"
        null_string: "NA"
        escape: '\'
        force_not_null: "c1,c2"
        fill_missing_fields: true
    properties:
      eof.when.idle: 1500

option:
  schedule: {}
```

See Also

gpsscli-v3.yaml submit, gpsscli submit, gpsscli load

rabbitmq-v2.yaml

GPSS load configuration file for a RabbitMQ data source (version 2).

Synopsis

```
DATABASE: <db_name>
USER: <user_name>
PASSWORD: <password>
HOST: <host>
PORT: <greenplum_port>
VERSION: 2
RABBITMQ:
   INPUT:
      SOURCE:
        SERVER: <rmq_user>:<rmq_password>@<rmq_host>:<rmq_port>
        VIRTUALHOST: <gpss_vhost>
        { STREAM: <name> | QUEUE: <name> }
        [FALLBACK_OFFSET: { earliest | latest }]
      DATA:
        COLUMNS:
           - NAME: { <column_name> | __IGNORED__ }
             TYPE: <column_data_type>
           [ ... ]
         FORMAT: <value_data_format>
         [[DELIMITED_OPTION:
            DELIMITER: <delimiter_string>
            [EOL_PREFIX: <prefix_string>]
            [QUOTE: <quote_char>]
            [ESCAPE: <escape_char>] ] |
         [CSV_OPTION:
            [DELIMITER: <delim_char>]
            [QUOTE: <quote_char>]
            [NULL_STRING: <nullstr_val>]
            [ESCAPE: <escape_char>]
            [FORCE_NOT_NULL: <columns>]
            [FILL_MISSING_FIELDS: <boolean>]] |
         [JSONL_OPTION:
            [NEWLINE: <newline_str>]] |
         [CUSTOM_OPTION:
            NAME: <udf_name>
            PARAMSTR: <udf_parameter_string>]]
      [META:
         COLUMNS:
            - NAME: <meta_column_name>
              TYPE: { json | jsonb }
         FORMAT: json]
      [TRANSFORMER:
         PATH: <path_to_plugin_transform_library>
         ON_INIT: <plugin_transform_init_name>
         TRANSFORM: <plugin_transform_name>
         PROPERTIES:
           <plugin_transform_property_name>: <property_value>
           [ ... ] ]
      [FILTER: <filter_string>]
      [ENCODING: <char_set>]
      [ERROR_LIMIT: { <num_errors> | <percentage_errors> }]
   OUTPUT:
      [SCHEMA: <output_schema_name>]
      TABLE: <table_name>
      [FILTER: <output_filter_string>]
      [MODE: <mode>]
```

```
      [MATCH_COLUMNS:
         - <match_column_name>
         [ ... ]]
      [ORDER_COLUMNS:
         - <order_column_name>
         [ ... ]]
      [UPDATE_COLUMNS:
         - <update_column_name>
         [ ... ]]
      [UPDATE_CONDITION: <update_condition>]
      [DELETE_CONDITION: <delete_condition>]
      [TRANSFORMER:
         TRANSFORM: <udf_transform_udf_name>
         PROPERTIES:
           <udf_transform_property_name>: <property_value>
           [ ... ]
         COLUMNS:
           - <udf_transform_column_name>
           [ ... ] ]
      [MAPPING:
         - NAME: <target_column_name>
           EXPRESSION: { <source_column_name> | <expression> }
         [ ... ]
           |
         <target_column_name> : { <source_column_name> | <expression> }
         [ ... ] ]
   [METADATA:
      [SCHEMA: <metadata_schema_name>]]
   [COMMIT:
      SAVE_FAILING_BATCH: <boolean>
      RECOVER_FAILING_BATCH: <boolean> (Beta)
      MAX_ROW: <num_rows>
      MINIMAL_INTERVAL: <wait_time>
      CONSISTENCY: { strong | at-least | at-most | none }
      IDLE_DURATION: <idle_time> ]
   [TASK:
      POST_BATCH_SQL: <udf_or_sql_to_run>
      BATCH_INTERVAL: <num_batches>
      PREPARE_SQL: <udf_or_sql_to_run>
      TEARDOWN_SQL: <udf_or_sql_to_run> ]
   [PROPERTIES:
      <rmq_property_name>: <rmq_property_value>
      [ ... ]]
[SCHEDULE:
   RETRY_INTERVAL: <retry_time>
   MAX_RETRIES: <num_retries>
   RUNNING_DURATION: <run_time>
   AUTO_STOP_RESTART_INTERVAL: <restart_time>
   MAX_RESTART_TIMES: <num_restarts>
   QUIT_AT_EOF_AFTER: <clock_time>]
[ALERT:
   COMMAND: <command_to_run>
   WORKDIR: <directory>
   TIMEOUT: <alert_time>]
```

Where you may specify any property value with a template variable that GPSS substitutes at runtime using the following syntax:

