Using the Spark CDM Connector

Initial limited preview release

5/3/2020

Overview

The Spark CDM connector enables a Spark program to read and write CDM entities in a CDM folder via dataframes. In principle, the Spark CDM connector will work in any Spark environment, however this initial limited preview release has only been tested with and is only supported with Azure Databricks (and shortly with the Spark environment in Azure Synapse). **During this limited preview, use of the Spark/CDM connector in production applications is not recommended or supported.** The connector capabilities and API may be changed in subsequent releases without notice.

Note: See CDM documentation for help in defining CDM documents using CDM 1.0.
https://docs.microsoft.com/en-us/common-data-model/

Installing the Spark CDM connector

The Spark CDM connector library is provided as a jar file in GitHub and Maven that must be installed in the Azure Databricks Spark environment. Sample code and CDM models and pre-release documentation are provided in GitHub.

https://mvnrepository.com/artifact/com.microsoft.azure/spark-cdm-connector

https://github.com/Azure/spark-cdm-connector

Important: Verify you are using version 0.8 of the jar or later.

Supported scenarios

The following scenarios are supported:

- Reading data from an entity in a CDM folder into a Spark dataframe.
- Writing from a Spark dataframe to an entity in a CDM folder based on a CDM entity definition.
- Writing from a Spark dataframe to an entity in a CDM folder based on the dataframe schema.

Capabilities/limitations

The following apply to the initial private preview release.

- Supports CDM folders in ADLS gen2 only.
- Supports reading CDM metadata in both manifest and model.json files.
- Supports writing CDM to a manifest file. Writing to a model.json file is not planned.
- Supports data in csv and Parquet format. Support for nested Parquet files is planned.
- Supports sub-manifests and partition patterns when reading and writing from manifest files.

See also the *Known issues* section below.

Unsupported scenarios

The following scenarios are not yet supported:

- Programmatic access to entity metadata after reading an entity.
- Programmatic access to set or override metadata when writing an entity.

Using the Spark CDM connector to read and write CDM data

The Spark CDM connector is used to modify normal Spark dataframe read and write behavior, with a series of options and modes used as described below.

When reading data, the library uses metadata in the CDM folder to create the dataframe based on the structure of the source entity. Attribute names are used as column names and attribute datatypes are mapped to the column datatype. When the dataframe is loaded it is populated from the partitions identified by the manifest.

When writing to a CDM folder, if the entity does not already exist a new entity definition is created and added to the CDM folder and referenced in the manifest. Two entity definition modes are supported:

- **Explicit**: a reference to an entity definition is provided (can be an unresolved logical entity definition or a resolved physical entity definition).
    - If the dataframe structure does not match the referenced entity definition, an error is returned.
    - If the dataframe is valid,
        - If the entity already exists in the manifest, the provided entity definition is resolved and validated against the definition in the CDM folder. If the definitions do not match an error is returned, otherwise data is written.
        - If the entity does not exist in the CDM folder, a resolved copy of the entity definition is written to the CDM folder and data is written.
- **Implicit**: the entity definition is derived from the dataframe structure.
    - If the entity does not exist in the CDM folder the implicit definition is used to create the resolved entity definition in the target CDM folder.
    - If the entity exists in CDM folder, the implicit definition is validated against the existing entity definition. If the definitions do not match an error is returned, otherwise data is written.
    - In addition, a derived logical entity definition is written into a subfolder of the entity folder

Data is written to data subfolder(s) within an entity subfolder subject to a save mode. The save mode determines whether the new data overwrites or is appended to existing data, or if an error is returned if data exists. The default is to return an error if data already exists.

Parameters, options and mode

For both read and write, the connector library name is provided as a parameter. A series of options are required which parameterize the behavior of the Spark CDM connector. For write, a save mode is also supported.

The connector library name, options and save mode are formatted as follows:

<dataframe>.read.format("com.microsoft.cdm") [.option("<option>", "<value>")]*

<dataframe>.write.format("com.microsoft.cdm") [.option("<option>", "<value>")]* .mode(<mode>)

Credential options

Credentials must be provided for the Spark CDM connector to access data. In Azure Active Directory, create an App Registration and then grant this App Registration access to your storage account using either of the following roles **Storage Blob Data Contributor** to allow the library to write to CDM folders, or **Storage Blob Data Reader** to allow only read. Once permissions are created, you can pass the app id,

app key, and tenant id to the connector. It is recommended to use Azure Key Vault to secure these values, to ensure they are not written in clear text in a notebook file.

