JCC LogMiner Loader

Version 3.6

Released February, 2019

From the JCC Toolset for Databases

JCC Consulting, Inc.
600 Newark Road
P.O. Box 381
Granville, Ohio 43023
U.S.A

Kafka Option

Table of Contents

Contact Information

E-mail	JCC-LMLoader@JCC.com
Phone	+1 (740)587-0157
FAX	+1 (740)587-0163
Post Office	LogMiner Loader JCC Consulting, Inc. Box 381 Granville, OH 43023

For JCC LogMiner Loader licensing and support, contact JCC Consulting.

JCC Consultants are also eager to hear your comments, questions, and examples.

JCC also provides consulting for solution architectures and on-site support for database review and for getting started with new architectures. Training and temporary licenses are available.

Notices

Uses in this Document

In this document, Oracle Rdb and Oracle's other RDBMS are referred to frequently and need to be distinguished. Consequently, Oracle Rdb is referred to as "Rdb" and Oracle's other database product (whatever its version) is referred to simply as "Oracle." Similarly, Oracle Rdb LogMiner is referred to as "Rdb LogMiner" or as "LogMiner." The JCC LogMiner Loader is sometimes referred to simply as "the Loader".

CHAPTER 1

JCC LogMiner Loader Kafka Option

Kafka is a message transport that is receiving increasing attention for use in advanced data architectures. The JCC LogMiner Loader is an established tool for collecting changes made to an Oracle Rdb database, transforming them as needed, and publishing them to a variety of databases and tools. The JCC LogMiner Loader Kafka Option combines these facilities to provide Rdb transactional data to a data lake or for evaluation with streaming analytical tools or for additional processing by a wide variety of existing and newly emerging software and tools.

Release 3.6 of the JCC LogMiner Loader is concurrent with release of the Kafka Option. The JCC LogMiner Loader Kafka Option is an add-on to the base JCC LogMiner Loader capabilities and requires an additional license. Trial licenses are available for proof of concept testing. Contact JCC to discuss licensing or other questions. Training and consulting for both detailed support and architectural review are also available from JCC.

None of this documentation attempts to provide a complete description of the JCC LogMiner Loader or of any of the Kafka distribution variants. For that, please consult the relevant documentation.[1]

Kafka Option

Kafka software uses a Publish – Subscribe (Pub-Sub) model. Software (in this case the JCC LogMiner Loader) publishes messages to a Kafka server. One or more “subscribers” can consume those messages. Messages are published by the Kafka software to one or more “topics”. Topics can be created in advance or created the first time a message is published to that topic. The exact naming of the topics used by the JCC LogMiner Loader can be controlled in the JCC LogMiner Loader Control File[2].

The JCC LogMiner Loader does not include Kafka consumers. The consumers are the responsibility of the project that will be using the Kafka data.

There are a number of Kafka distributions, including:

- Apache Kafka (https://kafka.apache.org/)
- Confluent Kafka (https://www.confluent.io/)
    - Confluent Open Source
    - Confluent Enterprise (licensed, supported)

The JCC LogMiner Loader Kafka Option can publish data in three formats[3]:

- XML
- JSON
- AVRO with Schema Registration.

1. The complete JCC LogMiner Loader documentation is available for download from http://www.jcc.com/products/jcc-logminer-loader-and-data-pump. For pointers to Kafka documentation, see “Kafka Documentation and Licensing” on page 10.
2. The Loader Control File is discussed in “Control File” on page 21 and elsewhere.
3. These formats are discussed further in “Output Format Examples” on page 45 and elsewhere.

In addition, the JCC LogMiner Loader Kafka option supports publishing data either in plain text or using SSL encryption[1].

Kafka documents these models[2] for publishing messages:

- ExactlyOnce
- Transaction

The current release of the JCC LogMiner Loader supports both with Transaction as the default.

Regression Testing

The JCC LogMiner Loader regression tests use a number of different Kafka versions, distributions, and configurations. The Kafka options are added to existing JCC regression testing which is automated and includes random choices for settings for features.

Regression testing for the JCC LogMiner Loader Kafka Option uses Apache Kafka for some of the testing of the XML and JSON formats. Apache Kafka is an open-source message broker that uses the Publish – Subscribe (Pub-Sub) model. Publishing to Kafka in XML and JSON formats are supported with any V1.0 (or later) Kafka distribution.

Publishing in AVRO format with Schema Registration requires a Confluent Kafka distribution, as Confluent Kafka supports Schema Registration capabilities that are not available in the Apache Kafka distribution. The JCC LogMiner Loader supports Schema Registration through use of Confluent Kafka.

1. SSL is discussed in “SSL Encryption” on page 36 and elsewhere.
2. Kafka Models, as used with the Loader, are discussed in “Kafka Model” on page 39 and elsewhere.

Kafka Documentation and Licensing

Apache Kafka V1.0 documentation is available from https://kafka.apache.org/10/documentation.html. Apache Kafka is covered by Apache License V2.0.

Documentation for the Confluent Kafka distribution is available at https://docs.confluent.io/current/. Confluent Open Source is covered by Apache License V2.0.

For Apache License V2.0 details, see http://www.apache.org/licenses/.

Introduction to Kafka for Rdb Users

When using the JCC LogMiner Loader to publish Rdb data to a Kafka server, it is helpful to understand the Kafka terminology. The following sections provide a high-level introduction to Kafka.

Kafka

Kafka is an open source utility that supports publishing and consuming messages. A message published to a Kafka server can be in whatever format the message consumer will understand. The JCC LogMiner Loader Kafka Option publishes messages in XML, JSON, or Avro formats. The connection to the Kafka server can either be unencrypted plain text or use SSL encryption.

Producer

A Kafka producer publishes messages to a topic or topics on a Kafka server. In the case of the JCC LogMiner Loader Kafka Option, the JCC LogMiner Loader is the producer. The messages published are committed transactions extracted from the Rdb AIJ files.

Consumer

Kafka consumers subscribe to topics on a Kafka server. When new messages are published to the Kafka server, they are available to the consumer. The consumer needs to understand the format of the messages but is otherwise independent of the producer. The JCC LogMiner Loader Kafka Option does not include a Kafka consumer.

Messages

The JCC LogMiner Loader publishes messages to the Kafka server. Each message corresponds to a record in an Rdb committed transaction. The messages can be published using XML, JSON, or Avro formats.

Topics

The JCC LogMiner Loader publishes messages to one topic per output table. The topic names default to the output table name, but can be configured to have any name chosen. It is also possible to include a variety of additional information, using VirtualColumns, message headers, and various formatting options.

Kafka Clusters

A Kafka installation can be configured on multiple servers banded together as a Kafka cluster. A cluster provides the advantage of higher availability and ensures the consistency of data while supporting parallel operations across the cluster.

Zookeeper

A Kafka installation uses Zookeeper to manage topics and cluster information. A JCC LogMiner Loader session needs the appropriate Zookeeper address and port to connect to a Kafka server, but does not need any additional information about the Zookeeper configuration.

Additional Kafka Details and This Document

The details of all possible Kafka configurations are beyond the scope of this documentation. Please contact JCC with specific questions that relate to the JCC LogMiner Loader and the Loader's Kafka Option.

Introduction to Rdb for Kafka Users

Oracle Rdb [1] is a robust, mature transactional database product.

OpenVMS

Rdb was designed for use on OpenVMS and has not been ported to other platforms. Consequently, the JCC LogMiner Loader runs on OpenVMS. A basic understanding of Rdb and OpenVMS topics and, of course, a knowledge of JCC LogMiner Loader options will facilitate your architecture and use.

Transactions

Rdb supports ACID (Atomic, Consistent, Isolated, Durable) transactions. That is, processes that change the data in an Rdb database cause a transaction to start and the changed data is not visible in the database unless and until the transaction is committed. A transaction takes the database from one consistent state to another with no partial changes. If a transaction is rolled back, the changes do not occur.

Data changes are not available until the transaction is completed. As part of the transaction commit, the changes are recorded in the Rdb After Image Journal (AIJ)[2]. When Rdb is configured to support the LogMiner, the AIJ is read by the Rdb LogMiner command (RMU/Unload/After/Continuous). When used with the JCC LogMiner Loader, the LogMiner writes the changes to an OpenVMS mailbox that is read by the Loader. The Loader processes the data and writes the changes to the Loader target.

Push

An architecture that includes Rdb and the JCC LogMiner Loader pushes committed data changes to the Kafka target. This differs from Kafka Connectors that pull data and are not driven by the transaction log. With the JCC LogMiner Loader Kafka Option, the data published to Kafka is limited to the data in the committed transactions that have been pushed from the transaction log.

1. Oracle Rdb was purchased by Oracle Corporation from Digital Equipment Corporation in 1994 and has received continuing development. It is referred to here and in other JCC LogMiner Loader documentation simply as “Rdb”. In this documentation, Oracle’s other database product is referred to simply as “Oracle”.
2. Rdb’s AIJ corresponds to what other database platforms refer to as the Transaction Log.

The JCC LogMiner Loader includes a tool called the Data Pump that can assist in pushing entire tables to a Kafka server. See the chapter on Data Pump in the full documentation for this valuable tool for setting up the target.

Guide to the Documentation

This document is specific to the Kafka Option of the JCC LogMiner Loader. It will also be important to consult the full documentation for the JCC LogMiner Loader, as well as the documentation for other products in the architecture.

