Oracle® Big Data Discovery

Data Processing Guide

Version 1.4.0 • October 2016

ORACLE®

Copyright and disclaimer

Table of Contents

Preface

Oracle Big Data Discovery is a set of end-to-end visual analytic capabilities that leverage the power of Apache Spark to turn raw data into business insight in minutes, without the need to learn specialist big data tools or rely only on highly skilled resources. The visual user interface empowers business analysts to find, explore, transform, blend and analyze big data, and then easily share results.

About this guide

This guide describes the Data Processing component of Big Data Discovery (BDD). It explains how the product behaves in Spark when it runs its processes, such as sampling, loading, updating and transforming data. It also describes Spark configuration, the Data Processing CLI for loading and updating data sets (via cron jobs and on demand), and the behavior of Data Enrichment Modules, such as GeoTagger and Sentiment Analysis. Lastly, it includes logging information for the Data Processing component in BDD, the Transform Service, and the Dgraph HDFS Agent.

Audience

This guide is intended for Hadoop IT administrators, Hadoop data developers, and ETL data engineers and data architects who are responsible for loading source data into Big Data Discovery.

The guide assumes that you are familiar with the Spark and Hadoop environment and services, and that you have already installed Big Data Discovery and used Studio for basic data exploration and analysis.

This guide is specifically targeted for Hadoop developers and administrators who want to know more about data processing steps in Big Data Discovery, and to understand what changes take place when these processes run in Spark.

The guide covers all aspects of data processing, from initial data discovery, sampling and data enrichments, to data transformations that can be launched at later stages of data analysis in BDD.

Conventions

The following conventions are used in this document.

Typographic conventions

The following table describes the typographic conventions used in this document.

Typeface	Meaning
User Interface Elements	This formatting is used for graphical user interface elements such as pages, dialog boxes, buttons, and fields.
`Code Sample`	This formatting is used for sample code segments within a paragraph.

Typeface	Meaning
Variable	This formatting is used for variable values. For variables within a code sample, the formatting is `Variable`.
`File Path`	This formatting is used for file names and paths.

Path variable conventions

This table describes the path variable conventions used in this document.

Path variable	Meaning
`$ORACLE_HOME`	Indicates the absolute path to your Oracle Middleware home directory, where BDD and WebLogic Server are installed.
`$BDD_HOME`	Indicates the absolute path to your Oracle Big Data Discovery home directory, `$ORACLE_HOME/BDD-<version>`.
`$DOMAIN_HOME`	Indicates the absolute path to your WebLogic domain home directory. For example, if your domain is named `bdd-<version>_domain`, then `$DOMAIN_HOME` is `$ORACLE_HOME/user_projects/domains/bdd-<version>_domain`.
`$DGRAPH_HOME`	Indicates the absolute path to your Dgraph home directory, `$BDD_HOME/dgraph`.

Contacting Oracle Customer Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. This includes important information regarding Oracle software, implementation questions, product and solution help, as well as overall news and updates from Oracle.

You can contact Oracle Customer Support through Oracle's Support portal, My Oracle Support at *https://support.oracle.com*.

Chapter 1
Introduction

This section provides a high-level introduction to the Data Processing component of Big Data Discovery.

BDD integration with Spark and Hadoop

Secure Hadoop options

Preparing your data for ingest

BDD integration with Spark and Hadoop

Hadoop provides a number of components and tools that BDD requires to process and manage data. The Hadoop Distributed File System (HDFS) stores your source data and Hadoop Spark on YARN runs all Data Processing jobs. This topic discusses how BDD fits into the Spark and Hadoop environment.

Hadoop is a platform for distributed storing, accessing, and analyzing all kinds of data: structured, unstructured, and data from the Internet Of Things. It is broadly adopted by IT organizations, especially those that have high volumes of data.