In Azure Databricks, create a secret scope which can be backed by Azure Key Vault. See: https://docs.microsoft.com/en-us/azure/databricks/security/secrets/secret-scopes#create-an-azure-key-vault-backed-secret-scope

Option	Description	Pattern and example usage
appId	The app registration ID used to authenticate to the storage account	<guid>
appKey	The key or secret.	<encrypted secret>
tenantId	The Azure Active Directory tenant ID under which the application is registered.	<guid>

Common options

The following options identify the entity in the CDM folder that is either being read or written to.

Option	**Description**	**Pattern and example usage**
storage	An ADLS gen2 storage account with HNS enabled in which the CDM folder is located	<accountName>.dfs.core.windows.net "myAccount.dfs.core.windows.net"
container	An ADLS gen2 file system / container in which the source or target CDM folder is located	<containerName> "myContainer"
manifest	The relative path to the manifest or model.json file from the container root. When reading, can be a root manifest or a sub-manifest. For write, must be the root manifest.	[<folderPath>/]<fullManifestName>, "default.manifest.cdm.json" "employees.manifest.cdm.json" "employees/person.manifest.cdm.json" "model.json" (read only)
entity	The name of the source or target entity in the manifest. When writing an entity for the first time in a folder, the resolved entity definition will be given this name.	<entityName> "customer"

Entity definition options

Explicit Write: The following options are used when an explicit entity definition is used to define and validate the entity being written. If the dataframe schema does not match the dataframe schema on write, an error is reported.

Implicit Write: If the entityDefinition options are not supplied when creating an entity in a CDM folder, the entity structure is defined by the schema of the dataframe and basic CDM metadata will be created.

Read: The entityDefinitionContainer (if different from container) and entityDefinitionModelRoot, or useCdmGithubModelRoot are required to allow resolution of the logical entity definition from which the physical entity in the CDM folder being read was resolved.

Option	Description	Pattern / example usage
useCdmGithubModelRoot **Overrides**: entityDefinitionStorage, entityDefinitionContainer, entityDefinitionModelRoot	Indicates the source model root is located at https://github.com/microsoft/CDM/tree/master/schemaDocuments. Explicit write only.	"useCdmGithubModelRoot" Only required when referencing entity types defined in the CDM GitHub.
useCdmGithub	Sets the modelroot alias to https://github.com/microsoft/CDM/tree/master/schemaDocuments to resolve the reference to foundations.cdm.json Required for Implicit write and Read only.	"useCdmGithub"
entityDefinitionStorage [NOT SUPPORTED YET]	The ADLS gen2 storage account containing the entity definition. Required if different to the storage account hosting the CDM folder.	<accountName>.dfs.core.windows.net "myAccount.dfs.core.windows.net"
entityDefinitionContainer	The storage container containing the entity definition. Required if different to the CDM folder container or if entityDefinitionStorage is provided.	<containerName> "models"
entityDefinitionModelRoot	The location of the model root or corpus within the container.	<folderPath> "crm/core"

	If not specified, defaults to the root folder.	("/" can be used to indicate the root folder of the container)
entityDefinition	File path within the model root to the CDM definition file, including the name of the entity in that file.	<folderPath>/<entity>.cdm.json/<entity> "sales/customer.cdm.json/customer"

In the example above, the full path to the customer entity definition object is https://myAccount.dfs.core.windows.net/models/crm/core/sales/customer.cdm.json/entity, where 'models' is the container in ADLS.

IMPORTANT: requires the entityDefinitionContainer (if different from container) and entityDefinitionModelRoot for the logical entity definition from which the physical entity in the CDM folder was resolved to be specified.

Folder structure and data format options on write

The folder organization and file format used on write can be changed with the following options.

Option	Description	Pattern and example usage
useSubManifest	If true, causes the target entity to be included in the root manifest via a sub-manifest. The sub-manifest and the entity definition are written into an entity folder beneath the root.	"true"|"false" False by default.
format	Defines the file format. Current supported file formats are CSV and Parquet	"csv"|"parquet" Default is "csv"
compression	Defines the compression format used with Parquet. Default is "snappy"	"uncompressed"|"snappy"|"gzip"|"lzo" see note below on using lzo with Azure Databricks

Note that the lzo codec is not available by default in Azure Databricks but must be installed. See *https://docs.microsoft.com/en-us/azure/databricks/data/data-sources/read-lzo*

By default, data is written as CSV into a default Data subfolder within the target entity folder using a default partition pattern, <entity>/Data/<entityName>-nnnnn.<ext>.

Options to control the partition pattern, and to provide CSV options are not yet supported.