TABLE 1. **Documentation**

Sections in Documentation for the Kafka Option	**Related Resources in the Full JCC LogMiner Loader Documentation**
Chapter 1 - introduction	Chapters 1, 2, and 3
Chapter 2 - Installation	Chapter 4 - Installaation
Chapter3 - Control File	Chapter 13 - Control File
Chapter 4 - Notes for the Loader Administrator	Chapter 15 - Performance Considerations Chapter 16 - Aids for the Administrator
Chapter 5 - Output Format Examples	Kafka specific examples
Chapter 6 - Kafka Command Examples	Kafka specific examples
Other chapters of interest	Chapter 14 - Monitoring an Ongoing Loader Operation Chapter 17 - Schema Changes, Data Transforms, ...

CHAPTER 2

Installation

The full documentation for the JCC LogMiner Loader has chapters on installation and configuration of the JCC Loader product. This chapter pertains only to the extra installation steps for the Kafka Option.

Configuration options of special concern to those using the Kafka Option are covered in “Control File” on page 21 and in the Control File chapter in the full documentation. Examples there and here include syntax created for the JCC LogMiner Loader Control File. That syntax is fully specified in the full documentation.

Kafka Libraries

For the JCC LogMiner Loader to publish data to Kafka, it needs a number of Kafka routines that exist in a variety of languages, including Java. The JCC LogMiner Loader uses the Kafka Java libraries to publish messages to a Kafka server.

The Kafka libraries needed on OpenVMS to support the JCC LogMiner Loader Kafka option are not included in the JCC LogMiner Loader installation kit. The required libraries must be extracted from a Kafka kit or installation and migrated to an OpenVMS server.[1]

Which libraries are required depend on the Kafka output formats that will be used. If only the XML and JSON output formats are being used, the Apache Kafka libraries are sufficient. If the Avro output format will be used, the Confluent Kafka libraries are needed. Apache Kafka is open source; Confluent Kafka must be licensed.

Transferring the Kafka Java libraries to an OpenVMS server requires some manual steps. These are covered in the following sections.

Apache Kafka Windows Installation

To extract the library:

- Download the Kafka installation to a Windows system. This file has a name such as kafka_2.11-1.1.0.tgz
- Use 7zip (or some other utility) to extract kafka_2.11-1.1.0.tar from kafka_2.11-1.1.0.tgz
- Use 7zip (or some other utility) to extract the Kafka folder tree from kafka_2.11-1.1.0.tar
- FTP the contents of the "libs" folder in binary mode to a directory on the OpenVMS server. In the example to follow, the contents of the bin folder DISK$STATIC:[KAFKA.1-1-0.LIBS] are copied
- Reset the Jar file attributes with the command:

```
$ set file/attr=(rfm:stmlf,rat:cr,lrl:0,mr:0) -
    DISK$STATIC:[KAFKA.1-1-0.LIBS]*.jar
```

The directory used on OpenVMS is arbitrary. It can be whatever makes sense in your environment. The folder will be pointed to in the classpath specification in a JCC LogMiner Loader Control File. See Keyword: JDBC in the Control File chapter in the full Loader documentation. Also, see an example in the section "Kafka Specific Configuration Directives - XML" on page 34 of this document.

1. Documentation and licensing for Kafka products must be obtained from the appropriate source. See "Kafka Documentation and Licensing" on page 10 for assistance.

Multiple JCC LogMiner Loader sessions can use the same Kafka Jar files, so this step only needs to be done once.

The example above uses the Apache Kafka V1.1.0 Jar files to publish messages to an Apache Kafka V1.0.0 server. That version skew has been tested. Future testing by JCC developers and testers will add more information on how much version skew can be supported.

Confluent Kafka Installation

The Avro Kafka Option takes advantage of the Confluent Avro Schema Registration and Serialization and De-serialization routines that are available only with the Confluent Kafka distribution. On a Confluent Open Source Kafka installation on a Linux server, the required libraries are in the following folders:

- /usr/share/java/confluent-common
- /usr/share/java/kafka
- /usr/share/java/kafka-serde-tools

The associated license files are in the following folders:

- /usr/share/doc/confluent-common
- /usr/share/doc/kafka
- /usr/share/doc/kafka-serde-tools

In the example for this and the next section, the JAVA libraries from a Confluent 5.0.0 installation have been downloaded to the OpenVMS directory DISK$STATIC:[kafka.CONFLUENT-5-0-0]

```
$ set file/attr=(rfm:stmlf,rat:cr,lrl:0,mr:0) -
    DISK$STATIC:[kafka.CONFLUENT-5-0-0...]*.jar
```

Kafka Libraries from VMS Software, Inc.

VMS Software Inc. (VSI) Open-Source libraries contain a Kafka V0.9.5.1 installation kit:

- vsi-i64vms-librdkafka-v0009-5-1.zip

JCC testing has only used Kafka V1.0 and later releases.

Kafka Library Location

The best location for the Kafka libraries depends on your environment. The above examples, from the JCC Consulting, Inc. development environment, used a directory structure DISK$STATIC:[kafka...]. This structure works well for an environment where multiple Kafka versions and distributions are being tested with multiple versions of the JCC LogMiner Loader.

In an environment that is relatively static, it may make sense to place the Kafka libraries in the directory tree jcc_tool_root:[java.lib...]. For example:

```
$ dire/size=all/date=create/prot jcc_tool_root:[java.lib.kafka.CONFLUENT-4-1-1]

Directory JCC_TOOL_ROOT:[JAVA.LIB.kafka.CONFLUENT-4-1-1]

confluent-common.DIR;1
               1/16     15-NOV-2018 10:05:26.42  (RWE,RWE,RE,RE)
doc.DIR;1      1/16     15-NOV-2018 10:05:27.27  (RWE,RWE,RE,RE)
kafka-serde-tools.DIR;1
               2/16     15-NOV-2018 10:05:28.07  (RWE,RWE,RE,RE)
kafka.DIR;1    8/16     15-NOV-2018 10:05:28.96  (RWE,RWE,RE,RE)

Total of 4 files, 12/64 blocks.
$
```

VMS Privileges Needed with Avro

According to the “HP TCP/IP Services for OpenVMS Sockets API and System Services Programming” documentation, a “process must have SYSPRV, BYPASS, or OPER privilege to bind port numbers 1 to 1023.”

The Avro Schema Registration uses HTTP or HTTPS, although not on port 80 or 443. However JCC LogMiner Loader testing found that one of these additional privileges is required when using Avro Schema Registration. JCC Recommends adding OPER privilege and none of the others.

JCC Local Environment Command Procedure

The JCC LogMiner Loader supports a procedure to define logical names specific to a local environment called JCC_LOCAL_ENVIRONMENT.COM. See the full documentation for Version 3.6 for details.

If this procedure exists when the JCC LogMiner Loader startup is executed, the startup procedure is executed with a parameter that is the appropriate logical name table.

There is a template procedure in jcc_tool_com that must be copied to jcc_tool_local directory to be used:

```
$ copy jcc_tool_com:JCC_LOCAL_ENVIRONMENT.COM -
    jcc_tool_root:[local]*.*/log
```

The following example shows logical names added when the Confluent Kafka libraries are installed in the JCC_tool_root:[java.lib] directory tree.

JCC_TOOL_ROOT:[local]JCC_LOCAL_ENVIRONMENT.COM can be modified as shown in "Configuring the Local Environment" on page 19.

With those additional lines, the next time the JCC LogMiner Loader startup is executed, the logical names will be added to the appropriate logical name table. In the following example in "Example Startup" on page 20, the V3.6 startup was executed with the MV parameter, so the logical names were added to the JCC_CLML_03_06 logical name table.

Configuring the Local Environment

The following provides the configuration used in the examples.

```
$! Logical Names to support Kafka
$! 17-Nov-2018
$!
$! define paths for use in JCC LogMiner Loader configuration files
$!
$ define/'p1' confluent_kafka_serde                                 -
     jcc_tool_root:[java.lib.kafka.CONFLUENT-4-1-1.kafka-serde-tools]
$ define/'p1' confluent_common                                      -
     jcc_tool_root:[java.lib.kafka.CONFLUENT-4-1-1.confluent-common]
$ define/'p1' confluent_kafka                                       -
     jcc_tool_root:[java.lib.kafka.CONFLUENT-4-1-1.kafka]
```

```
$!
$! define class path logical for use with Kafka command line tools
$!
$ define/'p1' confluent_kafka_path -
     jcc_tool_root:[java.lib.kafka.confluent-4-1-1.kafka-serde-tools], -
     jcc_tool_root:[java.lib.kafka.confluent-4-1-1.confluent-common],  -
     jcc_tool_root:[java.lib.kafka.confluent-4-1-1.kafka]
$!
```

Example Startup

This startup is executed with the MV parameter.[1]

```
$ show log/table=JCC_CLML_03_06

(JCC_CLML_03_06)

  "CONFLUENT_COMMON"
        = "JCC_TOOL_ROOT:[JAVA.LIB.KAFKA.CONFLUENT-4-1-1.CONFLUENT-COMMON]"
  "CONFLUENT_KAFKA"
        = "JCC_TOOL_ROOT:[JAVA.LIB.KAFKA.CONFLUENT-4-1-1.KAFKA]"
  "CONFLUENT_KAFKA_PATH"
        = "JCC_TOOL_ROOT:[JAVA.LIB.KAFKA.CONFLUENT-4-1-1.KAFKA-SERDE-TOOLS]"
        = "JCC_TOOL_ROOT:[JAVA.LIB.KAFKA.CONFLUENT-4-1-1.CONFLUENT-COMMON]"
        = "JCC_TOOL_ROOT:[JAVA.LIB.KAFKA.CONFLUENT-4-1-1.KAFKA]"
  "CONFLUENT_KAFKA_SERDE"
        = "JCC_TOOL_ROOT:[JAVA.LIB.KAFKA.CONFLUENT-4-1-1.KAFKA-SERDE-TOOLS]"
  "JAVA$JCC_LOGMINER_LOADER_JDBC2_SHARE_SHR"
        = "JCC_LOGMINER_LOADER_JDBC2_SHARE"
  "JCCLML_INSTALLED_VERSION"
        = "T03.06.00"
  "JCCLML_LINK_DATETIME"
        = " 9-NOV-2018 16:41:23.39"
  "JCC_LOGMINER_LOADER_BASE_SHARE"
        = "JCC_TOOL_SHARE:JCC_LOGMINER_LOADER_BASE_SHARE.EXE"
 ...
```

1. Spacing and line feeds are added to improve readability.

CHAPTER 3

Control File

The Administrator(s) for the JCC LogMiner Loader control the actions of the Loader through a set of logical names[1] and through the Control File.