As a data scientist, you often must practice two kinds of analytics work:

- In operational analytics, you may work on model fitting and its analysis. For this, you may write code for machine-learning models, and issue queries to these models at scale, with real-time incoming updates to the data. Such work involves relying on the Hadoop ecosystem. Big Data Discovery allows you to work without leaving the Spark environment that the rest of your work takes place in. BDD supports an enterprise-quality business intelligence experience directly on Hadoop data, with high numbers of concurrent requests and low latency of returned results.

- In investigative analytics, you may use interactive statistical environments, such as R to answer ad-hoc, exploratory questions and gain insights. BDD also lets you export your data from BDD back into Hadoop, for further investigative analysis with other tools within your Hadoop deployment.

By coupling tightly with Spark and Hadoop, Oracle Big Data Discovery achieves data discovery for any data, at significantly-large scale, with high query-processing performance.

About Hadoop distributions

Big Data Discovery works with very large amounts of data stored within HDFS. A Hadoop distribution is a prerequisite for the product, and it is critical for the functionality provided by the product.

BDD uses the HDFS, Hive, Spark, and YARN components packaged with a specific Hadoop distribution. For detailed information on Hadoop version support and packages, see the *Installation Guide*.

BDD inside the Hadoop Infrastructure

Big Data Discovery brings itself to the data that is natively available in Hadoop.

BDD maintains a list of all of a company's data sources found in Hive and registered in HCatalog. When new data arrives, BDD lists it in Studio's **Catalog**, decorates it with profiling and enrichment metadata, and, when you take this data for further exploration, takes a sample of it. It also lets you explore the source data further by providing an automatically-generated list of powerful visualizations that illustrate the most interesting characteristics of this data. This helps you cut down on time spent for identifying useful source data sets, and on data set preparation time; it increases the amount of time your team spends on analytics leading to insights and new ideas.

BDD is embedded into your data infrastructure, as part of Hadoop ecosystem. This provides operational simplicity:

- Nodes in the BDD cluster deployment can share hardware infrastructure with the existing Hadoop cluster at your site. Note that the existing Hadoop cluster at your site may still be larger than a subset of Hadoop nodes on which data-processing-centric components of BDD are deployed.

- Automatic indexing, data profiling, and enrichments take place when your source Hive tables are discovered by BDD. This eliminates the need for a traditional approach of cleaning and loading data into the system, prior to analyzing it.

- BDD performs distributed query evaluation at a high scale, letting you interact with data while analyzing it.

  A Studio component of BDD also takes advantage of being part of Hadoop ecosystem:

  - It brings you insights without having to work for them — this is achieved by data discovery, sampling, profiling, and enrichments.

  - It lets you create links between data sets.

  - It utilizes its access to Hadoop as an additional processing engine for data analysis.

Benefits of integration of BDD with Hadoop and Spark ecosystem

Big Data Discovery is deployed directly on a subset of nodes in the pre-existing Hadoop cluster where you store the data you want to explore, prepare, and analyze.

By analyzing the data in the Hadoop cluster itself, BDD eliminates the cost of moving data around an enterprise's systems — a cost that becomes prohibitive when enterprises begin dealing with hundreds of terabytes of data. Furthermore, a tight integration of BDD with HDFS allows profiling, enriching, and indexing data as soon as the data enters the Hadoop cluster in the original file format. By the time you want to see a data set, BDD has already prepared it for exploration and analysis. BDD leverages the resource management capabilities in Spark to let you run mixed-workload clusters that provide optimal performance and value.

Finally, direct integration of BDD with the Hadoop ecosystem streamlines the transition between the data preparation done in BDD and the advanced data analysis done in tools such as Oracle R Advanced Analytics for Hadoop (ORAAH), or other 3rd party tools. BDD lets you export a cleaned, sampled data set as a Hive table, making it immediately available for users to analyze in ORAAH. BDD can also export data as a file and register it in Hadoop, so that it is ready for future custom analysis.

Secure Hadoop options

This section describes how BDD workflows can be used in a secure Hadoop environment.

Additional information on BDD security is provided in the *Security Guide.*

Kerberos authentication

TLS/SSL and Encryption options

Kerberos authentication

Data Processing components can be configured to run in a Hadoop cluster that has enabled Kerberos authentication.