Save Mode

The save mode specifies how existing entity data in the CDM folder is handled. Options are to overwrite, append to, or error if data already exists. The default save mode is ErrorIfExists.

Mode	Description
SaveMode.Overwrite	Will overwrite existing partitions with data being written. Note: overwrite does not support changing the schema; if the schema of the data being written is incompatible with the existing entity definition an error will be thrown.
SaveMode.Append	Will append data being written as new partitions alongside the existing partitions. Note: append does not support changing the schema; if the schema of the data being written is incompatible with the existing entity definition an error will be thrown.
SaveMode.ErrorIfExists	Will return an error if partitions already exist.

See folder organization for details of how data files are organized.

Examples

The following examples all use the appId, appKey and tenantId values set earlier in the code for an Azure app registration that has separately been given Storage Blob Data Contributor permissions on the storage for write and Storage Blob Data Reader permissions for read.

Implicit Write – using dataframe schema only

This code writes the dataframe df to a CDM folder with a manifest at https://mystorage.dfs.core.windows.net/cdmdata/Contacts/default.manifest.cdm.json with an Event entity.

Event data is written as Parquet files, compressed with gzip, that are appended to the folder (new files are added without deleting existing files).

```
df.write.format("com.microsoft.cdm")
  .option("storage", "mystorage.dfs.core.windows.net")
  .option("container", "cdmdata")
  .option("manifest", "/Contacts/default.manifest.cdm.json")
  .option("entity", "Event")
  .option("format", "parquet")
  .option("compression", "gzip")
  .option("appId", appid)
  .option("appKey", appkey)
  .option("tenantId", tenantid)
  .mode(SaveMode.Append)
  .save()
```

Explicit Write - using entity definition in GitHub

This code writes the dataframe df to a CDM folder with the manifest at https://mystorage.dfs,core.windows.net/cdmdata/Teams/root.manifest.cdm.json and a sub-manifest containing the TeamMembership entity, created in a TeamMembership subdirectory.

TeamMembership data is written as CSV files (by default) that overwrite any existing data files.

The entity definition is retrieved from the CDM GitHub repo, at: https://github.com/microsoft/CDM/tree/master/schemaDocuments/core/applicationCommon/TeamMembership.cdm.json/TeamMembership

```
df.write.format("com.microsoft.cdm")
  .option("storage", "mystorage.dfs.core.windows.net")
  .option("container", "cdmdata")
  .option("manifest", "Teams/root.manifest.cdm.json")
  .option("entity", "TeamMembership")
  .option("entityDefinition", "core/applicationCommon/TeamMembership.cdm.json/TeamMembership")
  .option("useCdmGithubModelRoot", true)
  .option("useSubManifest", true)
  .option("appId", appid)
  .option("appKey", appkey)
  .option("tenantId", tenantid)
  .mode(SaveMode.Overwrite)
  .save()
```

Explicit Write - using entity definition in ADLS

This code writes the dataframe df to a CDM folder with manifest at https://mystorage.dfs.core.windows.net/cdmdata/Contacts/root.manifest.cdm.json with the entity Person.

Person data is written as new CSV files (by default) which overwrite existing files in the folder.

The entity definition is retrieved from https://mystorage.dfs.core.windows.net/models/cdmmodels/core/Contacts/Person.cdm.json/Person

```
df.write.format("com.microsoft.cdm")
  .option("storage", "mystorage.dfs.core.windows.net")
  .option("container", "cdmdata")
  .option("manifest", "/contacts/root.manifest.cdm.json")
  .option("entity", "Person")
  .option("entityDefinitionContainer", "cdmmodels")
  .option("entityDefinitionModelRoot", "core")
  .option("entityDefinition", "/Contacts/Person.cdm.json/Person")
  .option("appId", appid)
  .option("appKey", appkey)
  .option("tenantId", tenantid)
  .mode(SaveMode.Overwrite)
  .save()
```

Read

This code reads the Person entity from the CDM folder with manifest at https://mystorage.dfs.core.windows.net/cdmdata/contacts/root.manifest.cdm.json

```
val df = spark.read.format("com.microsoft.cdm")
  .option("storage", "mystorage.dfs.core.windows.net")
```

```
  .option("container", "cdmdata")
  .option("manifest", "/contacts/root.manifest.cdm.json")
  .option("entity", "Person")
  .option("useCDMGithub", true)
  .option("appId", appid)
  .option("appKey", appkey)
  .option("tenantId", tenantid)
  .load()
```

Other Considerations

Handling Date and Time Formats

Date and Time datatype values are handled as normal for Spark and Parquet, and in CSV are read/written in ISO 8601 format.