The Control File is made up of keywords. Each keyword has specific syntax and may have both required and optional parameters.

For those not familiar with the JCC LogMiner Loader, it may be important to read the first few sections of the Control File chapter in the full documentation, as well as any sections related to specific keywords that are referenced here.

This chapter includes

- Update to the Output keyword
- Section for each keyword introduced to support the Kafka Option

1. Logical names are discussed in the full documentation and in "Logical Names" on page 41.

- Section that mentions keyword enhancements designed for the Kafka Option and added to the general release
- Section that describes enhancements to the procedure that can be used to generate a Control File.

Output Keyword and Kafka

The output keyword specifies the kind of target data store the Loader will be maintaining.

Syntax

```
OUTPUT~<output type>[~<synchronous>[~<output target> \
[~<message contents>[~<output conversion>]]]
```

Parameters

<output type>. The Kafka output type is required for the Kafka Option.

Remaining parameters. The remaining parameters for the Output Keyword are generally optional. However, for Kafka, there are two specifics.

The synchronous/asynchronous choice used with some Loader topics is inappropriate. Synchronous is used whether or not it is specified, although Kafka itself supports much asynchronous activity.

The output conversion must be specified. Options are XML, JSON, and AVRO. Examples follow.

See the full documentation for specifics of the other parameters.

Example

```
output~Kafka~synch~connect~record~JSON
```

Additional examples are available in “Output Format Examples” on page 45.

Keyword: Kafka

All keywords that begin with Kafka are related solely to the Kafka Option. Some of them may have parallels for other target options.

Syntax

```
Kafka~<attribute>~<value>
```

Parameters

<attribute>. The following attributes are supported. Each is described as a separate keyword in the following sections.

- CONNECT (optional)
- CLASSPATH (required)
- TOPIC (optional)
- MODEL (optional)
- HEADER (optional)
- AVRO (required, if output conversion is AVRO)

<value>. The value options vary with the attribute. See the following sections for details.

Keyword: Kafka~connect

Optionally, define the Kafka broker list.

Syntax

```
Kafka~connect~<Kafka boot servers>
```

Parameters

<Kafka boot servers>. This is a comma separated list of bootservers and the aso-ciated ports.

Example

See “Define connect” on page 52 and other examples in that appendix.

Keyword: Kafka~classpath

Definition of the classpath is required when using the Kafka Option.

Syntax

```
Kafka~classpath~<required jar file for JDBC driver>
```

Parameters

<required jar file for JDBC driver>. This should be the path to the Kafka jar files. Note that the Kafka classpath must be specified in OpenVMS format with the wildcard.

Example

```
kafka~classpath~DISK$STATIC:[KAFKA.1-1-0.LIBS]*.jar
```

See also “Define the classpath” on page 46 and “Define the classpath” on page 49.

Keyword: Kafka~topic

Optionally, define the topics. Note that the topic names must be coordinated with the Kafka application.

Syntax

```
Kafka~topic~<tag>|<quoted constant>[,tag|<quoted constant>  \
[,tag|<quoted constant>]...]
```

Parameters

<tag>|<quoted constant>. The topics are either tags or quoted constants.[1] These are concatenated into a string. One topic is required (if the keyword is used), additional ones are optional. Valid characters are any alphanumeric character, plus, period (‘.’), underscore (‘_’), and dash (‘-’). Maximum length is 249 characters.

Examples

The following examples show values for <topic> and the resulting topic name. The first row is the default. The table name (or the text that is it renamed[2]) will be used as the topic name, if no other is provided.

TABLE 1. Topics and Resultant Text

Topic	Text for Topic Name
table_name	“<MapTable [re]name>”
'xml.',Table_Name	"xml.<MapTable [re]name>"
Loadername, ‘.json.’, Table_Name	“<LoaderName>.json.<MapTable [re]name>”
Loadername,’.’, Table_Name	“<LoaderName>.<MapTable [re]name>”
Table_name,Loadername	“<MapTable [re]name><LoaderName>
‘SingleTopicforAll’	“SingleTopicforAll”

The topic naming must be coordinated with the Kafka consumers.

Keyword: Kafka~model

The Kafka Model is also discussed in “Kafka Model” on page 39. This is an optional parameter. Transaction, as described in that section is the default. This keyword is not required unless a different model is desired.

1. Quoted constants must use single quotes.
2. Renaming is done with the target table rename parameter of either the Table Keyword or the MapTable Keyword.

Syntax

```
Kafka~model~<choice of model>
```

Parameters

Attribute. Model types are Transaction and ExactlyOnce. Transaction is the default.

Value. The value is dependent on the attribute.

TABLE 2. **Avro Attributes and Values**

Attribute	**Value**
Transaction	Use start/commit/rollback transactions
ExactlyOnce	Use Kafka ExactlyOnce semantics

Examples

```
Kafka~model~transaction
Kafka~model~ExactlyOnce
```

Note that, since TRANSACTION is the default, the first example changes nothing.

Keyword: Kafka~header

In some Kafka environments, there is a requirement to publish headers with each message. The JCC LogMiner Loader Kafka Option provides configuration directives to customize the header information that will be published.

Syntax

```
Kafka~header~name~<tag>|<quoted constant>                 \
[,tag|<quoted constant>[,tag|<quoted constant>]...]
```

Parameters

name. <a string to name the header>

<tag>|<quoted constant>. The topics are either tags or quoted constants.[1] These are concatenated into a string. One header is required (if the keyword is used), additional ones are optional. Valid characters are any alphanumeric character, plus, period ('.'), underscore ('_'), and dash ('-'). Maximum length is 249 characters.

Examples

The following example is from the Kafka regression test:

```
!
! The following are published with messages as the
! Kafka Message Header
!
kafka~header~DF_SCHEMA~'RegTest.',Table_Name
kafka~header~DF_SOURCE~'JCCRegTest'
kafka~header~DF_ENTITY~Table_Name
kafka~header~DF_UTID~loader_sequence_number
kafka~header~DF_UTID_T~'',loader_sequence_number-
kafka~header~DF_ACTION~action
!
! DF_UTID will be sent as a 8 byte numeric
! DF_UTID_T will be sent as a text string because it
!  contains multiple values
! DF_ACTION will contain M (modify) or D (delete)
!
```

The names (DF_SCHEMA, etc.) are arbitrary strings. The values can be quoted strings and/or most of the JCC LogMiner Loader virtual columns[2]. Multiple items must be separated by commas.

Keyword: Kafka~Avro

If output~Kafka~...~AVRO is used, it must be followed by two other keyword structures that are required.

1. Quoted constants must use single quotes.
2. See "Extensions to Existing Keywords" on page 30 and the VirtualColumn keyword in the Control File chapter of the full documentation.

Syntax

```
Kafka~Avro~<attribute>~<value>
```

Parameters

Attribute. Recognized attributes are SchemaRegistry and NameSpace. Both are required.

Value. The value is dependent on the attribute.

TABLE 3. **Avro Attributes and Values**

Attribute	**Value**
SchemaRegistry	<Avro Schema Registry URL>
NameSpace	<Avro NameSpace>

The Avro NameSpace qualifies schema definitions to prevent name collisions between different sources. It is the name of the location within the Avra Schema Registry where the record definitions for the tables the Loader sends will be saved.

Examples

See an extensive AVRO example in “AVRO Format for Kafka” on page 50.

Keyword: XML

XML is a supported option for the <output conversion> parameter of the output keyword. XML can be used with multiple types of targets, including Kafka targets.

The XML keyword is not generally used because the defaults that are defined are highly likely to be what is wanted. See the full documentation for more discussion on this.

Attributes that were new with the JCC LogMiner Loader Version 3.6 were introduced because of their importance in some Kafka implementations.

Header Syntax

By default, the Loader includes a header for XML output. The header has two parts: one called Prolog and one called DOCTYPE.

Prolog consists of "<?xml version='1.0'?>" which specifies the version of the XML standard that is used for this XML document.

DOCTYPE consists of "<!DOCTYPE pkt SYSTEM 'packet-commented.dtd'>". The JCC LogMiner Loader DTD's can be found in JCC_TOOL_-ROOT:[SOURCE].

Because some Kafka related products do not support this header, the Loader has been modified to include a keyword for the XML header. The syntax has four options.