The Kerberos Network Authentication Service version 5, defined in RFC 1510, provides a means of verifying the identities of principals in a Hadoop environment. Hadoop uses Kerberos to create secure communications among its various components and clients. Kerberos is an authentication mechanism, in which users and services that users want to access rely on the Kerberos server to authenticate each to the other. The Kerberos server is called the Key Distribution Center (KDC). At a high level, it has three parts:

- A database of the users and services (known as principals) and their respective Kerberos passwords
- An authentication server (AS) which performs the initial authentication and issues a Ticket Granting Ticket (TGT)
- A Ticket Granting Server (TGS) that issues subsequent service tickets based on the initial TGT

The principal gets service tickets from the TGS. Service tickets are what allow a principal to access various Hadoop services.

To ensure that Data Processing workflows can run on a secure Hadoop cluster, these BDD components are enabled for Kerberos support:

- Dgraph and Dgraph HDFS Agent
- Data Processing workflows (whether initiated by Studio or the DP CLI)
- Studio

All these BDD components share one principal and keytab. Note that there is no authorization support (that is, these components do not verify permissions for users).

The BDD components are enabled for Kerberos support at installation time, via the `ENABLE_KERBEROS` parameter in the `bdd.conf` file. The `bdd.conf` file also has parameters for specifying the name of the Kerberos principal, as well as paths to the Kerberos keytab file and the Kerberos configuration file. For details on these parameters, see the *Installation Guide*.

Note: If you use Sentry for authorization in your Hadoop cluster, you must configure it to grant BDD access to your Hive tables.

Kerberos support in DP workflows

Support for Kerberos authentication ensures that Data Processing workflows can run on a secure Hadoop cluster. The support for Kerberos includes the DP CLI, via the Kerberos properties in the `edp-cli.properties` configuration file.

The `spark-submit` script in Spark's `bin` directory is used to launch DP applications on a cluster, as follows:

1. Before the call to `spark-submit`, Data Processing logs in using the local keytab. The `spark-submit` process grabs the Data Processing credentials during job submission to authenticate with YARN and Spark.

2. Spark gets the HDFS delegation tokens for the name nodes listed in the `spark.yarn.access.namenodes` property and this enables the Data Processing workflow to access HDFS.

3. When the workflow starts, the Data Processing workflow logs in using the Hadoop cluster keytab.

4. When the Data Processing Hive Client is initialized, a SASL client is used along with the Kerberos credentials on the node to authenticate with the Hive Metastore. Once authenticated, the Data Processing Hive Client can communicate with the Hive Metastore.

When a Hive JDBC connection is used, the credentials are used to authenticate with Hive, and thus be able to use the service.

Kerberos support in Dgraph and Dgraph HDFS Agent

In BDD, the Dgraph HDFS Agent is a client for Hadoop HDFS because it reads and writes HDFS files from and to HDFS. If your Dgraph databases are stored on HDFS, you must also enable Kerberos for the Dgraph.

For Kerberos support for the Dgraph, make sure these `bdd.conf` properties are set correctly:

- `KERBEROS_TICKET_REFRESH_INTERVAL` specifies the interval (in minutes) at which the Dgraph's Kerberos ticket is refreshed.

- `KERBEROS_TICKET_LIFETIME` sets the amount of time that the Dgraph's Kerberos ticket is valid.

See the *Administrator's Guide* for instructions on setting up the Dgraph for Kerberos support.

For Kerberos support, the Dgraph HDFS Agent will be started with three Kerberos flags:

- The `--principal` flag specifies the name of the principal.

- The `--keytab` flag specifies the path to the principal's keytab.

- The `--krb5conf` flag specifies the path to the `krb5.conf` configuration file.

The values for the flag arguments are set by the installation script.

When started, the Dgraph HDFS Agent logs in with the specified principal and keytab. If the login is successful, the Dgraph HDFS Agent passed Kerberos authentication and starts up successfully. Otherwise, HDFS Agent cannot be started.