```
<PROPERTY:> {{<template_var>}}
```

Description

You specify load configuration parameters for the `gpsscli` utilities in a YAML-formatted configuration file. (This reference page uses the name `rabbitmq-v2.yaml` when referring to this file; you may choose your own name for the file.) Load parameters include Greenplum Database connection and target table information, RabbitMQ data source information, and error and commit thresholds.

The `gpsscli` utility processes the YAML configuration file in order, using indentation (spaces) to determine the document hierarchy and the relationships between the sections. The use of white space in the file is significant, and keywords are case-sensitive.

Keywords and Values

Greenplum Database Options

DATABASE: db_name
: The name of the Greenplum database.

USER: user_name
: The name of the Greenplum Database user/role. This user_name must have permissions as described in the Greenplum Streaming Server documentation.

PASSWORD: password
: The password for the Greenplum Database user/role.

HOST: host
: The host name or IP address of the Greenplum Database coordinator host.

PORT: greenplum_port
: The port number of the Greenplum Database server on the coordinator host.

VERSION: 2
: The version of the GPSS configuration file. You must specify `VERSION: 2` when you configure the `DATA` block in the file.

RABBITMQ:INPUT: Options

SOURCE
: RabbitMQ input configuration parameters.

  SERVER: rmq_user:rmq_password@rmq_host:rmq_port
  : The RabbitMQ server connection string; includes the user name with which RabbitMQ logs in to the broker, the password for rmq_user, the hostname or IP address of the RabbitMQ server, and the port number on which the RabbitMQ server is listening. rmq_user and rmq_password are optional, and must be omitted when loading from a RabbitMQ queue over a TLS-encrypted connection.

  VIRTUALHOST: gpss_vhost
  : The RabbitMQ virtual host that represents the GPSS server.

  STREAM: name
  : The name of the RabbitMQ stream from which to read the data. You may specify only one of `STREAM` or `QUEUE`.

QUEUE: name

The name of the RabbitMQ queue from which to read the data. You may specify only one of `STREAM` or `QUEUE`.

FALLBACK_OFFSET: { earliest | latest }

When reading from a RabbitMQ *stream*, specifies the behaviour of GPSS when it detects a message offset gap. When set to `earliest`, GPSS automatically resumes a load operation from the earliest available published message. When set to `latest`, GPSS loads only new messages to the RabbitMQ stream.

DATA:

The RabbitMQ message value field names, data types, and format. You must specify all RabbitMQ data elements in the order in which they appear in the RabbitMQ message.

COLUMNS:NAME: column_name

The name of a data column. column_name must match the column name of the target Greenplum Database table. Specify `__IGNORED__` to omit this RabbitMQ message data element from the load operation.

The default source-to-target data mapping behaviour of GPSS is to match a column name as defined in `COLUMNS:NAME` with a column name in the target Greenplum Database `TABLE`. You can override the default mapping by specifying a `MAPPING` block.

COLUMNS:TYPE: data_type

The data type of the column. You must specify an equivalent data type for each non-ignored RabbitMQ message data element and the associated Greenplum Database table column.

FORMAT: data_format

The format of the RabbitMQ message data. You may specify a `FORMAT` of `binary`, `csv`, `custom`, `delimited`, `json`, or `jsonl` for the data, with some restrictions.

binary

When you specify the `binary` data format, you must define only a single `bytea` type column in `COLUMNS`.

csv

When you specify the `csv` data format, the message content cannot contain line ending characters (CR and LF).

custom

When you specify the `custom` data format, you must provide a `CUSTOM_OPTION`.

delimited

When you specify the `delimited` data format, you must provide a `DELIMITED_OPTION`.

json

When you specify the `json` data format, you must define only a single `json` type column in `COLUMNS`.

jsonl

When you specify the `jsonl` data format, you may provide a `JSONL_OPTION` to define a newline character.

CSV_OPTION

When you specify `FORMAT: csv`, you may provide the following options:

DELIMITER: delim_char

Specifies a single ASCII character that separates columns within each message or row of data. The default delimiter is a comma (`,`).

QUOTE: quote_char
: Specifies the quotation character. Because GPSS does not provide a default value for this property, you must specify a value.

NULL_STRING: nullstr_val
: Specifies the string that represents the null value. Because GPSS does not specify a default value for this property, you must specify a value.

ESCAPE: escape_char
: Specifies the single character that is used for escaping data characters in the content that might otherwise be interpreted as row or column delimiters. Make sure to choose an escape character that is not used anywhere in your actual column data. Because GPSS does not provide a default value for this property. you must specify a value.

FORCE_NOT_NULL: columns
: Specifies a comma-separated list of column names to process as though each column were quoted and hence not a NULL value. For the default `null_string` (nothing between two delimiters), missing values are evaluated as zero-length strings.

FILL_MISSING_FIELDS: boolean
: Specifies the action of GPSS when it reads a row of data that has missing trailing field values (the row has missing data fields at the end of a line or row). The default value is `false`, GPSS returns an error when it encounters a row with missing trailing field values.
: If set to `true`, GPSS sets missing trailing field values to `NULL`. Blank rows, fields with a `NOT NULL` constraint, and trailing delimiters on a line will still generate an error.

CUSTOM_OPTION

Optional. When you specify `FORMAT: custom`, you are required to provide the `CUSTOM_OPTION` properties. This block identifies the name and the arguments of a custom formatter user-defined function.

NAME: udf_name
: The name of the custom formatter user-defined function.

PARAMSTR: udf_parameter_string
: A string specifying the comma-separated list of arguments to pass to the custom formatter user-defined function.

JSONL_OPTION

Optional. When you specify `FORMAT: jsonl`, you may choose to provide the `JSONL_OPTION` properties.

NEWLINE: newline_str
: A string that specifies the new line character(s) that end each JSON record. The default newline is `"\n"`.

DELIMITED_OPTION

Optional. When you specify `FORMAT: delimited`, you may choose to provide the `DELIMITER_OPTION` properties.

DELIMITER: delimiter_string

When you specify the `delimited` format, delimiter_string is required and must identify the data element delimiter. delimiter_string may be a multi-byte value, and up to 32 bytes in length. It may not contain quote and escape characters.

EOL_PREFIX: prefix_string

Specifies the prefix before the end of line character (`\n`) that indicates the end of a row. The default prefix is empty.

QUOTE: quote_char

Specifies the single ASCII quotation character. The default quote character is empty.
If you do not specify a quotation character, GPSS assumes that all columns are unquoted. If you do not specify a quotation character and do specify an escape character, GPSS assumes that all columns are unquoted and escapes the delimiter, end-of-line prefix, and escape itself.
When you specify a quotation character, you must specify an escape character. GPSS reads any content between quote characters as-is, except for escaped characters.

ESCAPE: escape_char

Specifies the single ASCII character used to escape special characters (for example, the delimiter, end-of-line prefix, quote, or escape itself). Therdefault escape character is empty.
When you specify an escape character and do not specify a quotation character, GPSS escapes only the delimiter, end-of-line prefix, and escape itself.
When you specify both an escape character and a quotation character, GPSS escapes only these characters.

META:

The field name, type, and format of the RabbitMQ meta data. META must specify a single `json` or `jsonb` (Greenplum 6 only) type column and `FORMAT: json`.

The available RabbitMQ meta data properties for a streaming source include:

- `stream` (text) - the RabbitMQ stream name
- `offset` (bigint) - the message offset

The available RabbitMQ meta data properties for a queue source include:

- `queue` (text) - the RabbitMQ queue name
- `messageId` (text) - the message identifier
- `correlationId` (text) - the correlation identifier
- `timestamp` (bitint) - the time that the message was added to the RabbitMQ queue

TRANSFORMER:

Input data transform block. An input data transformer is a plugin, a set of `go` functions that transform the data after it is read from the source. The semantics of the transform are function-specific. You specify the library and function names in this block, as well as the properties that GPSS passes to these functions:

PATH: path_to_plugin_transform_library

The file system location of the plugin transformer library on the Greenplum Streaming Server server host.

ON_INIT: plugin_transform_init_name

The name of an initialization function that GPSS calls when it loads the transform library.

TRANSFORM: plugin_transform_name

The name of the transform function. GPSS invokes this function for every message it reads.

PROPERTIES: plugin_transform_property_name: property_value

One or more property name and value pairs that GPSS passes to plugin_transform_init_name and plugin_transform_name.

FILTER: filter_string

The filter to apply to the RabbitMQ input messages before GPSS loads the data into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. If the filter evaluates to `false`, the message is dropped. filter_string must be a valid SQL conditional expression and may reference one or more `DATA` column names.

`ENCODING: <char_set>`

The source data encoding. You can specify an encoding character set when the source data is of the `csv`, `custom`, `delimited`, or `json` format. GPSS supports the character sets identified in Character Set Support in the Tanzu Greenplum documentation.

ERROR_LIMIT: { num_errors | percentage_errors }

The error threshold, specified as either an absolute number or a percentage. `gpsscli load` exits when this limit is reached. The default `ERROR_LIMIT` is zero; GPSS deactivates error logging and stops the load operation when it encounters the first error. Due to a limitation of the Greenplum Database external table framework, GPSS does not accept `ERROR_LIMIT: 1`.

RABBITMQ:OUTPUT: Options

SCHEMA: output_schema_name

The name of the Greenplum Database schema in which table_name resides. Optional, the default schema is the `public` schema.

TABLE: table_name

The name of the Greenplum Database table into which GPSS loads the RabbitMQ data.

FILTER: output_filter_string

The filter to apply to the output data before GPSS loads the data into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. If the filter evaluates to `false`, the message is dropped. output_filter_string must be a valid SQL conditional expression and may reference one or more `META` or `VALUE` column names.

MODE: mode

The table load mode. Valid mode values are `INSERT`, `MERGE`, or `UPDATE`. The default value is `INSERT`.

`UPDATE` - Updates the target table columns that are listed in `UPDATE_COLUMNS` when the input columns identified in `MATCH_COLUMNS` match the named target table columns and the optional `UPDATE_CONDITION` is true.

`UPDATE` is not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes (" ") to identify the column.

`MERGE` - Inserts new rows and updates existing rows when:

- columns are listed in `UPDATE_COLUMNS`,
- the `MATCH_COLUMNS` target table column values are equal to the input data, and
- an optional `UPDATE_CONDITION` is specified and met.

Deletes rows when:

- the MATCH_COLUMNS target table column values are equal to the input data, and
- an optional DELETE_CONDITION is specified and met.

New rows are identified when the MATCH_COLUMNS value in the source data does not have a corresponding value in the existing data of the target table. In those cases, the entire row from the source file is inserted, not only the MATCH_COLUMNS and UPDATE_COLUMNS. If there are multiple new MATCH_COLUMNS values in the input data that are the same, GPSS inserts or updates the target table using a random matching input row. When you specify ORDER_COLUMNS, GPSS sorts the input data on the specified column(s) and inserts or updates from the input row with the largest value. MERGE is not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes (" ") to identify the column.

MATCH_COLUMNS:

Required if MODE is MERGE or UPDATE.

match_column_name
: Specifies the column(s) to use as the join condition for the update. The attribute value in the specified target column(s) must be equal to that of the corresponding source data column(s) in order for the row to be updated in the target table.

ORDER_COLUMNS:

Optional. May be specified in MERGE MODE to sort the input data rows.

order_column_name
: Specify the column(s) by which GPSS sorts the rows. When multiple matching rows exist in a batch, ORDER_COLUMNS is used with MATCH_COLUMNS to determine the input row with the largest value; GPSS uses that row to write/update the target.

UPDATE_COLUMNS:

Required if MODE is MERGE or UPDATE.

update_column_name
: Specifies the column(s) to update for the rows that meet the MATCH_COLUMNS criteria and the optional UPDATE_CONDITION.

UPDATE_CONDITION: update_condition
: Optional. Specifies a boolean condition, similar to that which you would declare in a WHERE clause, that must be met in order for a row in the target table to be updated (or inserted, in the case of a MERGE).

DELETE_CONDITION: delete_condition
: Optional. In MERGE MODE, specifies a boolean condition, similar to that which you would declare in a WHERE clause, that must be met for GPSS to delete rows in the target table that meet the MATCH_COLUMNS criteria.

TRANSFORMER:

Optional. Output data transform block. An output data transformer is a user-defined function (UDF) that transforms the data before it is loaded into Greenplum Database. The semantics of the UDF are transform-specific.

GPSS currently supports specifying only one of the `MAPPING` or (UDF) `TRANSFORMER` blocks in the load configuration file, not both.

TRANSFORM: udf_transform_udf_name
: The name of the output transform UDF. GPSS invokes this function for every batch of data it writes to Greenplum Database.

PROPERTIES: udf_transform_property_name: property_value
: One or more property name and value pairs that GPSS passes to udf_transform_udf_name.

COLUMNS: udf_transform_column_name
: The name of one or more columns involved in the transform.

MAPPING:

Optional. Overrides the default source-to-target column mapping. GPSS supports two mapping syntaxes.

GPSS currently supports specifying only one of the `MAPPING` or (UDF) `TRANSFORMER` blocks in the load configuration file, not both.

When you specify a `MAPPING`, ensure that you provide a mapping for all RabbitMQ message data elements of interest. GPSS does not automatically match column names when you provide a `MAPPING`.

NAME: target_column_name
: Specifies the target Greenplum Database table column name.

EXPRESSION: { source_column_name | expression }
: Specifies a RabbitMQ `COLUMNS:NAME` (source_column_name) or an expression. When you specify an expression, you may provide a value expression that you would specify in the `SELECT` list of a query, such as a constant value, a column reference, an operator invocation, a built-in or user-defined function call, and so on.

target_column_name: { source_column_name | expression }
: When you use this `MAPPING` syntax, specify the target_column_name and {source_column_name | expression} as described above.

RABBITMQ:METADATA: Options

SCHEMA: metadata_schema_name
: The name of the Greenplum Database schema in which GPSS creates external tables. The default metadata_schema_name is `RABBITMQ:OUTPUT:SCHEMA`.

Greenplum Database COMMIT: Options

COMMIT:

Controls how GPSS commits a batch of data to Greenplum Database. You may specify both `MAX_ROW` and `MINIMAL_INTERVAL` as long as both values are not zero (`0`). Try setting and tuning

`MINIMAL_INTERVAL` to your environment; introduce a `MAX_ROW` setting only if you encounter high memory usage associated with message buffering.

SAVE_FAILING_BATCH: boolean
: Determines whether or not GPSS saves data into a backup table before it writes the data to Greenplum Database. Saving the data in this manner aids recovery when GPSS encounters errors during the evaluation of expressions. The default is `false`; GPSS does not use a backup table, and returns immediately when it encounters an expression error. When you set this property to `true`, GPSS writes both the good and the bad data in the batch to a backup table named `gpssbackup_<jobhash>`, and continues to process incoming messages. You must then manually load the good data from the backup table into Greenplum **or** set `RECOVER_FAILING_BATCH` (Beta) to `true` to have GPSS automatically reload the good data.

Using a backup table to hedge against mapping errors may impact performance, especially when the data that you are loading has not been cleaned.

RECOVER_FAILING_BATCH: boolean (Beta)
: When set to `true` and `SAVE_FAILING_BATCH` is also `true`, GPSS automatically reloads the good data in the batch and retains only the error data in the backup table. The default value is `false`; GPSS does not process the backup table.

Enabling this property requires that GPSS has the Greenplum Database privileges to create a function.

MAX_ROW: number_of_rows
: The number of rows to batch before triggering an `INSERT` operation on the Greenplum Database table. The default value of `MAX_ROW` is `0`, which instructs GPSS to ignore this commit trigger condition.

MINIMAL_INTERVAL: wait_time
: The minimum amount of time to wait (milliseconds) between each `INSERT` operation on the table. The default value is `5000`.

CONSISTENCY: { strong | at-least | at-most | none }
: Specify how GPSS should manage message offsets when it acts as a consumer of a RabbitMQ queue or stream. Valid values are `at-least` (GPSS stores the offsets before commit), `at-most` (GPSS stores the offsets after commit), and `none`. For streams, GPSS also supports `strong` consistency. The default value is `at-least`. Refer to Understanding RabbitMQ Message Offset Management for more detailed information.

IDLE_DURATION: idle_time
: The maximum amount of time to wait (milliseconds) for the first batch of data. When you use this property to enable lazy load, GPSS waits until RabbitMQ data is available before locking the target Greenplum table. You can specify:

  - `0` (lazy load is deactivated)

- `-1` (lazy load is activated, the job never stops), or
- a positive value (lazy load is activated, the job stops after idle_time duration of no data in the RabbitMQ queue or stream) The default value is `0`.

Greenplum Database TASK: Options

TASK:

Controls the running and scheduling of a periodic (maintenance) task.

POST_BATCH_SQL: udf_or_sql_to_run
: The user-defined function or SQL command(s) that you want to run after the specified number of batches are read from RabbitMQ. The default is null.

BATCH_INTERVAL: num_batches
: The number of batches to read before running udf_or_sql_to_run. The default batch interval is 0.

PREPARE_SQL: udf_or_sql_to_run
: The user-defined function or SQL command(s) that you want GPSS to run before it executes the job. The default is null, no command to run.

TEARDOWN_SQL: udf_or_sql_to_run
: The user-defined function or SQL command(s) that you want GPSS to run after the job stops. GPSS runs the function or command(s) on job success and job failure. The default is null, no command to run.

RabbitMQ PROPERTIES: Options

PROPERTIES:
: RabbitMQ configuration property names and values.

rmq_property_name
: The name of a RabbitMQ property.

rmq_property_value
: The RabbitMQ property value.

Job SCHEDULE: Options

SCHEDULE:

Controls the frequency and interval of restarting jobs.

RETRY_INTERVAL: retry_time
: The period of time that GPSS waits before retrying a failed job. You can specify the time interval in day (`d`), hour (`h`), minute (`m`), second (`s`), or millisecond (`ms`) integer units; do not mix units. The default retry interval is `5m` (5 minutes).

MAX_RETRIES: num_retries
: The maximum number of times GPSS attempts to retry a failed job. The default is 0, do not retry. If you specify a negative value, GPSS retries the job indefinitely.

RUNNING_DURATION: run_time
: The amount of time after which GPSS automatically stops a job. GPSS does not automatically stop a job by default.

AUTO_STOP_RESTART_INTERVAL: restart_time

The amount of time after which GPSS restarts a job that it stopped due to reaching `RUNNING_DURATION`.

MAX_RESTART_TIMES: num_restarts
: The maximum number of times that GPSS restarts a job that it stopped due to reaching `RUNNING_DURATION`. The default is 0, do not restart the job.

QUIT_AT_EOF_AFTER: clock_time
: The clock time after which GPSS stops a job every day when it encounters an EOF. By default, GPSS does not automatically stop a job that reaches EOF. GPSS never stops a job when the current time is before `clock_time`, even when GPSS encounters an EOF.

Job ALERT: Options

Controls notification when a job is stopped for any reason (success, completion, error, user-initiated stop).

COMMAND: command_to_run
: The program that the GPSS server runs on the GPSS server host, including arguments. The command must be executable by GPSS.
command_to_run has access to job-related environment variables that GPSS sets, including: `$GPSSJOB_NAME`, `$GPSSJOB_STATUS`, and `$GPSSJOB_DETAIL`.

WORKDIR: directory
: The working directory for command_to_run. The default working directory is the directory from which you started the GPSS server process. If you specify a relative path, it is relative to the directory from which you started the GPSS server process.

TIMEOUT: alert_time
: The amount of time after a job stops, prompting GPSS to trigger the alert (and run command_to_run). You can specify the time interval in day (`d`), hour (`h`), minute (`m`), or second (`s`) integer units; do not mix units. The default alert timeout is `-1s` (no timeout).

Template Variables

GPSS supports using template variables to specify property values in the load configuration file.

You specify a template variable value in the load configuration file as follows:

```
<PROPERTY>: {{<template_var>}}
```

For example:

```
MAX_RETRIES: {{numretries}}
```

GPSS substitutes the template variable with a value that you specify via the `-p | --property template\_var=value` option to the `gpsscli dryrun`, `gpsscli submit`, or `gpsscli load` command.

For example, if the command line specifies:

```
--property numretries=10
```

GPSS substitutes occurrences of `{{numretries}}` in the load configuration file with the value `10` before submitting the job, and uses that value while the job is running.

Notes

If you created a database object name using a double-quoted identifier (delimited identifier), you must specify the delimited name within single quotes in the `rabbitmq-v2.yaml` configuration file. For example, if you create a table as follows:

```
CREATE TABLE "MyTable" ("MyColumn" text);
```

Your `rabbitmq-v2.yaml` YAML configuration file would refer to the above table and column names as:

```
  COLUMNS:
     - name: '"MyColumn"'
       type: text
OUTPUT:
   TABLE: '"MyTable"'
```

You can specify backslash escape sequences in the CSV `DELIMITER`, `QUOTE`, and `ESCAPE` options. GPSS supports the standard backslash escape sequences for backspace, form feed, newline, carriage return, and tab, as well as escape sequences that you specify in hexadecimal format (prefaced with `\x`). Refer to Backslash Escape Sequences in the PostgreSQL documentation for more information.

Examples

Load data from RabbitMQ as defined in the Version 2 configuration file named `rmq2greenplumv2.yaml`:

```
gpsscli load rmq2greenplumv2.yaml
```

Example `rmq2greenplumv2.yaml` configuration file:

```
DATABASE: testdb
USER: gpadmin
PASSWORD: changeme
HOST: mdw-1
PORT: 15432
VERSION: 2
RABBITMQ:
    INPUT:
      SOURCE:
        SERVER: gpdmin:changeme@localhost:5672
        QUEUE: test
        VIRTUALHOST: gpadmin
      DATA:
        COLUMNS:
          - NAME: c1
            TYPE: int
          - NAME: c2
            TYPE: int
        FORMAT: CSV
        CSV_OPTION:
          DELIMITER: ","
          QUOTE: "'"
          NULL_STRING: "NA"
          ESCAPE: '\'
          FORCE_NOT_NULL: "c1,c2"
```

```
        FILL_MISSING_FIELDS: true
    ERROR_LIMIT: 25
  OUTPUT:
    SCHEMA: "public"
    TABLE: tbl_int_text_column
    MODE: INSERT
    MAPPING:
      - NAME: c1
        EXPRESSION: c1::int
      - NAME: c2
        EXPRESSION: c2::int
  METADATA:
    SCHEMA: staging_schema
  COMMIT:
    MAX_ROW: 1000
    MINIMAL_INTERVAL: 200
  PROPERTIES:
    eof.when.idle: 1500
    qos.prefetch.count: 10
```

See Also

rabbitmq-v3.yaml, gpsscli load, gpsscli submit

s3source-v3.yaml (Beta)

GPSS load configuration file for an s3 data source (version 3).

Synopsis