TABLE 4. **XML Header Options**

Syntax	Result
`XML~Header~Prolog,DOCTYPE`	The default includes both parts of the header.
`XML~Header~`	Includes neither part of the default header.
`XML~Header~Prolog`	Include the part of the default header called prolog, but not the part called DOCTYPE
`XML~Header~DOCTYPE`	Include the part of the default header called DOCTYPE, but not the part called Prolog

NULL Syntax

There are two alternative values for the XML keyword to indicate whether to show columns with NULL values.

```
XML~NULL~explicit|implicit
```

Explicit is the default and the specification is not required, unless implicit is desired. Explicit lists null columns.

Implicit causes the Loader to not list null columns. However, there are actually three possible results, depending on how the column itself is defined. Keywords for Column and for MapColumn include the option of defining <value if null>. If a value definition for null is specified, that value replaces NULL. In which case, for this column, implicit and explicit have the same effect.

Keyword: JAVA

The Keyword JAVA is used to define a file to be read by JAVA and the Loader.

Syntax

```
JAVA~PROPERTIES~<filename>
```

Parameters

The file named must be readable by JAVA and the Loader.

Example

See "Java Properties File" on page 35.

Extensions to Existing Keywords

VirtualColumn and MapColumn are standard keywords for the Loader. Version 3.6 of the Loader which is released along with the Kafka Option introduces wildcarding for VirtualColumns and MapColumns. The full documentation for the JCC LogMiner Loader Version 3.6 and the release notes for Version 3.6 described wildcarding.

Creating the Control File and Kafka Topics

The command JCC_LML_CREATE_CONTROL_FILE is provided with the standard Loader kit. The procedure it names can be used to create a Control File. Version 3.6 adds a Kafka specific option for this routine: KAFKA_TOPICS.

Using KAFKA_TOPICS creates a file named <db file>_CREATE_TOPICS.COM with command line statements to create Kafka topics. This is useful for Kafka environments that require topics to explicitly be created. The commands can be modified by defining DCL symbols prior to running the procedure.

The symbols and their default values are:

Item	**Default Setting**
JCC_KAFKA_TOPIC	"$ java -cp /confluent_kafka_path/*:. kafka.admin.TopicCommand"
JCC_ZOOKEEPER_ADDRESS	"cnflnt4-04:2181"
JCC_TOPIC_PREFIX	""
JCC_TOPIC_PARTITIONS	"6"
JCC_TOPIC_REPLICATION	"2"

These symbols do not need to be defined if the default values are correct for a specific application.

The syntax for creating the KAFKA_TOPICS procedure is:

```
$ JCC_LML_CREATE_CONTROL_FILE <db> KAFKA_TOPICS
```

Kafka Topics: Example 1

The following example shows the generation of a KAFKA_TOPICS command procedure using the default settings and part of the generated procedures.

```
$ jcc_lml_create_control_file mfp_db kafka_topics
JCC_ROOT:[KEITH.SQL_CLASS.LML_KAFKA]MF_PERSONNEL_CREATE_TOP
   ICS.COM;15 33 lines
[EOB]
%DELETE-I-FILDEL, JCC_ROOT:[KEITH.SQL_CLASS.LML_KAFKA]MF_PER
   SONNEL_CREATE_TOPICS.COM;14 deleted (515 blocks)
$ type MF_PERSONNEL_CREATE_TOPICS.COM

$!
$!  JCC LogMiner Loader Kafka Create Topic procedure
$!  Generated at 2018-11-26 11:10:13
$ java -cp /confluent_kafka_path/*:.                 -
      kafka.admin.TopicCommand                       -
      --zookeeper cnflnt4-04:2181                    -
```

```
     --create                                           -
     --topic CANDIDATES                                 -
     --partitions 6                                     -
     --replication-factor 2
$ java -cp /confluent_kafka_path/*:.                    -
     kafka.admin.TopicCommand                           -
     --zookeeper cnflnt4-04:2181                        -
     --create                                           -
     --topic COLLEGES                                   -
     --partitions 6                                     -
     --replication-factor 2
$ java -cp /confluent_kafka_path/*:.                    -
     kafka.admin.TopicCommand                           -
     --zookeeper cnflnt4-04:2181                        -
     --create                                           -
     --topic DEGREES                                    -
     --partitions 6                                     -
     --replication-factor 2
$ java -cp /confluent_kafka_path/*:.                    -
     kafka.admin.TopicCommand                           -
     --zookeeper cnflnt4-04:2181                        -
     --create                                           -
     --topic DEPARTMENTS                                -
     --partitions 6                                     -
     --replication-factor 2
$ java -cp /confluent_kafka_path/*:.                    -
     kafka.admin.TopicCommand                           -
     --zookeeper cnflnt4-04:2181                        -
     --create                                           -
     --topic EMPLOYEES                                  -
     --partitions 6                                     -
     --replication-factor 2
...
```

Kafka Topics: Example 2

This example changes the default value for JCC_ZOOKEEPER_ADDRESS to a different Kafka node and JCC_TOPIC_PREFIX to include prefix information:

```
$ JCC_ZOOKEEPER_ADDRESS = "confluent01:2181"
$ JCC_TOPIC_PREFIX = "JCC.MFP."
$ jcc_lml_create_control_file mfp_db kafka_topics
JCC_ROOT:[KEITH.SQL_CLASS.LML_KAFKA]MF_PERSONNEL_CREATE_TOP
   ICS.COM;19 33 lines
[EOB]
%DELETE-I-FILDEL, JCC_ROOT:[KEITH.SQL_CLASS.LML_KAFKA]MF_PER
   SONNEL_CREATE_TOPICS.COM;18 deleted (515 blocks)
$ type MF_PERSONNEL_CREATE_TOPICS.COM
$!
$!  JCC LogMiner Loader Kafka Create Topic procedure
$!  Generated at 2018-11-26 11:18:20
$ java -cp /confluent_kafka_path/*:.                 -
     kafka.admin.TopicCommand                        -
     --zookeeper confluent01:2181                    -
     --create                                        -
     --topic JCC.MFP.CANDIDATES                      -
     --partitions 6
     --replication-factor 2
$ java -cp /confluent_kafka_path/*:.                 -
     kafka.admin.TopicCommand                        -
     --zookeeper confluent01:2181                    -
     --create                                        -
     --topic JCC.MFP.COLLEGES                        -
     --partitions 6                                  -
     --replication-factor 2
$ java -cp /confluent_kafka_path/*:.                 -
     kafka.admin.TopicCommand                        -
     --zookeeper confluent01:2181                    -
     --create                                        -
     --topic JCC.MFP.DEGREES                         -
     --partitions 6                                  -
     --replication-factor 2
$ java -cp /confluent_kafka_path/*:.                 -
     kafka.admin.TopicCommand                        -
     --zookeeper confluent01:2181                    -
     --create                                        -
     --topic JCC.MFP.DEPARTMENTS                     -
     --partitions 6                                  -
     --replication-factor 2
```

```
$ java -cp /confluent_kafka_path/*:.                              -
     kafka.admin.TopicCommand                                     -
     --zookeeper confluent01:2181                                 -
     --create                                                     -
     --topic JCC.MFP.EMPLOYEES                                    -
     --partitions 6                                               -
     --replication-factor 2
$ java -cp /confluent_kafka_path/*:.                              -
     kafka.admin.TopicCommand                                     -
     --zookeeper confluent01:2181                                 -
     --create                                                     -
     --topic JCC.MFP.EMPLOYEES_TEST                               -
     --partitions 6                                               -
     --replication-factor 2
$ java -cp /confluent_kafka_path/*:.                              -
     kafka.admin.TopicCommand                                     -
     --zookeeper confluent01:2181                                 -
     --create                                                     -
     --topic JCC.MFP.Employees1                                   -
     --partitions 6                                               -
     --replication-factor 2
...
```

Note that the default values for KAFKA_TOPIC need to be changed to match the requirements for the Kafka Server in your environment.

CHAPTER 4

Notes for the Loader Administrator

This chapter includes a number of other discussions pertinent to successful implementation of the JCC LogMiner Loader Kafka Options. The full documentation for the Loader will provide additional understanding, particularly the chapters on Performance Consideration and Aids to the Administrator.

Java Properties File

The following example Properties file contains two Kafka Producer parameters that have been modified to tune a particular Kafka workload:

```
#
#  kafka_avro.properties
#
# 2018-10-19 kwh
# increase batch.size and linger.ms in an attempt to increase throughput
#
batch.size=65536
linger.ms=25
#
```

Batch Size

The example shows batch.size=65536. Batch size is the number of bytes to collect before sending messages to the Kafka broker. The default is 16384. Increasing this can delay message transmission and/or increase overall throughput.

Linger.ms

The example shows linger.ms=25. Linger.ms is the number of milliseconds to wait before sending a buffer of messages to the Kafka broker. The Kafka default is to send immediately when data is available, but the JCC LogMiner Loader sets this value to 20. Increasing linger.ms will reduce the number of network packets but may increase message latency.

Tuning with the Java Properties File

In each case, increasing the default *may* increase overall throughput, but delay the transmission of an individual message. The appropriate value for these producer parameters is highly dependent on the mix and frequency of the source transactions as well as the configuration of the network and the distance of the Kafka server, where the relevant distance is round-trip ping packet time between the OpenVMS system and the Kafka server.

SSL Encryption

Kafka supports creating a connection between a consumer and a Kafka server either using plaintext or an encrypted SSL[1] connection. The SSL connection can be used just to encrypt the data while it is on the network or to both encrypt and authenticate the Kafka producer.

The details of how a Kafka server is configured must be provided by whomever configures and manages the Kafka server.