Kerberos support in Studio

Studio also has support for running the following jobs in a Hadoop Kerberos environment:

- Transforming data sets

- Uploading files

- Export data

The Kerberos login is configured via the following properties in `portal-ext.properties`:

- `kerberos.principal`

- `kerberos.keytab`

- `kerberos.krb5.location`

The values for these properties are inserted during the installation procedure for Big Data Discovery.

TLS/SSL and Encryption options

BDD workflows can run on clusters that are secured with TLS/SSL and HDFS Data at Rest Encryption.

TLS/SSL

TLS/SSL provides encryption and authentication in communication between specific Hadoop services in the secured cluster. When TLS/SSL is enabled, all communication between the services is encrypted, and therefore provides a much higher level of security than a cluster that is not secured with TLS/SSL.

These BDD components can be configured to communicate in a cluster secured with TLS/SSL:

- Studio
- DP CLI
- Dgraph HDFS Agent
- Transform Service

The *Installation Guide* provides details on how to install BDD in a cluster secured with TLS/SSL.

HDFS Data at Rest Encryption

If HDFS Data at Rest Encryption is enabled in your Hadoop cluster, data is stored in encrypted HDFS directories called encryption zones. All files within an encryption zone are transparently encrypted and decrypted on the client side. Decrypted data is therefore never stored in HDFS.

If HDFS Data at Rest Encryption is enabled in your cluster, you must also enable it for BDD. For details, see the *Installation Guide.*

Preparing your data for ingest

Although not required, it is recommended that you clean your source data so that it is in a state that makes Data Processing workflows run smoother and prevents ingest errors.

Data Processing does not have a component that manipulates the source data as it is being ingested. For example, Data Processing cannot remove invalid characters (that are stored in the Hive table) as they are being ingested. Therefore, you should use Hive or third-party tools to clean your source data.

After a data set is created, you can manipulate the contents of the data set by using the Transform functions in Studio.

Removing invalid XML characters

During the ingest procedure that is run by Data Processing, it is possible for a record to contain invalid data, which will be detected by the Dgraph during the ingest operation. Typically, the invalid data will consist of invalid XML characters. A valid character for ingest must be a character according to production 2 of the XML 1.0 specification.

If an invalid XML character is detected, it is replaced with an escaped version. In the escaped version, the invalid character is represented as a decimal number surrounded by two hash characters (##) and a semi-colon (;). For example, a control character whose 32-bit value is decimal 15 would be represented as

```
##15;
```

The record with the replaced character would then be ingested.

Fixing date formats

Ingested date values come from one (or more) Hive table columns:

- Columns configured as `DATE` data types.

- Columns configured as `TIMESTAMP` data types.

- Columns configured as `STRING` data types but having date values. The date formats that are supported via this data type discovery method are listed in the `dateFormats.txt` file. For details on this file, see *Date format configuration on page 36*.

Make sure that dates in `STRING` columns are well-formed and conform to a format in the `dateFormats.txt` file, or else they will be ingested as string values, not as Dgraph `mdex:dateTime` data types.

In addition, make sure that the dates in a `STRING` column are valid dates. For example, the date `Mon, Apr 07, 1925` is invalid because April 7, 1925 is a Tuesday, not a Monday. Therefore, this invalid date would cause the column to be detected as a `STRING` column, not a `DATE` column.

Uploading Excel and CSV files

In Studio, you can create a new data set by uploading data from an Excel or CSV file. The data upload for these file types is always done as `STRING` data types.

For this reason, you should make sure that the file's column data are of consistent data types. For example, if a column is supposed to store integers, check that the column does not have non-integer data. Likewise, check that date input conforms to the formats in the `dateFormats.txt` file.

Note that BDD cannot load multimedia or binary files (other than Excel).

Non-splittable input data handling for Hive tables

Hive tables supports the use of input data that has been compressed using non-splittable compression at the individual file level. However, Oracle discourages using a non-splittable input format for Hive tables that will be processed by BDD. The reason is that when the non-splittable compressed input files are used, the suggested input data split size specified by the DP configuration will not be honored by Spark (and Hadoop), as there is no clear split point on those inputs. In this situation, Spark (and Hadoop) will read and treat each compressed file as a single partition, which will result in a large amount of resources being consumed during the workflow.