```
version: v3
```

```
targets:
- gpdb:
    host: <host>
    port: <greenplum_port>
    user: <user_name>
    password: <password>
    database: <db_name>
    work_schema: <work_schema_name>
    error_limit: <num_errors> | <percentage_errors>
    filter_expression: <filter_string>
    tables:
      - table: <table_name>
        schema: <schema_name>
        mode:
          # specify a single mode property block (described below)
          insert: {}
          update:
            <mode_specific_property>: <value>
            ...
          merge:
            <mode_specific_property>: <value>
            ...
        mapping:
```

```
          <target_column_name> : <source_column_name> | <expression>
          ...
        filter: <output_filter_string>
      ...
```

```
sources:
- s3:
    uri:
      - <s3_file_path>
      ...
    content:
      csv:
        <column_spec>
        <other_props>
    encoding: <char_set>
    s3param:
      version: <cfg_version>
      accessid: <s3_access_id>
      secret: <s3_secret>
      chunksize: <seg_buf_size>
      threadnum: <max_num>
      gpcheckcloud_newline: <newline_char>
      autocompress: <boolean>
      encryption: <boolean>
      proxy: <url>
      verifycert: <boolean>
      server_side_encryption: <boolean>
      low_speed_limit: <bps_limit>
      low_speed_time: <wait_secs>
option:
  schedule:
    max_retries: <num_retries>
    retry_interval: <retry_time>
    running_duration: <run_time>
    auto_stop_restart_interval: <restart_time>
    max_restart_times: <num_restarts>
    quit_at_eof_after: <clock_time>
  alert:
    command: <command_to_run>
    workdir: <directory>
    timeout: <alert_time>
```

Where the mode_specific_propertys that you can specify for `update` and `merge` mode follow:

```
update:
  match_columns: [<match_column_names>]
  order_columns: [<order_column_names>]
  update_columns: [<update_column_names>]
  update_condition: <update_condition>
```

```
merge:
  match_columns: [<match_column_names>]
  update_columns: [<update_column_names>]
  order_columns: [<order_column_names>]
  update_condition: <update_condition>
  delete_condition: <delete_condition>
```

And where you may specify any property value with a template variable that GPSS substitutes at runtime using the following syntax:

```
<property:> {{<template_var>}}
```

Description

Version 3 of the GPSS load configuration file is different in both content and format than previous versions of the file. Certain symbols used in the GPSS version 1 and 2 configuration file reference page syntax have different meanings in version 3 syntax:

- Brackets `[]` are literal and are used to specify a list in version 3. They are no longer used to signify the optionality of a property.
- Curly braces `{}` are literal and are used to specify YAML mappings in version 3 syntax. They are no longer used with the pipe symbol (`|`) to identify a list of choices.

You specify the configuration properties for a Greenplum Streaming Server (GPSS) s3 load job in a YAML-formatted configuration file that you provide to the `gpsscli submit` or `gpsscli load` commands. There are three types of configuration properties in this file - those that identify the Greenplum Database connection and target table, properties specific to the s3 data source that you will load into Greenplum, and job-related properties.

This reference page uses the name `s3source-v3.yaml` to refer to this file; you may choose your own name for the file.

The `gpsscli` utility processes the YAML configuration file keywords in order, using indentation (spaces) to determine the document hierarchy and the relationships between the sections. The use of white space in the file is significant. Keywords are not case-sensitive.

Keywords and Values

version Property

version: v3
: The version of the configuration file. You must specify `version: v3`.

targets:gpdb Properties

host: host
: The host name or IP address of the Greenplum Database coordinator host.

port: greenplum_port
: The port number of the Greenplum Database server on the coordinator host.

user: user_name
: The name of the Greenplum Database user/role. This user_name must have permissions as described in the Configuring Greenplum Database Role Privileges.

password: password
: The password for the Greenplum Database user/role.

database: db_name
: The name of the Greenplum database.

work_schema: work_schema_name
: The name of the Greenplum Database schema in which GPSS creates internal tables. The default work_schema_name is `public`.

error_limit: num_errors | percentage_errors
: The error threshold, specified as either an absolute number or a percentage. GPSS stops running the job when this limit is reached.

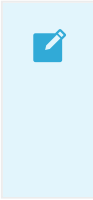
filter_expression: filter_string
: The filter to apply to the input data before GPSS loads the data into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. If the filter evaluates to `false`, the message is dropped. filter_string must be a valid SQL conditional expression and may reference one or more source value, key, or meta column names.

tables:
: The Greenplum Database tables, and the data that GPSS will load into each.

  table: table_name
  : The name of the Greenplum Database table into which GPSS loads the data.

  schema: schema_name
  : The name of the Greenplum Database schema in which table_name resides. Optional, the default schema is the `public` schema.

  mode:
  : The table load mode; `insert`, `merge`, or `update`. The default mode is `insert`.



`update` and `merge` are not supported if the target table column name is a reserved keyword, has capital letters, or includes any character that requires quotes (" ") to identify the column.

  insert:
  : Inserts source data into Greenplum.

  update:
  : Updates the target table columns that are listed in `update_columns` when the input columns identified in `match_columns` match the named target table columns and the optional `update_condition` is true.

  merge:
  : Inserts new rows and updates existing rows when:

    - columns are listed in `update_columns`,
    - the `match_columns` target table column values are equal to the input data, and
    - an optional `update_condition` is specified and met.

    Deletes rows when:

    - the `match_columns` target table column values are equal to the input data, and

- an optional `delete_condition` is specified and met.

  New rows are identified when the `match_columns` value in the source data does not have a corresponding value in the existing data of the target table. In those cases, the entire row from the source file is inserted, not only the `match_columns` and `update_columns`. If there are multiple new `match_columns` values in the input data that are the same, GPSS inserts or updates the target table using a random matching input row. When you specify `order_columns`, GPSS sorts the input data on the specified column(s) and inserts or updates from the input row with the largest value.

mode_property_name: value
: The name to value mapping for a mode property. Each `mode` supports one or more of the following properties as specified in the Synopsis.

match_columns: [match_column_names]
: A comma-separated list that specifies the column(s) to use as the join condition for the update. The attribute value in the specified target column(s) must be equal to that of the corresponding source data column(s) in order for the row to be updated in the target table. Required when `mode` is `merge` or `update`.

order_columns: [order_column_names]
: A comma-separated list that specifies the column(s) by which GPSS sorts the rows. When multiple matching rows exist in a batch, `order_columns` is used with `match_columns` to determine the input row with the largest value; GPSS uses that row to write/update the target. Optional. May be specified in `merge mode` to sort the input data rows.

update_columns: [update_column_names]
: A column-sparated list that specifies the column(s) to update for the rows that meet the `match_columns` criteria and the optional `update_condition`.
Required when `mode` is `merge` or `update`.

update_condition: update_condition
: Specifies a boolean condition, similar to that which you would declare in a `WHERE` clause, that must be met in order for a row in the target table to be updated (or inserted, in the case of a `merge`). Optional.

delete_condition: delete_condition
: In `merge mode`, specifies a boolean condition, similar to that which you would declare in a `WHERE` clause, that must be met for GPSS to delete rows in the target table that meet the `match_columns` criteria. Optional.

mapping:
: Optional. Overrides the default source-to-target column mapping.

: > **Note** When you specify a `mapping`, ensure that you provide a mapping for all source data elements of interest. GPSS does not automatically match column names when you provide a `mapping` block.

```
target\_column\_name: source\_column\_name \| expression
:   target\_column\_name specifies the target Greenplum Database table column nam
e. GPSS maps this column name to the source column name specified in source\_colu
mn\_name, or to an expression. When you specify an expression, you may provide a
value expression that you would specify in the `SELECT` list of a query, such as
```