1. Technically, Kafka supports TLS (Transport Layer Security) but the Kafka documentation refers to it as SSL (Secure Socket Layer).

The configuration file statements specific to SSL are listed following the example.

```
!
! Define the Kafka Avro Specific stuff
!
!
output~kafka~synch~connect~record~Avro
!
kafka~connect~confluent01:9093
kafka~avro~SchemaRegistry~http://confluent01:8081
!
kafka~avro~namespace~RegTest.avro
!
!  routine to expand wildcards does not work on unix-style names.
!
kafka~classpath~disk$static:[kafka.CONFLUENT-4-1-1.kafka-serde-tools]*.jar
kafka~classpath~disk$static:[kafka.CONFLUENT-4-1-1.confluent-common]*.jar
kafka~classpath~disk$static:[kafka.CONFLUENT-4-1-1.kafka]*.jar
!
kafka~Topic~'RegTest.',Table_Name
kafka~Model~ExactlyOnce
!
! External Properties File
!
java~properties~/control_files/kafka_avro.properties
```

The two differences in this example are:

- kafka~connect~confluent01:9093
    - Port 9093 on the Kafka server confluent01 has been configured to accept SSL connections
- kafka~avro~SchemaRegistry~http://confluent01:8081
    - The Confluent Schema Registry is using the unencrypted http:// protocol against port 8081 on Confluent01
    - It is possible to configure the Schema Registry to use https

The SSL parameters are specified in the Java Properties file kafka_avro.properties

Java Properties File with SSL

The following example Java Properties file adds SSL information to both authenticate the producer to the Kafka broker and to encrypt the Kafka traffic on the network connection.

```
#
#  kafka_avro.properties
#
```

```
# 2018-10-19 kwh
# increase batch.size and linger.ms in an attempt
# to increase throughput
#
batch.size=65536
linger.ms=25
#
# SSL Configuration
#
security.protocol=ssl
ssl.truststore.location=/regtest_ssl/kafka_truststore.jks
ssl.truststore.password=<password 1>
ssl.keystore.location=/regtest_ssl/kafka_keystore.jks
ssl.keystore.password=<password 2>
ssl.key.password=<password 3>
#
```

The SSL truststore and keystore files and passwords must be provided by the Kafka administrator.

Because this file contains passwords, the OpenVMS file protection and access control should be set so that the file is accessible to the JCC LogMiner Loader session, but not to unprivileged OpenVMS users.

In this example, REGTEST_SSL is a logical name that is defined in the command procedure that starts the JCC LogMiner Loader job. Alternately, it could be defined somewhere else in the environment. In this example, the command procedure contains:

```
$ define/nolog regtest_ssl -
      JCC_ROOT:[KAFKA_XML.JCC.DBA.REGRESSION_TEST.SSL]
```

The directory contains three files:

```
Directory JCC_ROOT:[KAFKA_XML.JCC.DBA.REGRESSION_TEST.SSL]
ca-key.;1                 2KB/2KB       18-OCT-2018 11:46:49.93  (RWED,RWED,RE,)
kafka_keystore.jks;1      4KB/5KB       18-OCT-2018 11:46:50.47  (RWED,RWED,RE,)
kafka_truststore.jks;1    1KB/2KB       18-OCT-2018 11:46:50.00  (RWED,RWED,RE,)
```

These are binary files that were provided by the Kafka administrator.

These files can be stored in the JCC_TOOL_LOCAL: directory or in some other location that makes sense for the environment.

Additional Tuning for Kafka Producer Performance

Often, JCC develops performance tuning recommendations from the regression testing. The biggest influence on performance with the Kafka Option is checkpoint interval. Checkpoint is a Control File keyword that enables grouping input transactions for output. For example,

```
CHECKPOINT~50~lml_internal~asynch~<checkpoint filename>
```

causes up to 50 source transactions to be committed to Kafka in one Kafka transaction.

Tuning and interpretation of performance may include:

- Balancing batch.size and linger.ms and checkpoint to achieve the best results for overall throughput versus writing each transaction as soon as possible. Checkpoint is defined in the Control File and the full documentation has more discussion of it. The batch.size and linger.ms parameters are defined in the Java Properties File and are discussed in "Java Properties File" on page 35.
- Kafka Option throughput is highly dependent on the source database transaction workload, including both the overall volume of data and the pattern of updates.
- Physical characteristics of the network can cause performance bottlenecks.
- The cluster configuration and the number of partitions and retention time for a topic can also influence performance.

Future performance recommendations will be available in documentation updates.

Kafka Model

Kafka documents these models for publishing messages.

- ExactlyOnce
- Transaction

The current release of the JCC LogMiner Loader supports both.

The Kafka Model is also described as a keyword in “Keyword: Kafka~model” on page 25.

ExactlyOnce

ExactlyOnce semantics are usually associated with a consumer that also is a publisher. ExactlyOnce is the 2PC (2 Phase Commit) between the consumer topic offsets and the messages written to the producer topic(s). For the Loader to (almost) guarantee this literal definition requires using only 1 thread. The 'almost' is because the JCC LogMiner Loader checkpoint file is not maintained in a 2PC transaction with the Kafka transaction. There is a small time window between the point at which a Kafka Transaction is committed and the point at which the Rdb LogMiner restart information is written. If a failure occurred during this time, when the JCC LogMiner Loader restarts, data that had previously been committed to Kafka will be resent.

When the JCC LogMiner Loader is configured to use multiple threads, the JCC LogMiner Loader checkpoint contains multiple checkpoints, one for each thread. On a restart, the JCC LogMiner Loader starts from the oldest checkpoint information and so may potentially resend data that had been committed to Kafka prior to the failure that preceded the restart.

For more information about how Kafka implements ExactlyOnce, see the blog:

https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/

The section titled “Wait, but what’s exactly-once for non-deterministic operations anyway?” in the referenced blog is more like the 'ExactlyOnce' provided by the JCC LogMiner Loader Kafka Option.

Transaction

Kafka transactions are committed using the Checkpoint specified in the JCC LogMiner Loader configuration file. These occur at source database transaction boundaries. The checkpoint value specifies how many source transactions will be bundled into a single Kafka transaction.

Logical Names

Logical names provide additional control of the Loader's actions. Logical names are mentioned often in the full documentation.

Logical names that are specifically useful for the Kafka Option include these that are also available for the JDBC target.

- JCC_LML_JAVA_BOOTCLASSPATH
- JCC_LML_JAVA_COMMAND_LINE

Kafka Partitions and Keys

When the JCC LogMiner Loader publishes a message to a Kafka server, it uses the formatted DBKey of the source as the Kafka key. This ensures that messages for the same source record go to the same topic partition. Within Kafka, messages on the same partition will be consumed in the order in which they are published to the partition. Messages in different partitions may be consumed in different orders.

This example shows that the DETAILS topic on the server kafka01 has 5 partitions:

```
$ java kafka.admin.TopicCommand --zookeeper kafka01:2181 --topic DETAILS -
_$     --describe
Topic:DETAILS   PartitionCount:5        ReplicationFactor:1     Configs:reten-
tion.ms=14400000
        Topic: DETAILS  Partition: 0    Leader: 0       Replicas: 0     Isr: 0
        Topic: DETAILS  Partition: 1    Leader: 0       Replicas: 0     Isr: 0
        Topic: DETAILS  Partition: 2    Leader: 0       Replicas: 0     Isr: 0
        Topic: DETAILS  Partition: 3    Leader: 0       Replicas: 0     Isr: 0
        Topic: DETAILS  Partition: 4    Leader: 0       Replicas: 0     Isr: 0
$
```

The appropriate number of partitions for a topic depends on a number of factors including the Kafka cluster configuration, the volume of data that will be published, and how the data will be consumed.

Note that this example shows a retention of 14,400,000 milliseconds (4 hours). Messages published to this topic will be retained by Kafka for 4 hours before they are discarded. In this example, the retention time was explicitly set with the command:

```
$ java  kafka.admin.TopicCommand --zookeeper kafka01:2181 -
_$  --topic DETAILS --alter --config retention.ms=14400000
```

By default, Kafka retains messages for 168 hours. By default, Kafka checks log segments to see if they need to be retained every 300 seconds. Both the default retention and the interval between log segment checks can be configured in the Kafka server configuration file.

The appropriate retention time for a topic depends on a number of factors including the Kafka cluster configuration, the volume of data that will be published, how the data will be consumed, and the latency of the consumer.

Date and Timestamp Precision with Avro

The JCC LogMiner Loader converts the OpenVMS timestamp data type to the Avro LogicalType timestamp-micros data type.

The Avro LogicalType timestamp-micros has a limitation in that it represents, in a 64-bit integer, the number of microseconds (10^-6) from 1970-01-01T00:00:00.00Z. Since OpenVMS uses 10^-7 increments from 1858-11-17T00:00:00.00Z, there is much more headroom with Avro timestamp-micros, but at the loss of precision in the 7th digit after the decimal point. In Avro, negative values for the number of microseconds are dates before 1970, whereas OpenVMS uses delta dates.

That means that the valid range of values for Avro far exceeds that of OpenVMS.[1]

- Minimum: -290308-12-21T19:59:05.224192Z
- Maximum +294247-01-10T04:00:54.775807Z

Method	**Incr.**	**Starting Date**	**Early Dates**	**Advantage**
AVRO	10^-6	1970-01-01T00:00:00.00Z	negative values for time before start	range
OpenVMS	10^-7	1858-11-01T00:00:00.00Z	delta dates	precision

1. There is much more that can be said about date time variance, including a difference in clock ticks on different systems, but this is sufficient for understanding the Kafka Option for JCC LogMiner Loader.