In CDM, DateTime datatype values are interpreted as UTC, and in CSV written in ISO 8601 format, e.g. 2020-03-13 09:49:00Z.

DateTimeOffset values intended for recording local time instants are handled differently in Spark and Parquet from CSV. While CSV and other formats can express a local time instant as a structure, comprising a datetime and a UTC offset, formatted in CSV like, 2020-03-13 09:49:00-08:00, Parquet and Spark don't support such structures. Instead, they use a TIMESTAMP datatype that allows an instant to be recorded in UTC time (or in some unspecified time zone).

The Spark CDM connector will convert a DateTimeOffset value in CSV to a **UTC timestamp**. This will be persisted as a Timestamp in Parquet and if subsequently persisted to CSV, the value will be serialized as a DateTimeOffset with a +00:00 offset. Importantly, there is no loss of temporal accuracy – the serialized values represent the same instant as the original values, although the offset is lost. Spark systems use their system time as the baseline and normally express time using that local time. UTC times can always be computed by applying the local system offset. Note that for Azure systems in all regions, system time is always UTC, so **all timestamp values will normally be in UTC**.

As Azure system values are always UTC, when using implicit write, where a CDM definition is derived from a dataframe, timestamp columns translated to attributes a CDM DateTime datatype which implies a UTC time.

If it is important to persist a local time and the data will be processed in Spark or persisted in Parquet, then it is recommended to use a DateTime attribute and keep the offset in a separate attribute, for example as a signed integer value representing minutes. In CDM, DateTime values are UTC, so the offset must be applied when needed to compute local time.

In most cases, persisting local time is not important. Local times are often only required in a UI for user convenience and based on the user's time zone, so not storing a UTC time is often a better solution.

Folder organization

When writing CDM folders, the following folder organization is used. Data files are written into a single data folder. Options for organizing data files into subfolders will be supported in a later release.

Explicit (existing entity definition is referenced)

+-- <CDMFolder>

|-- default.manifest.cdm.json *<< with entity ref and partition info*

|-- <Entity>

|-- <entity>.cdm.json << *resolved physical entity definition*

+-- Data

+-- data files…

With sub-manifest:

+-- <CDMFolder>

|-- default.manifest.cdm.json *<< contains reference to sub-manifest*

+-- <Entity>

|-- <entity>.cdm.json

+-- <entity>.manifest.cdm.json << sub-manifest with entity ref and partition info

+-- Data

+-- data files…

Implicit (entity definition is derived from dataframe schema)

+-- <CDMFolder>

|-- default.manifest.cdm.json

+-- <Entity>

|-- <Entity>.cdm.json *<< resolved physical entity definition*

+-- Data

| +-- data files…

+-- LogicalDefinition

+-- <entity>.cdm.json *<< logical entity definition(s)*

With sub-manifest:

+-- <CDMFolder>

|-- default.manifest.cdm.json *<< contains reference to sub-manifest*

+-- <Entity>

|-- <entity>.cdm.json *<< resolved physical entity definition*

|-- <entity>.manifest.cdm.json *<< sub-manifest with entity ref and partition info*

+-- Data

| +-- data files…

+-- LogicalDefinition

+-- <entity>.cdm.json *<< logical entity definition(s)*

Known issues

- When writing Parquet, in Azure Databricks, there need to be fewer executors than rows in the dataframe. The number of executors is normally determined by the cluster configuration. With any significant volumes of data this is not a problem. If testing with only a few rows you can simply reduce the number of executors. The following code snippet creates a dataframe bound to just two executors.
  ```
  spark.createDataFrame(spark.sparkContext.parallelize(data, 2), schema)
  ```
- When using Parquet in Azure Databricks, lzo compression is not currently supported.
- When using implicit write, the implied entity definition is written into a subfolder and has attributes declared with a CDM data format rather than a CDM data type. These declarations will be changed in a later release to use CDM primitive data types as normally used in a CDM logical entity definition.

Not yet supported

The following features are not yet supported:

- Use of headers when reading or writing CSV files and using a separator character other than a comma.
- Use and configuration of partition patterns for organizing data files.
- Nested Parquet support.
- In explicit write, use of a distinct ADLS storage account for the entity definition. Initially, the entity definition must be in the same storage account as the target CDM folder.

Sample Code

See GitHub for an initial sample notebook and CDM files.

Changes

5/3, qualified SaveMode descriptions to clarify that Overwrite and Append do not allow schema change