```
a constant value, a column reference, an operator invocation, a built-in or user-
defined function call, and so on.
```

filter: output_filter_string
: The filter to apply to the output data before GPSS loads the data into Greenplum Database. If the filter evaluates to `true`, GPSS loads the message. If the filter evaluates to `false`, the message is dropped. output_filter_string must be a valid SQL conditional expression and may reference one or more `META` or `VALUE` column names.

sources:s3: Options

uri
: The path to an s3 file or bucket.

s3_file_path
: A URL identifying a file or files on s3 to be loaded. You can specify wildcards in any element of the path. To load all files in a directory, specify `dirname/*`.

content:
: The file type, field names, and type-specific properties of the file data. You must specify all data elements in the order in which they appear in the file. And you must specify `csv` format to read CSV- or text-format data.

csv
: When you specify the `csv` data format, GPSS reads the data into the list of columns that you specify. The data content cannot contain line ending characters (CR and LF).

column_spec
: The source to Greenplum column mapping. The supported column specification differs for different data formats as described below.
The default source-to-target data mapping behaviour of GPSS is to match a column name as defined in `source_column_name`, `column:name`, or `columns:name` with a column name in the target Greenplum Database `table`. You can override the default mapping by specifying a `mapping:` block.

columns:
: A set of column name/type mappings. The value `[]` specifies all columns.

name: column_name
: The name of a key or value column. column_name must match the column name of the target Greenplum Database table.

type: column_data_type
: The data type of the column. You must specify an equivalent data type for each data element and the associated Greenplum Database table column.

delimiter: delim_char
: Specifies a single ASCII character that separates columns within each message or row of data. The default delimiter is a comma (`,`).

encoding: char_set
: The source data encoding. You can specify an encoding character set when the source data is of the `csv`, `custom`, `delimited`, or `json` format. GPSS supports the character sets identified in Character

Set Support in the Tanzu Greenplum documentation.

s3param

The configuration parameters for the s3 data source are the same as those identified for the Tanzu Greenplum `s3` protocol. Refer to the Tanzu Greenplum s3 Protocol Configuration File documentation for the use and description of these parameters.

You can specify the s3 configuration parameters individually in the GPSS load configuration file. Alternatively, you can choose to provide the `config=<filepath>` option in the *s3_file_path* URI to specify the absolute path of a file on the local file system that contains the configuration parameter settings.

option: Properties

schedule:

Controls the frequency and interval of restarting jobs.

retry_interval: retry_time
: The period of time that GPSS waits before retrying a failed job. You can specify the time interval in day (`d`), hour (`h`), minute (`m`), second (`s`), or millisecond (`ms`) integer units; do not mix units. The default retry interval is `5m` (5 minutes).

max_retries: num_retries
: The maximum number of times that GPSS attempts to retry a failed job. The default is 0, do not retry. If you specify a negative value, GPSS retries the job indefinitely.

running_duration: run_time
: The amount of time after which GPSS automatically stops a job. GPSS does not automatically stop a job by default.

auto_stop_restart_interval: restart_time
: The amount of time after which GPSS restarts a job that it stopped due to reaching `running_duration`.

max_restart_times: num_restarts
: The maximum number of times that GPSS restarts a job that it stopped due to reaching `running_duration`. The default is 0, do not restart the job.

quit_at_eof_after: clock_time
: The clock time after which GPSS stops a job every day when it encounters an EOF. By default, GPSS does not automatically stop a job that reaches EOF. GPSS never stops a job when the current time is before `clock_time`, even when GPSS encounters an EOF.

alert:

Controls notification when a job is stopped for any reason (success, completion, error, user-initiated stop).

command: command_to_run
: The program that the GPSS server runs on the GPSS server host, including arguments. The command must be executable by GPSS.
: command_to_run has access to job-related environment variables that GPSS sets, including: `$GPSSJOB_NAME`, `$GPSSJOB_STATUS`, and `$GPSSJOB_DETAIL`.

workdir: directory

The working directory for command_to_run. The default working directory is the directory from which you started the GPSS server process. If you specify a relative path, it is relative to the directory from which you started the GPSS server process.

timeout: alert_time

The amount of time after a job stops, prompting GPSS to trigger the alert (and run command_to_run). You can specify the time interval in day (`d`), hour (`h`), minute (`m`), or second (`s`) integer units; do not mix units. The default alert timeout is `-1s` (no timeout).

Template Variables

GPSS supports using template variables to specify property values in the load configuration file.

You specify a template variable value in the load configuration file as follows:

```
<property>: {{<template_var>}}
```

For example:

```
max_retries: {{numretries}}
```

GPSS substitutes the template variable with a value that you specify via the `-p | --property <template_var=value>` option to the `gpsscli dryrun`, `gpsscli submit`, or `gpsscli load` command.

For example, if the command line specifies:

```
--property numretries=10
```

GPSS substitutes occurrences of `{{numretries}}` in the load configuration file with the value `10` before submitting the job, and uses that value while the job is running.

Notes

If you created a database object name using a double-quoted identifier (delimited identifier), you must specify the delimited name within single quotes in the load configuration file. For example, if you create a table as follows:

```
CREATE TABLE "MyTable" (c1 text);
```

Your YAML configuration file would refer to the table name as:

```
targets:
- gpdb:
    tables:
      - table: '"MyTable"'
```

You can specify backslash escape sequences in the CSV `delimiter`, `quote`, and `escape` options. GPSS supports the standard backslash escape sequences for backspace, form feed, newline, carriage return, and tab, as well as escape sequences that you specify in hexadecimal format (prefaced with `\x`). Refer to Backslash Escape Sequences in the PostgreSQL documentation for more information.

Examples

Submit a job to load data from a CSV file on s3 as defined in the v3 load configuration file named `loadfroms3_v3.yaml`:

```
$ gpsscli submit loadfroms3_v3.yaml
```

Example `loadfroms3_v3.yaml` configuration file:

```
version: v3
targets:
- gpdb:
    host: mdw-1
    port: 15432
    user: gpadmin
    password: changeme
    database: testdb
    work_schema: public
    error_limit: "25"
    tables:
      - table: orders
        schema: public
        mode:
          merge:
            match_columns: [pk]
            order_columns: [seq]
            delete_condition: flag = 0
        mapping:
          data: data
          pk: pk
          seq: seq
sources:
- s3:
    uri:
      - "s3://s3-us-east-1.amazonaws.com/mydir/mybucket/data0000"
    content:
      csv:
        columns:
          - name: pk
            type: int
          - name: seq
            type: int
          - name: data
            type: text
          - name: flag
            type: int
        delimiter: ";"
    s3params:
      version: 1
      accessid: 123
      secret: 456
      chunksize: 4096
      threadnum: 4
      gpcheckcloud_newline: "\n"
      autocompress: false
      encryption: false
      verifycert: false
```

```
      low_speed_limit: 1
option:
  schedule: {}
```

See Also

gpsscli load, gpsscli submit

Developing a Greenplum Streaming Server Client

The Greenplum Streaming Server (GPSS) is a gRPC server. gRPC provides a language- and platform-neutral client/server communication framework. For more information about gRPC, refer to the gRPC documentation.

GPSS exposes a Batch Data API gRPC interface.

You can develop a GPSS gRPC client on your operating system of choice and with the IDE or build environment of your choice. You can also develop the client using any programming language supported by gRPC.

You will perform the following tasks when you develop a GPSS client:

1. Examine the GPSS Service Definition.
2. Set up your development environment.
3. Compile the GPSS service definition to generate the GPSS client classes.
4. Code your GPSS client application.

Developing to the GPSS Batch Data API

You will perform the following tasks when you develop a client to the GPSS Batch Data API:

1. Examine the GPSS Batch Data API Service Definition.
2. Set up your development environment. For an example Java development environment setup refer to this procedure.
3. Compile the GPSS service definition to generate the GPSS client classes.
4. Code your GPSS client application. Each GPSS client will include code to perform the following tasks:
    - Connect to the GPSS Server
    - Connect to Greenplum Database
    - Retrieve Greenplum schema and table information
    - Prepare a Greenplum table for writing
    - Write data to the Greenplum table

GPSS Batch Data API Service Definition

The Greenplum Streaming Server (GPSS) is a gRPC server. GPSS uses gRPC protocol buffers (`protobuf`) to define the GPSS client interfaces and their message interchange format. With protocol buffers, the structure of the data (messages) and the operations supported (services) are defined in a `.proto` file, an ordinary text file. Refer to the Protocol Buffers Language Guide for detailed information about this data serialization framework.

The GPSS Batch Data API `.proto` file defines the methods that clients can invoke to obtain metadata information from, and write data to, Greenplum Database. For example, a GPSS client that you develop can submit a request to list the tables that reside in a specific Greenplum schema, or to insert data into a specific Greenplum table.

The GPSS Batch Data API service definition follows. **Copy/paste the contents to a file named `gpss.proto`, and note the file system location.**

```
syntax = "proto3";
import "google/protobuf/empty.proto";
import "google/protobuf/struct.proto";
import "google/protobuf/timestamp.proto";

package api;

option java_multiple_files = true;

// Connect service Request message
message ConnectRequest {
  string Host = 1;      // Host address of Greenplum coordinator; must be accessible f
rom gpss server system
  int32 Port = 2;       // Greenplum coordinator port
  string Username = 3;  // User or role name that gpss uses to access Greenplum
  string Password = 4;  // User password
  string DB = 5;        // Database name
  bool UseSSL = 6;      // Use SSL or not; ignored, use the gpss config file to config
SSL
  int32 SessionTimeout = 7;      // Release the session after idle for specified numbe
r of seconds
}

// Connect service Response message
message Session {
  string ID = 1;  // Id of client connection to gpss
}

// Operation mode
enum Operation {
  Insert = 0;  // Insert all data into table; behavior of duplicate key or data depend
s upon the constraints of the target table.
  Merge = 1;   // Insert and Update
  Update = 2;  // Update the value of "UpdateColumns" if "MatchColumns" match
  Read = 3;    // Not supported
}

// Required parameters of the Insert operation
message InsertOption {
  repeated string InsertColumns = 1;    // Names of the target table columns the inser
t operation should update; used in 'INSERT INTO', useful for partial loading
  bool TruncateTable = 2;               // Truncate table before loading?
  int64 ErrorLimitCount = 4;            // Error limit count; used by external table
```

```
  int32 ErrorLimitPercentage = 5;       // Error limit percentage; used by external ta
ble
}