For most applications, the difference in precision between OpenVMS timestamps and Avro timestamp-micros is small enough that it will be unnoticed.

Comparison of Rdb and Avro Datatypes

The following table documents how Rdb data types are mapped to Avro types.

Rdb SQL Type	**Prec**	**Scale**	**Avro Type**
BIGINT	20	= 0	long
		> 0	{"type":"bytes","logicalType":"decimal","precision":<precision>, "scale":<scale>}
		< 0	double
INTEGER	10	= 0	int
		> 0	{"type":"bytes","logicalType":"decimal","precision":<precision>, "scale":<scale>}
		< 0	double
SMALLINT	5	= 0	int
		> 0	{"type":"bytes","logicalType":"decimal","precision":<precision>, "scale":<scale>}
		< 0	double
TINYINT	3	= 0	int
		> 0	{"type":"bytes","logicalType":"decimal","precision":<precision>, "scale":<scale>}
		< 0	double
DOUBLE PRECISION			double
FLOAT			float
CHAR, VARCHAR			string
BINARY, VARBINARY			bytes
TIMESTAMP, DATE VMS, DATE ANSI			{"type":"long","logicalType":"timestamp-micros"}
DATE			{"type":"int","logicalType":"date"}
TIME			{"type":"long","logicalType":"time-micros"}

See http://avro.apache.org/docs/1.8.0/spec.html for definitions of logicalTypes.

CHAPTER 5

Output Format Examples

The JCC LogMiner Loader can publish messages to a Kafka server in one of three formats:

- XML
- JSON
- AVRO

The remainder of this chapter gives examples and comments pertinent to each of these. Note that the examples use different settings for various options to illustrate the possibilities. The material supplied here is a set of examples. The formal definition of syntax for Kafka specific keywords is in “Control File” on page 21 and the formal definition of other keywords may be found in the full documentation.

You are unlikely to use more than one format, but there is no restriction against separate Loader sessions using different ones of these and all running simultaneously.

XML Format for Kafka

This section provides an annotated example to illustrates some possible configurations.

Kafka Specific Configuration Directives - XML

The following shows an example of JCC LogMiner Loader configuration directives to publish changes to a Kafka server as XML documents.

```
! Define the Kafka XML Specific stuff
!
! Note that the Kafka classpath *must* be specified in
! OpenVMS format with the wildcard
!
output~kafka~synch~kafka01:9092~record~xml
kafka~classpath~DISK$STATIC:[KAFKA.1-1-0.LIBS]*.jar
!
! The following XML configuration and date format
! were added to support a specific Kafka consumer.
xml~header~
xml~null~implicit
date_format~|!Y4-!MN0-!D0 !H04:!M0:!S0.!C6|
```

Define the target. Use the output keyword to define the target (output type) and other characteristics. This is required.

```
output~kafka~synch~kafka01:9092~record~xml
```

- The Loader target is “kafka”
- Sync is an output keyword option discussed in the full documentation.
- The Kafka server is the node “kafka01” using port 9092
- The output record type is XML

Define the classpath. Provide the OpenVMS disk for finding the jar files. This is required.

```
kafka~classpath~DISK$STATIC:[KAFKA.1-1-0.LIBS]*.jar
```

- The classpath for the Kafka jars is
  DISK$STATIC:[KAFKA.1-1-0.LIBS]*.jar
  This is the OpenVMS directory in the example above.

Define the XML header. Defining a header is optional. By default, a header is provided.

```
xml~header~
```

The format shown in this example excludes the default header. See “Header Syntax” on page 29. The XML Header information was removed in this example to meet the requirements of a specific Kafka consumer.

Define how to handle NULLS. This is optional. The default is explicit which includes NULL or the value indicated elsewhere.[1]

```
xml~null~implicit
```

- With the example showen, NULLs are excluded when a column in the Rdb source is NULL and there is no definition of the column that includes a value to use instead of NULL.
- Setting xml~null~explicit would send “val=NULL” for a column when the Rdb source is NULL and there is no definition of the columns that includes a value to use instead of NULL.

Define the date format. Defining the date format is optional. See also “Date and Timestamp Precision with Avro” on page 42 and “Comparison of Rdb and Avro Datatypes” on page 43.

```
date_format~|!Y4-!MN0-!D0 !H04:!M0:!S0.!C6|
```

- This Formatted Output mask matches the default format used for timestamps.[2]
- If only two faction digits of preciesion for seconds are required, use: date_format~|!Y4-!MN0-!D0 !H04:!M0:!S0.!C2|

XML Message Example

Messages are published to Kafka as a single Unicode document -- a single line. The following example of an XML document is formatted for readability:

```
<pkt>
  <row name="DETAILS" actn="M">
    <col name="DETAILS_ID" type="num" val="371596"/>
```

1. See “NULL Syntax” on page 29 or the Version 3.6 documentation or the 3.6 release notes for a more complete discussion.
2. See Keyword: Date_format in the Control File chapter of the full documentation.

```
    <col name="PROGNAME" type="str" len="31" val="DETAILS"/>
    <col name="PROGINDEX" type="str" len="3" val="005"/>
    <col name="SEQNUM" type="num" val="1"/>
    <col name="AMOUNT_F" type="num" val="0.000000000e+00"/>
    <col name="AMOUNT_G" type="num" val="  0.0000000000000000e+00"/>
    <col name="AMOUNT_N" type="num" val="0.00"/>
    <col name="DATE_VMS" type="dt" val="1918-07-02 10:47:05.936995"/>
    <col name="DATE_ANSI" type="dt" val="2002-10-28 00:00:00.000000"/>
    <col name="TIMESTAMP_A" type="dt" val="2058-04-09 15:53:11.460000"/>
    <col name="ORIGINATING_DBKEY" type="num" val="29273399110598677"/>
  </row>
</pkt>
```

JSON Format for Kafka

This section provides an example and explanations for Kafka use of the JSON format.

Kafka Specific Configuration Directives - JSON

The following example, from JCC regression testing, shows the JCC LogMiner Loader configuration directives to publish changes to a Kafka server as JSON documents:

```
!
! Define the Kafka JSON Specific directives
!
! Note that the Kafka classpath *must* be specified in
! OpenVMS format with the wildcard
!
output~kafka~synch~kafka03:9092~record~json
!kafka~classpath~DISK$STATIC:[KAFKA.1-1-0.LIBS]*.jar
kafka~classpath~DISK$STATIC:[KAFKA.2-0-0.LIBS]*.jar
!
JSON~NULL~EXPLICIT
JSON~SCHEMA~REGTEST
!
```

Define the target. Use the output keyword to define the target (output type) and other characteristics. This is required and each parameter shown except synch is required.

```
output~kafka~synch~kafka03:9092~record~json
```

- The Loader target is “kafka”

- See the full doc for an explanation of synch.
- The Kafka server is the node “kafka03” using port 9092
- The output record type is JSON

Define the classpath. Provide the OpenVMS disk for finding the jar files. This is required.

```
kafka~classpath~DISK$STATIC:[KAFKA.2-0-0.LIBS]*.jar
```

- The classpath for the Kafka jars is DISK$STATIC:[KAFKA.2-0-0.LIBS]*.jar
  This is using the JAR files from the Kafka 2.0 distribution.
- The asterisk is the required wildcard.

Define how to handle NULLS. This is optional. The default is explicit which includes NULL or the value indicated elsewhere.[1]

```
xml~null~implicit
```

- With the example showen, NULLs are excluded when a column in the Rdb source is NULL and there is no definition of the column that includes a value to use instead of NULL.
- Setting xml~null~explicit would send “val=NULL” for a column when the Rdb source is NULL and there is no definition of the columns that includes a value to use instead of NULL.

Define the schema. Schema definition is optional. The schema, if named, will be included in the document header.

`JSON~SCHEMA~REGTEST`[2]

- The JSON document header will include
  `"schema_name": "REGTEST"`

JSON Message Example

The following example is a JSON document formatted for readability:

1. See “NULL Syntax” on page 29 or the Version 3.6 documentation or the 3.6 release notes for a more complete discussion.
2. Examples are derived from JCC regression testing. “REGTEST” and other naming relates to that testing.

```
{
  "row": {
    "source_name": "KAFKA_JSON2",
    "schema_name": "REGTEST",
    "name": "DETAILS",
    "action": "M",
    "loader_sequence_number": 34020,
    "column_values": {
      "DETAILS_ID": 605335,
      "PROGNAME": "DETAILS",
      "PROGINDEX": "008",
      "SEQNUM": 1,
      "AMOUNT_F": 3.163302002e+02,
      "AMOUNT_G": 3.1633020019531250e+02,
      "AMOUNT_N": 316.00,
      "DATE_VMS": "2053-02-02 07:14:19.8313170",
      "DATE_ANSI": "2008-04-10 00:00:00.0000000",
      "TIMESTAMP_A": "1921-05-03 12:31:36.3300000",
      "ORIGINATING_DBKEY": 29273399475372062,
    }
  }
}
```

AVRO Format for Kafka

From Wikipedia https://en.wikipedia.org/wiki/Apache_Avro

Avro is a remote procedure call and data serialization framework developed within Apache's Hadoop project. It uses JSON for defining data types and protocols, and serializes data in a compact binary format. Its primary use is in Apache Hadoop, where it can provide both a serialization format for persistent data, and a wire format for communication between Hadoop nodes, and from client programs to the Hadoop services.