// Required parameters of the Update operation
message UpdateOption {
  repeated string MatchColumns = 1;     // Names of the target table columns to compar
e when determining to update or not
  repeated string UpdateColumns = 2;    // Names of the target table columns to update
if MatchColumns match
  string Condition = 3;                 // Optional additional match condition; SQL sy
ntax and used after the 'WHERE' clause
  int64 ErrorLimitCount = 4;            // Error limit count; used by external table
  int32 ErrorLimitPercentage = 5;       // Error limit percentage; used by external ta
ble
}

// Required parameters of the Merge operation
// Merge operation creates a session-level temp table in StagingSchema
message MergeOption {
  repeated string InsertColumns = 1;
  repeated string MatchColumns = 2;
  repeated string UpdateColumns = 3;
  string Condition = 4;
  int64 ErrorLimitCount = 5;
  int32 ErrorLimitPercentage = 6;
}

// Open service Request message
message OpenRequest {
  Session Session = 1;      // Session ID returned by Connect
  string SchemaName = 2;    // Name of the Greenplum Database schema
  string TableName = 3;     // Name of the Greenplum Database table
  string PreSQL = 4;        // SQL to execute before gpss loads the data
  string PostSQL = 5;       // SQL to execute after gpss loads the data
  int32 Timeout = 6;        // Time to wait before aborting the operation (seconds); n
ot supported
  string Encoding = 7;      // Encoding of text data; not supported
  string StagingSchema = 8; // Schema in which gpss creates external and temp tables;
default is to create these tables in the same schema as the target table

  FormatAvro avro = 9;
  FormatBinary binary = 10;
  FormatCSV csv = 11;
  FormatDelimited delimited = 12;
  FormatJSON json = 13;
  FormatCustom custom = 14;
  string proto = 15;
  }

  oneof Option {            // Identify the type of write operation to perform
    InsertOption InsertOption = 100;
    UpdateOption UpdateOption = 101;
    MergeOption MergeOption = 102;
  }
}

message DBValue {
  oneof DBType {
```

```
    int32 Int32Value = 1;
    int64 Int64Value = 2;
    float Float32Value = 5;
    double Float64Value = 6;
    string StringValue = 7;  // Includes types whose values are presented as string bu
t are not a real string type in Greenplum; for example: macaddr, time with time zone,
box, etc.
    bytes BytesValue = 8;
    google.protobuf.Timestamp TimeStampValue = 10;  // Time without timezone
    google.protobuf.NullValue NullValue = 11;
    string OtherValue = 12;
  }
}

message Row {
  repeated DBValue Columns = 1;
}

message RowData {
  bytes Data = 1;     // A single protobuf-encoded Row
}

// Write service Request message
message WriteRequest {
  Session Session = 1;
  repeated RowData Rows = 2;     // The data to load into the target table
}

// Close service Response message
message TransferStats {          // Status of the data load operation
  int64 SuccessCount = 1;        // Number of rows successfully loaded
  int64 ErrorCount = 2;          // Number of error lines if Errorlimit is not reached
  repeated string ErrorRows = 3; // Number of rows with incorrectly-formatted data; no
t supported
}

// Close service Request message
message CloseRequest {
  Session session = 1;
  int32 MaxErrorRows = 2;        // -1: returns all, 0: nothing, >0: max rows
  bool Abort = 3;
}

// ListSchema service request message
message ListSchemaRequest {
  Session Session = 1;
}

message Schema {
  string Name = 1;
  string Owner = 2;
}

// ListSchema service response message
message Schemas {
  repeated Schema Schemas = 1;
}

// ListTable service request message
```

```
message ListTableRequest {
  Session Session = 1;
  string Schema = 2;    // 'public' is the default if no Schema is provided
}

// DescribeTable service request message
message DescribeTableRequest {
  Session Session = 1;
  string SchemaName = 2;
  string TableName = 3;
}

enum RelationType {
  Table = 0;
  View = 1;
  Index = 2;
  Sequence = 3;
  Special = 4;
  Other = 255;
}

message TableInfo {
  string Name = 1;
  RelationType Type = 2;
}

// ListTable service response message
message Tables {
  repeated TableInfo Tables = 1;
}

// DescribeTable service response message
message Columns {
  repeated ColumnInfo Columns = 1;
}

message ColumnInfo {
  string Name = 1;            // Column name
  string DatabaseType = 2;    // Greenplum data type

  bool HasLength = 3;         // Contains length information?
  int64 Length = 4;           // Length if HasLength is true

  bool HasPrecisionScale = 5; // Contains precision or scale information?
  int64 Precision = 6;
  int64 Scale = 7;

  bool HasNullable = 8;       // Contains Nullable constraint?
  bool Nullable = 9;
}

service Gpss {
  // Establish a connection to Greenplum Database; returns a Session object
  rpc Connect(ConnectRequest) returns (Session) {}

  // Disconnect, freeing all resources allocated for a session
  rpc Disconnect(Session) returns (google.protobuf.Empty) {}

  // Prepare and open a table for write
```

```
  rpc Open(OpenRequest) returns(google.protobuf.Empty) {}

  // Write data to table
  rpc Write(WriteRequest) returns(google.protobuf.Empty) {}

  // Close a write operation
  rpc Close(CloseRequest) returns(TransferStats) {}

  // List all available schemas in a database
  rpc ListSchema(ListSchemaRequest) returns (Schemas) {}

  // List all tables and views in a schema
  rpc ListTable(ListTableRequest) returns (Tables) {}

  // Decribe table metadata(column name and column type)
  rpc DescribeTable(DescribeTableRequest) returns (Columns) {}
}

// The format of the source data.
// If there is an intermediate column inside Format,
// then the source data will be transformed to the intermediate column.
// If there is no source_column_name in Format,
// then the column name will be the Target table column name,
// and the source column data type will be matched with Target column type.
message SourceDataFormat {
  oneof unit {
    FormatAvro avro = 1;
    FormatBinary binary = 2;
    FormatCSV csv = 3;
    FormatDelimited delimited = 4;
    FormatJSON json = 5;
    FormatCustom custom = 6;
    string protobuf = 7;
  }
}

message FormatAvro {
  string source_column_name = 1; // The source column name
  string schema_url = 2; // If specified, gpss requests the avro schema from url
  bool   bytes_to_base64 = 3; // When true and schema_url is specified, gpss converts
bytes field in avro message to base64-encoded string
  bool   ignore_deserialize_error = 4; // When true, gpss ignores avro deserialize err
ors, and puts data into log error
  string schema_path_on_gpdb = 5; // Used for standalone avro schema; if exists, gpss
retrieves the avro schema from the path on every node in the greenplum cluster
  string schema_ca_on_gpdb = 6; // The path to the specified CA certificate file for g
pss verifying the peer; the CA file must exist at that path on every greenplum segment
  string schema_cert_on_gpdb = 7; // The path to the specified client certificate file
for gpss connecting to HTTPS schema registry; required if the registry's client authen
tication is enabled
  string schema_key_on_gpdb = 8; // The path to the specified private key file for gps
s connecting to HTTPS schema registry; required if the registry's client authenticatio
n is enabled
  string schema_min_tls_version = 9; // The minimum transport layer security (TLS) ver
sion that gpss requests on the registry connection; the default value is 1.0, and gpss
supports minimum TLS versions of 1.0, 1.1, 1.2, and 1.3
}

message FormatBinary {
```

```
  string source_column_name = 1; // The source column name
}

message FormatCSV {
  repeated IntermediateColumn columns = 1; // Source column, move to format.Column c1:
bin, c2:json ...
  string delimiter = 2;
  string quote = 3;
  string null = 4;
  string escape = 5;
  string force_not_null = 6;
  string newline = 7;
  bool fill_missing_fields = 8;
  bool header = 9;
}

message FormatDelimited {
  repeated IntermediateColumn columns = 1; // The source column names
  string delimiter = 2;
}

message FormatJSON {
  IntermediateColumn column = 1; // The source column name
}

message FormatCustom {
  repeated IntermediateColumn columns = 1;
  string name = 2;
  repeated string options = 3;
}

// IntermediateColumn is an intermediate result after parsing SourceDataFormat,
// IntermediateColumn looks like a virtual table column. It
// will be used to filter or convert types.
// The Source Data is parsed to a table column style data.
//   source column: name and type, the type must be valid.
//   ex: convert a string "123" to 123 integer.
// Caution: the FormatJSON is not a decomposed format, json is treated as an integral
type.
message IntermediateColumn {
  string name = 1;
  string type = 2; // Greenplum Database basic data types are supported
}
```

Data Type Mapping

The GPSS Data API service definition includes messages that represent rows and columns of supported Greenplum Database data types.

Because Greenplum Database supports more data types than `protobuf`, the GPSS Data API provides a mapping between the types as follows:

gRPC Type	Greenplum Type
Int32Value	integer, serial
Int64Value	bigint, bigserial

gRPC Type	Greenplum Type
Float32Value	real
Float64Value	double
StringValue	text (any kind of data)
BytesValue	bytea
TimeStampValue	time, timestamp (without time zone)

In the simplest case, all Greenplum data types can be mapped to a string.

Setting up a Java Development Environment

Prerequisites

Before setting up your GPSS client development environment, ensure that you have:

- Access to a system on which you can develop code.
- Administrative access to your development system.

Example Procedure for Java

Perform the following procedure to set up a GPSS client Java development environment. This procedure assumes a Linux-based development system.

1. If not already present on your development system, install Java Development Kit version 1.8. You must have superuser permissions to install operating system packages. For example, to install the JDK on a CentOS development system:

   ```
   root@devsys$ yum install java-1.8.0-openjdk-1.8.0*
   ```

2. If it is not already installed on your development system, download the protocol buffer compiler version 3, and follow the installation instructions in the README.

3. Create a work directory. For example:

   ```
   user@devsys$ mkdir gpss_dev
   user@devsys$ cd gpss_dev
   user@devsys$ export GPSSDEV_DIR=`pwd`
   ```

   Examples in this guide reference your work directory. You may consider adding `$GPSSDEV_DIR` to your `.bash_profile` or equivalent shell initialization script.

4. Download the gRPC Java code generation plugin and example code by cloning its repository. For example:

   ```
   user@devsys$ git clone https://github.com/grpc/grpc-java.git
   ```

   The command clones the repository to a directory named `grpc-java` in the current directory.

5. Prepare the work directory for Java development. Create directories for the source code and the `gpss.proto` service definition file. For example:

    ```
    user@devsys$ mkdir -p client/src/main/java
    user@devsys$ mkdir client/src/main/proto
    ```

6. Navigate to the `proto` directory and copy the `gpss.proto` service definition file to this directory. Refer to GPSS Batch Data API Service Definition if you have not already created the file. For example:

    ```
    user@devsys$ cd client/src/main/proto
    user@devsys$ cp <dir>/gpss.proto .
    ```

7. Navigate back to your work directory:

    ```
    user@devsys$ cd $GPSSDEV_DIR
    ```

Generating the Batch Data API Client Classes

The GPSS Batch Data API service definition defines messages and services exposed by the GPSS gRPC server. This definition resides in the `gpss.proto` file. You must run the protocol buffer compiler (`protoc`) on the `gpss.proto` file to generate the classes that the GPSS client uses to query metadata information from, and write data to, Greenplum Database.

In some build environments such as `gradle`, you can configure the build to automatically generate the GPSS client classes for you. Refer to the `gRPC` examples for your programming language for sample build configurations with automatic code generation. For example, the `gradle` build and settings files for the examples in the `grpc-java` repository are configured to automatically generate the Java classes for you.

You may choose to run the protocol buffer compiler manually. If you choose to run the command manually, be sure to specify the gRPC codegen plugin for the programming language in which you are developing the GPSS client. For example, you may run a command similar to the following to generate the gRPC GPSS Java client code:

```
$ protoc --plugin=protoc-gen-grpc-java=../grpc-java/compiler/build/exe/java_plugin/pro
toc-gen-grpc-java \
    --proto_path=./src/main/proto --grpc-java_out=./gen/grpc/ \
    --java_out=./gen/java/ gpss.proto
```

Coding the GPSS Batch Data Client

You will develop GPSS client code to perform the following tasks:

1. Connect to the GPSS Server
2. Connect to Greenplum Database
3. Retrieve Greenplum schema and table information
4. Prepare a Greenplum table for writing
5. Write data to the Greenplum table

The code excerpts in this topic are written in the Java programming language.

The code excerpts build upon each other. For example, the client instantiates a `Session` object in the Connect to Greenplum Database section. Subsequent code excerpts reference the same `Session` object instance.

Connecting to the GPSS Server

Before a GPSS client can load data into Greenplum Database, the client must first establish a connection to the Greenplum Streaming Server.

When you start a GPSS server instance, you provide a JSON-format configuration file. You identify the hostname and port on which the GPSS server instance listens for connection requests in this configuration file. For more information about the GPSS server, refer to the gpss reference page.

The GPSS client application must create a gRPC managed channel to this GPSS instance. Because GPSS supports the simple RPC service, the client creates a blocking stub on the channel.

This sample Java client code connects to and disconnects from a GPSS service instance listening for connections on port number 5019 on the local host:

```
java.util.concurrent.TimeUnit;
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;

String gpssHost = "localhost";
Integer gpssPort = 5019;
ManagedChannel channel = null;
GpssGrpc.GpssBlockingStub bStub = null;

try {
  // connect to GPSS gRPC service instance; create a channel and a blocking stub
  channel = ManagedChannelBuilder.forAddress(gpssHost, gpssPort)
    .usePlaintext(true)
    .build();
  bStub = GpssGrpc.newBlockingStub(channel);

  // (placeholder) do stuff here

  // shutdown the channel
  channel.shutdown().awaitTermination(7, TimeUnit.SECONDS);

} catch (Exception e) {
  ...
}
```

After the GPSS client instantiates a stub, the client can use the stub to invoke the GPSS service methods.

Connecting to Greenplum Database

The client uses the GPSS `Connect` service to connect to a specific Greenplum database. The `Disconnect` service closes the connection to Greenplum.

The `Connect` and `Disconnect` service definitions follow:

```
rpc Connect(ConnectRequest) returns (Session) {}
rpc Disconnect(Session) returns (google.protobuf.Empty) {}
```

The client specifies the information required to connect to Greenplum Database in a `ConnectRequest` message. The `Connect` service returns a `Session` message. The session identifies the client's connection to the GPSS server. The client must provide the session when it invokes metadata- and table-related services on Greenplum Database. The client must also provide the session when it disconnects from Greenplum Database.

The `ConnectRequest` and `Session` message definitions:

```
message ConnectRequest {
  string Host = 1;
  int32 Port = 2;
  string Username = 3;
  string Password = 4;
  string DB = 5;
  bool UseSSL = 6;
  int32 SessionTimeout=7;
}

message Session {
  string ID = 1;
}
```

The following sample includes Java client code to:

- Create and populate a `ConnectRequest` protocol buffer object.
- Use the blocking stub to call the `Connect` service.
- Save the `Session` response object.
- Disconnect the session.

```
Session mSession = null;
String gpCoordHost = "localhost";
Integer gpCoordPort = 15432;
String gpRoleName = "gpadmin";
String gpPasswd = "changeme";
String dbname = "testdb";

// create a connect request builder
ConnectRequest connReq = ConnectRequest.newBuilder()
    .setHost(gpCoordHost)
    .setPort(gpCoordPort)
    .setUsername(gpRoleName)
    .setPassword(gpPasswd)
    .setDB(dbname)
    .setUseSSL(false)
    .setSessionTimeout(1000)
  .build();

// use the blocking stub to call the Connect service
mSession = bStub.connect(connReq);
```

```
// (placeholder) do greenplum stuff here

// use the blocking stub to call the Disconnect service
bStub.disconnect(mSession);
```

After the GPSS client connects to Greenplum Database, the client can invoke service requests to retrieve information about schemas and tables, and write to Greenplum tables.

Retrieving Greenplum Schema and Table Info

Your GPSS client may need to examine Greenplum Database schemas or the definition of a Greenplum table. The GPSS Data API defines three services to obtain metadata information about Greenplum schemas and tables:

- `ListSchema` - list all schemas defined in the database
- `ListTable` - list all tables in a schema
- `DescribeTable` - return the definition of each column in a table

Listing the Schemas in the Database

The GPSS Data API defines the `ListSchema` service and supporting messages to list the schemas defined in a Greenplum database:

```
rpc ListSchema(ListSchemaRequest) returns (Schemas) {}

message ListSchemaRequest {
  Session Session = 1;
}

message Schemas {
  repeated Schema Schemas = 1;
}

message Schema {
  string Name = 1;
  string Owner = 2;
}
```

`ListSchema` returns the list of schemas defined in the database identified by the session specified by the client. This service returns the name of the schema and the Greenplum Database role that owns the schema.

This sample Java client code collects the names of the schemas in the database identified by the specified session:

```
import java.util.ArrayList;
import java.util.List;

// create a list schema request builder
ListSchemaRequest lsReq = ListSchemaRequest.newBuilder()
    .setSession(mSession)
  .build();
```

```
// use the blocking stub to call the ListSchema service
List<Schema> listSchema = bStub.listSchema(lsReq).getSchemasList();

// extract the name of each schema and save in an array
ArrayList<String> schemaNameList = new ArrayList<String>();
for(Schema s : listSchema) {
  schemaNameList.add(s.getName());
}
```

Listing the Tables in a Schema

The GPSS Data API defines the `ListTable` service and supporting messages to list the tables defined in a specific Greenplum Database schema:

```
rpc ListTable(ListTableRequest) returns (Tables) {}

message ListTableRequest {
  Session Session = 1;
  string Schema = 2;
}

enum RelationType {
  Table = 0;
  View = 1;
  Index = 2;
  Sequence = 3;
  Special = 4;
  Other = 255;
}

message TableInfo {
  string Name = 1;
  RelationType Type = 2;
}

message Tables {
  repeated TableInfo Tables = 1;
}
```

`ListTable` returns a list of the tables in the schema and the database (session) specified by the client. This service also returns the type of Greenplum Database table/relation.

This sample Java code collects a list of the names of all of the tables defined in the specified schema and database:

```
// use the first schema name returned in the ListSchema code excerpt
String schemaName = schemaNameList.get(0);

// create a list table request builder
ListTableRequest ltReq = ListTableRequest.newBuilder()
    .setSession(mSession)
    .setSchema(schemaName)
  .build();