While this explanation references Hadoop, Avro is equally useful for Kafka. For Kafka, the important points are:

- The schema registry enables a Kafka consumer to get a description of a record.

- The data serialization means that the message published to Kafka is compressed.
- The consumer must reference the appropriate schema registry information and de-serialization routine to uncompress the message.

Avro and Schema Registration

The JCC LogMiner Loader Kafka option supports publishing messages in Avro format with schema registration when the Kafka server includes the Confluent Schema Registration option. The JCC LogMiner Loader uses the Confluent serialization routine to compress the message. The consumer must use the Confluent schema registry information and the Confluent de-serialization routine to retrieve the message as a JSON document.

Avro Configuration File Example:

The following example is a configuration file to set up output to a Kafka cluster using Avro and Schema Registration.

```
!
! Define the Kafka Avro Specific elements
!
output~kafka~synch~connect~record~Avro
kafka~connect~cnflnt4-04:9092,cnflnt4-03:9092,cnflnt4-02:9092
kafka~avro~SchemaRegistry~http://cnflnt4-04:8081
!
!
kafka~avro~namespace~RegTest.avro
!
!routine to expand wildcards does not work on unix-style names.
!
kafka~classpath~disk$static:[kafka.CONFLUENT-4-1-1.kafkaserde-tools]*.jar
kafka~classpath~disk$static:[kafka.CONFLUENT-4-1-1.confluent-common]*.jar
kafka~classpath~disk$static:[kafka.CONFLUENT-4-1-1.kafka]*.jar
!
kafka~Topic~'RegTest.',Table_Name
!kafka~Model~ExactlyOnce | Transaction | Flush?
!kafka~Model~Transaction
kafka~Model~ExactlyOnce
!
!
! External Properties File
!
java~properties~/control_files/kafka_avro.properties
```

Define the target. Use the output keyword to define the target (output type) and other characteristics. This is required.

```
output~kafka~synch~connect~record~Avro
```

- Output is to a Kafka server in Avro format

Define connect. Connect defines the Kafka broker list. Defining connect is optional, unless schema registry is defined.

```
kafka~connect~cnflnt4-04:9092,cnflnt4-03:9092,cnflnt4-02:9092
```

- In this example, the target is a list of Karka brokers in a Kafka Cluster. If one Kafka broker fails, the JCC LogMiner Loader will fail over and connect to the next broker
- The broker addresses and port numbers must be provided by whomever is configuring and managing your Kafka server.
- This is a Kafka specific keyword not discussed in the full documentation.
- This is required for schema registry.

Define the schema registry. This optional parameter provides the URL for Confluent Schema Registry.

```
kafka~avro~SchemaRegistry~http://cnflnt4-04:8081
```

- The schema registry address and port numbers must be provided by whoever is configuring and managing your Kafka server.
- This is a Kafka specific keyword not discussed in the full documentation.

Define the namespace. This provides a unique name to use in the schema registry. It is optional, unless the schema registry is defined.

```
kafka~avro~namespace~RegTest.avro
```

- The namespace is the unique name within the schema registry under which all the table definitions reside. The value specified is a field in each of the Avro table definition JSON documents.
- This is a Kafka specific keyword not discussed in the full documentation.
- This is required for schema registry.

Define the classpath. This is required and may require multiple statements.

```
kafka~classpath~<directory>*.jar
```

- One classpath entry for each directory that needs to be added to the Java classpath

- OpenVMS Directory syntax
- For Confluent Kafka, three entries are required:
  disk$static:[kafka.CONFLUENT-4-1-1.kafka-serde-tools]*.jar
  disk$static:[kafka.CONFLUENT-4-1-1.confluent-common]*.jar
  disk$static:[kafka.CONFLUENT-4-1-1.kafka]*.jar
- Disk and directory are site specific

Define Topics. Defining topics in the Control File is optional. The Topic naming supplied here must be coordinated with the Kafka application. See also “Keyword: Kafka~topic” on page 24.

```
kafka~Topic~'RegTest.',Table_Name
```

- ‘RegTest.’ -- literal prefix that will be the first part of the Kafka Topic
- Table_name -- Keyword, will be replaced by the output table name
- The full syntax is documented in “Keyword: Kafka~topic” on page 24.

Define the Kafka Model. Defining the Kafka model is optional. See also “Keyword: Kafka~model” on page 25 and “Kafka Model” on page 39.

```
kafka~Model~ExactlyOnce
```

- The default is transaction which defines that the Kafka server guarantees that a published message will be available to a consumer at least once.
- The example sets the model to exactly_once which defines that the Kafka server guarantees that a published message will be available to a consumer exactly once.

Point to the Properties File. Naming the properties file is optional. See also “Java Properties File” on page 35.

```
java~properties~/control_files/kafka_avro.properties
```

- Control_files is a logical name pointing to the directory that contains the properties file.
- Properties file is a text file that can contain a variety of Java properties.

Avro Message Example

Once an Avro message has been uncompressed (deserialized), it can be viewed as a JSON document. The following example is such a JSON message formatted for readability.

```
{
  "ORIGINATING_DBKEY": 29273399897423884,
  "DETAILS_ID": 875713,
  "PROGNAME": "DETAILS                        ",
  "PROGINDEX": "004",
  "SEQNUM": 1,
  "AMOUNT_F": 5846.9395,
  "AMOUNT_G": 5846.939453125,
  "AMOUNT_N": {
    "bytes": "\bë?
  },
  "DATE_VMS": -2529777414675889,
  "DATE_ANSI": -40587,
  "TIMESTAMP_A": 2213681432640000,
  "TIMESTAMP_A_CHAR": "2040-02-24 07:30:32.64",
  "JCCLML_ACTION": "M",
  "LOADER_SEQUENCE_NUMBER": 57,
  "LOADERNAME": "KAFKA_AVRO",
  "LOADER_VERSION": "T03.06.00",
  "LOADER_LINK_DATE_TIME": 1544449674300000,
  "TRANSACTION_COMMIT_TIME": 1544600905798528,
  "TRANSACTION_START_TIME": 1544600905797528,
  "JCCLML_READ_TIME": 1544601031372273,
  "JCCLML_AERCP": "1-28-4761-1202-206182924-206182924",
  "TRANSMISSION_DATE_TIME": 1544601034003222
}
```

CHAPTER 6

Kafka Command Examples

The following examples use the Java Kafka commands on OpenVMS and are useful for testing and understanding the Kafka interface.

Setting the Environment for the Examples

Note that Kafka requires Java V8 and that using Java commands requires that the process be set to parse_style=extended. With parse_style=extended, DCL does not uppercase all command line input. All of the Kafka commands in the following sections are case sensitive.

The following example sets the JCC LogMiner Loader version, the Java version, and the DCL parse_style:

```
$  @jcc_tool_com:jcc_lml_user 3.6
Setting JCC LogMiner Loader version 3.6
$ jcc_lml_jdbc_user 8.0 -
_$ SYS$COMMON:[java$80.com]java$setup.com
JCC LogMiner Loader Java version 8.0
$ java -version
java version "1.8.0.11-OpenVMS"
Java(TM) SE Runtime Environment (build 1.8.0.11-vms-b1)
```

```
Java HotSpot(TM) 64-Bit Server VM (build 25.11-b1, mixed mode)
$ jcc_version
JCC Version T03.06.00 (built  9-NOV-2018 16:41:23.39)

$ set process/parse_style=extended
$
```

These examples also use the logical name confluent_kafka_path that has been defined in the JCC Local Environment Command procedure.

Start a Consumer Console

The following example sets up the Kafka ConsoleConsumer to consume messages from the topic Personnel.EMPLOYEES. The SSL parameters are specified in the properties.config file KAFKA_AVRO.PROPERTIES.

```
$ java -cp /confluent_kafka_path/*:. kafka.tools.ConsoleConsumer -
_$  --bootstrap-server confluent01.jcc.com:9093 -
_$  --topic Personnel.EMPLOYEES -
_$  --formatter io.confluent.kafka.formatter.AvroMessageFormatter -
_$  --property schema.registry.url="http://confluent01.jcc.com:8081" -
_$  --consumer.config KAFKA_AVRO.PROPERTIES
{"EMPLOYEE_ID":"00164","LAST_NAME":{"string":"Toliver
"},"FIRST_NAME":{"string":"Alvin     "},"MIDDLE_INI-
TIAL":{"string":"A"},"ADDRESS_DATA_1":{"string":"146 Parnell Place
"},"ADDRESS_DATA_2":{"string":"                    "},"CITY":{"string":"Chocoru
a           "},"STATE":{"string":"NH"},"POST-
AL_CODE":{"string":"03817"},"SEX":{"string":"M"},"BIRTHDAY":{"long":-
718416000000000},"STATUS_CODE":{"string":"2"},"TRANSACTION_COMMIT_TIME":{"long"
:1542632686434735},"JCCLML_READ_TIME":{"long":1542632686518735},"TRANSMISSION_D
ATE_TIME":{"long":1542632774582159},"LOADER_SEQUENCE_NUMBER":{"long":12}}
{"EMPLOYEE_ID":"00165","LAST_NAME":{"string":"Smith
"},"FIRST_NAME":null,"MIDDLE_INITIAL":{"string":"D"},"ADDRESS_-
DATA_1":{"string":"120 Tenby Dr.          "},"ADDRESS_-
DATA_2":{"string":"                   "},"CITY":{"string":"Chocorua
"},"STATE":{"string":"NH"},"POST-
AL_CODE":{"string":"03817"},"SEX":{"string":"M"},"BIRTHDAY":{"long":-
493344000000000},"STATUS_CODE":{"string":"2"},"TRANSACTION_COMMIT_TIME":{"long"
:1542632693562609},"JCCLML_READ_TIME":{"long":1542632782489033},"TRANSMISSION_D
ATE_TIME":{"long":1542632782491033},"LOADER_SEQUENCE_NUMBER":{"long":13}}
```

Note that the output format is a single line per message and is not formatted for readability.