// use the blocking stub to call the ListTable service
```

```
List<TableInfo> tblList = bStub.listTable(ltReq).getTablesList();

// extract the name of each table only and save in an array
ArrayList<String> tblNameList = new ArrayList<String>();
for(TableInfo ti : tblList) {
  if(ti.getTypeValue() == RelationType.Table_VALUE) {
    tblNameList.add(ti.getName());
  }
}
```

Acquiring the Column Definitions of a Table

The GPSS Data API defines the `DescribeTable` service and supporting messages to retrieve the column definitions of a Greenplum Database table:

```
rpc DescribeTable(DescribeTableRequest) returns (Columns) {}

message DescribeTableRequest {
  Session Session = 1;
  string SchemaName = 2;
  string TableName = 3;
}

message Columns {
  repeated ColumnInfo Columns = 1;
}

message ColumnInfo {
  string Name = 1;
  string DatabaseType = 2;
  bool HasLength = 3;
  int64 Length = 4;
  bool HasPrecisionScale = 5;
  int64 Precision = 6;
  int64 Scale = 7;
  bool HasNullable = 8;
  bool Nullable = 9;
}
```

`DescribeTable` returns a list of column definitions for the table in the schema and the database (session) specified by the client. The column definition includes the name and the type of the Greenplum Database column. The definition also includes length, precision, and scale information, if applicable.

Sample Java code to retrieve the column definitions of the table in the specified schema and database, and print the column name and type to `stdout`:

```
// the name of the first table returned in the ListTable code excerpt
String tableName = tblNameList.get(0);

// create a describe table request builder
DescribeTableRequest dtReq = DescribeTableRequest.newBuilder()
    .setSession(mSession)
    .setSchemaName(schemaName)
    .setTableName(tableName)
  .build();
```

```
// use the blocking stub to call the DescribeTable service
List<ColumnInfo> columnList = bStub.describeTable(dtReq).getColumnsList();

// print the name and type of each column
for(ColumnInfo ci : columnList) {
  String colname = ci.getName();
  String dbtype = ci.getDatabaseType();
  // display the column name and type to stdout
  System.out.println( "column " + colname + " type: " + dbtype );
}
```

Specifying and Preparing a Greenplum Table for Writing

The client uses the GPSS `Open` service to specify and prepare a Greenplum Database table for writing. The `Close` service closes, or ends, a write operation on the table.

The `Open` service definition follows:

```
rpc Open(OpenRequest) returns(google.protobuf.Empty) {}
```

The GPSS client can insert or merge data into or update the data in a Greenplum Database table. The client specifies the mode of the write operation via a mode-specific `Option` that it provides to the `OpenRequest` message. Supported write operation modes include:

- Insert - add data to the table, optionally truncating before writing
- Update - update table data, specifying the join column and an optional update condition
- Merge - insert table data, specifying the join column and an optional insert condition

Relevant messages for the `Open` service include:

```
message InsertOption {
  repeated string InsertColumns = 1;
  bool TruncateTable = 2;
  int64 ErrorLimitCount = 4;
  int32 ErrorLimitPercentage = 5;
}

message UpdateOption {
  repeated string MatchColumns = 1;
  repeated string UpdateColumns = 2;
  string Condition = 3;
  int64 ErrorLimitCount = 4;
  int32 ErrorLimitPercentage = 5;
}

message MergeOption {
  repeated string InsertColumns = 1;
  repeated string MatchColumns = 2;
  repeated string UpdateColumns = 3;
  string Condition = 4;
  int64 ErrorLimitCount = 5;
  int32 ErrorLimitPercentage = 6;
}

message OpenRequest {
```

```
  Session Session = 1;
  string SchemaName = 2;
  string TableName = 3;
  string PreSQL = 4;
  string PostSQL = 5;
  int32 Timeout = 6; //seconds
  string Encoding = 7;
  string StagingSchema = 8;

  oneof Option {
    InsertOption InsertOption = 100;
    UpdateOption UpdateOption = 101;
    MergeOption MergeOption = 102;
  }
}
```

After it completes loading data or encounters an error from GPSS or the source, the GPSS client invokes the `Close` service on the table. `Close` returns the success and error row counts and any error strings in the `TransferStats` message.

The `Close` service definition and relevant messages follow:

```
rpc Close(CloseRequest) returns(TransferStats) {}

message CloseRequest {
  Session session = 1;
  int32 MaxErrorRows = 2;
  bool Abort = 3;
}

message TransferStats {
  int64 SuccessCount = 1;
  int64 ErrorCount = 2;
  repeated string ErrorRows = 3;
}
```

Use `MaxErrorRows` to identify the form and amount of error information that GPSS returns:

MaxErrorRows Value	Description
-1	Returns an `ErrorCount` and all `ErrorRows` (error messages).
0	Returns only an `ErrorCount`; no `ErrorRows`. The default.
n > 0	Returns an `ErrorCount` and a maximum of *n* `ErrorRows`.

If the GPSS client encounters an unrecoverable error that affects the load operation to Greenplum Database, it may choose to cancel writing the current batch of data. When the `CloseRequest` message is instantiated with `.setAbort(true)`, GPSS cancels and rolls back the pending write transaction. This rolls back all writes since the `Open`.

Sample Code

Suppose you create a Greenplum Database table with the following command:

```
CREATE TABLE public.loaninfo( loantitle text, riskscore int, d2iratio text);
```

Sample Java code to prepare to open the `loaninfo` table for insert, and then close the table follows:

```
Integer errLimit = 25;
Integer errPct = 25;
// create an insert option builder
InsertOption iOpt = InsertOption.newBuilder()
    .setErrorLimitCount(errLimit)
    .setErrorLimitPercentage(errPct)
    .setTruncateTable(false)
    .addInsertColumns("loantitle")
    .addInsertColumns("riskscore")
    .addInsertColumns("d2iratio")
  .build();

// create an open request builder
OpenRequest oReq = OpenRequest.newBuilder()
    .setSession(mSession)
    .setSchemaName(schemaName)
    .setTableName(tableName)
    //.setPreSQL("")
    //.setPostSQL("")
    //.setEncoding("")
    .setTimeout(5)
    //.setStagingSchema("")
    .setInsertOption(iOpt)
  .build();

// use the blocking stub to call the Open service; it returns nothing
bStub.open(oReq);

// (placeholder) write data here

// create a close request builder
TransferStats tStats = null;
CloseRequest cReq = CloseRequest.newBuilder()
    .setSession(mSession)
    //.setMaxErrorRows(15)
    //.setAbort(true)
  .build();

// use the blocking stub to call the Close service
tStats = bStub.close(cReq);

// display the result to stdout
System.out.println( "CloseRequest tStats: " + tStats.toString() );
```

Writing Data to a Greenplum Table

After opening a Greenplum Database table with a specific open mode, a GPSS client can write one or more rows of data to the table. The client must map the source data to a gRPC data type. The GPSS server maps the gRPC type to a Greenplum Database type as specified in Data Type Mapping.

The `Write` service definition and relevant messages follow:

```
rpc Write(WriteRequest) returns(google.protobuf.Empty) {}

message DBValue {
  oneof DBType {
    int32 Int32Value = 1;
    int64 Int64Value = 2;
    float Float32Value = 5;
    double Float64Value = 6;
    string StringValue = 7;
    bytes BytesValue = 8;
    google.protobuf.Timestamp TimeStampValue = 10;
    google.protobuf.NullValue NullValue = 11;
  }
}

message Row {
  repeated DBValue Columns = 1;
}

message RowData {
  bytes Data = 1;
}

message WriteRequest {
  Session Session = 1;
  repeated RowData Rows = 2;
}
```

The GPSS client application must provide values for all columns declared in the Greenplum table definition. If there is no data to write for a specific column, you must explicitly specify a `null` value for the column when generating the row's `RowData`.

Sample Java code to write two rows of data to the `loaninfo` table that you opened in insert mode in the previous section follows:

```
// create an array of rows
ArrayList<RowData> rows = new ArrayList<>();
for (int row = 0; row < 2; row++) {
  // create a row builder
  api.Row.Builder builder = api.Row.newBuilder();

  // create builders for each column, in order, and set values - text, int, text
  api.DBValue.Builder colbuilder1 = api.DBValue.newBuilder();
  colbuilder1.setStringValue("xxx");
  builder.addColumns(colbuilder1.build());
  api.DBValue.Builder colbuilder2 = api.DBValue.newBuilder();
  colbuilder2.setInt32Value(77);
  builder.addColumns(colbuilder2.build());
  api.DBValue.Builder colbuilder3 = api.DBValue.newBuilder();
  colbuilder3.setStringValue("yyy");
  builder.addColumns(colbuilder3.build());

  // build the row
  RowData.Builder rowbuilder = RowData.newBuilder().setData(builder.build().toByteStri
ng());

  // add the row
```

```
  rows.add(rowbuilder.build());
}

// create a write request builder
WriteRequest wReq = WriteRequest.newBuilder()
    .setSession(mSession)
    .addAllRows(rows)
  .build();

// use the blocking stub to call the Write service; it returns nothing
bStub.write(wReq);
```

If GPSS encounters an error, it rolls back the pending write transaction; rolling back all writes since the `Open`.

The client determines the success or failure of the write operation from the `TransferStats` returned when the client invokes the `Close` service to close the table.