Start a Producer Console

An interactive Kafka producer console is useful for testing configuration files and connectivity. The following example creates a Kafka producer console session for the topic Test.Topic.

```
$ set proc/parse_style=extended
$ java -cp /confluent_kafka_path/*:. -
_$    kafka.tools.ConsoleProducer -
_$    --broker-list  confluent01:9093 -
_$    --topic Test.Topic -
_$    --producer.config KAFKA_AVRO.PROPERTIES
>This is a test message to Test.Topic
>This is a second test message.
>
```

This publishes arbitrary text messages to the topic Test.Topic to the Karka server on the node Confluent01, port 9093.

In another terminal session, a Kafka consumer console can be set up to consume the messages from the Kafka server:

```
$ set proc/parse_style=extended
$ java -version
$ java -cp /confluent_kafka_path/*:. -
_$    kafka.tools.ConsoleConsumer -
_$    --bootstrap-server confluent01:9093 -
_$    --topic Test.Topic -
_$    --consumer.config KAFKA_AVRO.PROPERTIES
This is a test message to Test.Topic
This is a second test message.
```

Setting up a Producer and Consumer is useful for testing connectivity to the Kafka environment. Because of the complexity of manually formating a JSON or XML record, a Kafka Producer Console is not useful for production.

List Topics

List the topics on a Kafka server with JCC_Kafka_Topics.

```
$ set process/parse_style=extended
$ java -cp /confluent_kafka_path/*:. kafka.admin.TopicCommand -
_$    --zookeeper CNFLNT4-04:2181 --list
ALL_DATATYPES_TABLE
RegTest.DETAILS
RegTest.DETAILS_AUDIT
RegTest.PEOPLE
__confluent.support.metrics
__consumer_offsets
__transaction_state
_schemas
avro.ALL_DATATYPES_TABLE
$
```

See Details of a Topic

Describe the details of a particular topic

```
$ java -cp /confluent_kafka_path/*:. kafka.admin.TopicCommand -
_$    --zookeeper CNFLNT4-04:2181 --topic RegTest.DETAILS --describe
Topic:RegTest.DETAILS   PartitionCount:5   ReplicationFactor:2     Configs:
   Topic: RegTest.DETAILS Partition: 0   Leader: 2    Replicas: 2,3   Isr: 3,2
   Topic: RegTest.DETAILS Partition: 1   Leader: 3    Replicas: 3,1   Isr: 3,1
   Topic: RegTest.DETAILS Partition: 2   Leader: 1    Replicas: 1,2   Isr: 2,1
   Topic: RegTest.DETAILS Partition: 3   Leader: 2    Replicas: 2,1   Isr: 2,1
   Topic: RegTest.DETAILS Partition: 4   Leader: 3    Replicas: 3,2   Isr: 3,2
```

In this example, the topic RetTest.DETAILS is split across five partitions. Each partition is replicated on two nodes in a three node Kafka cluster.

Change the Partitions for a Topic

The following example increases the number of partitions for the topic RegTest.DETAILS from 5 to 6

```
$ java -cp /confluent_kafka_path/*:. kafka.admin.TopicCommand -
_$    --zookeeper CNFLNT4-04:2181 --topic "RegTest.DETAILS" -
_$    --alter --partitions 6
WARNING: If partitions are increased for a topic that has a key, the partition
logic or ordering of the messages will be affected
Adding partitions succeeded!
$ java -cp /confluent_kafka_path/*:. kafka.admin.TopicCommand
_$    --zookeeper CNFLNT4-04:2181 --topic RegTest.DETAILS --describe
Topic:RegTest.DETAILS   PartitionCount:6        ReplicationFactor:2   Configs:
   Topic: RegTest.DETAILS Partition: 0   Leader: 2    Replicas: 2,3   Isr: 3,2
   Topic: RegTest.DETAILS Partition: 1   Leader: 3    Replicas: 3,1   Isr: 3,1
   Topic: RegTest.DETAILS Partition: 2   Leader: 1    Replicas: 1,2   Isr: 2,1
   Topic: RegTest.DETAILS Partition: 3   Leader: 2    Replicas: 2,1   Isr: 2,1
   Topic: RegTest.DETAILS Partition: 4   Leader: 3    Replicas: 3,2   Isr: 3,2
   Topic: RegTest.DETAILS Partition: 5   Leader: 1    Replicas: 1,3   Isr: 1,3
```

Note that the optimal number of partitions and replicas depends on the Kafka configuration and how it is being used. Choosing the correct configuration is beyond the scope of this manual.

Create a New Topic

The following example creates a new topic called Personnel.EMPLOYEES

```
$ java -cp /confluent_kafka_path/*:. kafka.admin.TopicCommand -
_$    --zookeeper CNFLNT4-04:2181 --create --topic Personnel.EMPLOYEES -
_$    --partitions 6 --replication-factor 2
WARNING: Due to limitations in metric names, topics with a period ('.') or
underscore ('_') could collide. To avoid issues it is best to use either, but
not both.
Created topic "Personnel.EMPLOYEES".

The topic Personnel.EMPLOYEES now shows up in the list of topics

$ java -cp /confluent_kafka_path/*:. kafka.admin.TopicCommand --zookeeper CNFL-
NT4-04:2181 --list
ALL_DATATYPES_TABLE
Personnel.EMPLOYEES
RegTest.DETAILS
RegTest.DETAILS_AUDIT
RegTest.PEOPLE
__confluent.support.metrics
__consumer_offsets
__transaction_state
_schemas
avro.ALL_DATATYPES_TABLE
```

The new topic Personnel.EMPLOYEES has six partitions with two replicas for each partition.

```
$ java -cp /confluent_kafka_path/*:. kafka.admin.TopicCommand -
_$    --zookeeper CNFLNT4-04:2181 --topic Personnel.EMPLOYEES --describe
Topic:Personnel.EMPLOYEES   PartitionCount:6    ReplicationFactor:2   Configs:
   Topic: Personnel.EMPLOYEES Partition: 0  Leader: 3  Replicas: 3,1   Isr: 3,1
   Topic: Personnel.EMPLOYEES Partition: 1  Leader: 1  Replicas: 1,2   Isr: 1,2
   Topic: Personnel.EMPLOYEES Partition: 2  Leader: 2  Replicas: 2,3   Isr: 2,3
   Topic: Personnel.EMPLOYEES Partition: 3  Leader: 3  Replicas: 3,2   Isr: 3,2
   Topic: Personnel.EMPLOYEES Partition: 4  Leader: 1  Replicas: 1,3   Isr: 1,3
   Topic: Personnel.EMPLOYEES Partition: 5  Leader: 2  Replicas: 2,1   Isr: 2,1
$
```

It is possible to configure a Kafka server so that topics are created automatically with the first message published to a topic or to restrict topic creation so that a topic must be explicitly created. The default number of partitions and replicas can be configured in the Kafka server, but the create topic command requires that the partitions and replication-factor be specified.

Check the status of Consumers

Kafka Consumers can use an explicit group . If multiple consumers specify the same group, the consumers can run in parallel. Kafka tracks the offset for each consumer so that consumers within a group do not get duplicate copies of messages.

The kafka.admin.ConsumerGroupCommand can be used to retrieve information about the current status of consumers. In the following example, the Kafka consumers are using the group “GroupAvro”. The interesting column in this example is LAG. This is the number of messages that have been published to the Kafka server but have not yet been consumed.

```
$ java -cp /confluent_kafka_path/*:. kafka.admin.ConsumerGroupCommand -
_$    --bootstrap-server cnflnt4-02.jcc.com:9093 -
_$    --group  GroupAvro -
_$    --describe -
_$    --command-config KAFKA_AVRO.PROPERTIES
Note: This will not show information about old Zookeeper-based consumers.

TOPIC                 PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG
RegTest.DETAILS       0          15167958        15170338        2380
RegTest.DETAILS       1          15161694        15164178        2484
RegTest.DETAILS       2          15169594        15171990        2396
RegTest.DETAILS       3          15171111        15173487        2376
RegTest.DETAILS       4          15162272        15164735        2463
RegTest.DETAILS       5          6488404         6490773         2369
RegTest.DETAILS_AUDIT 0          15132437        15132437        0
RegTest.DETAILS_AUDIT 1          15124631        15124631        0
RegTest.DETAILS_AUDIT 2          15129652        15129652        0
RegTest.DETAILS_AUDIT 3          15135289        15135289        0
RegTest.DETAILS_AUDIT 4          15126100        15126100        0
RegTest.DETAILS_AUDIT 5          6602087         6602087         0
RegTest.PEOPLE        0          10963940        10964178        238
RegTest.PEOPLE        1          10957708        10958010        302
RegTest.PEOPLE        2          10961103        10961103        0
RegTest.PEOPLE        3          10971694        10971694        0
RegTest.PEOPLE        4          10973754        10974100        346
RegTest.PEOPLE        5          5131130         5132037         907
$
```

For readability of the example, three columns have been truncated from the right side of each line, CONSUMER-ID, HOST, and CLIENT-ID.