If you must non-splittable compression, you should use block-based compression, where the data is divided into smaller blocks first and then the data is compressed within each block. More information is available at: *https://cwiki.apache.org/confluence/display/Hive/CompressedStorage*

In summary, you are encouraged to use splittable compression, such as BZip2. For information on choosing a data compression format, see: *http://www.cloudera.com/content/cloudera/en/documentation/core/v5-3-x/topics/admin_data_compression_performance.html*

Anti-Virus and Malware

Oracle strongly encourages you to use anti-virus products prior to uploading files into Big Data Discovery. The Data Processing component of BDD either finds Hive tables that are already present and then loads them, or lets you load data from new Hive tables, using DP CLI. In either case, use anti-virus software to ensure the quality of the data that is being loaded.

Chapter 2

Data Processing Workflows

This section describes how Data Processing discovers data in Hive tables and prepares it for ingest into the Dgraph.

Overview of workflows

This topic provides an overview of Data Processing workflows.

When the Data Processing component runs, it performs a series of steps; these steps are called a ***data processing workflow***. Many workflows exist, for loading initial data, updating data, or for cleaning up unused data sets.

All Data Processing workflows are launched either from Studio (in which case they run automatically) or from the DP CLI (Command Line Interface) utility.

In either case, when the workflow runs, it manifests itself in various parts of the user interface, such as **Explore**, and **Transform** in Studio. For example, new source data sets become available for your discovery, in **Explore**. Or, you can make changes to the project data sets in **Transform**. Behind all these actions, lie the processes in Big Data Discovery known as ***Data Processing workflows***. This guide describes these processes in detail.

For example, a Data Processing (DP) workflow for loading data is the process of extracting data and metadata from a Hive table and ingesting it as a data set in the Dgraph. The extracted data is turned into Dgraph records while the metadata provides the schema for the records, including the Dgraph attributes that define the BDD data set.

Once data sets are ingested into the Dgraph, Studio users can view the data sets and query the records in them. Studio users can also modify (transform) the data set and even delete it.

All Data Processing jobs are run by Spark workers. Data Processing runs asynchronously — it puts a Spark job on the queue for each Hive table. When the first Spark job on the first Hive table is finished, the second Spark job (for the second Hive table) is started, and so on.

Note that although a BDD data set can be deleted by a Studio user, the Data Processing component of BDD software can never delete a Hive table. Therefore, it is up to the Hive administrator to delete obsolete Hive tables.

Workflow Manager

When a Data Processing workflow is launched from Studio or the DP CLI, the workflow is instantiated and managed by the Workflow Manager Service.

Workflows are implemented in terms of state machines, to simplify their logic and to support asynchronous calls (within the service) and failure recovery. Each workflow has these well-defined states representing the life cycle of the workflow:

- `START_STATE` — The workflow job is started. A Spark job is built and submitted as a YARN job.
- `YARN_JOB_SUBMITTED_STATE` — The YARN job has been submitted. If the job is successfully completed, an ingest job is built and submitted to the Dgraph HDFS Agent.
- `INGEST_JOB_SUBMITTED_STATE` — The ingest job has been submitted. If the ingest was successful, the `ingestStatus` attribute for the data set is updated and a completion notification is sent to the client via the Notification Service.
- `SUCCEEDED_STATE` — The workflow job has successfully completed.
- `FAILED_STATE` — The workflow job failed.

These states are part of the contract between the Workflow Manager Service, its clients (e.g., Studio), and its dependencies (e.g., Dgraph HDFS Agent).

Workflow jobs are assigned a unique identifier. This job ID is returned to DP CLI jobs, which can then be monitored with the DP CLI `get-job-status` flag.

DataSet Inventory

The *DataSet Inventory* (DSI) is an internal structure that lets Data Processing keep track of the available data sets. Each data set in the DSI includes metadata that describes the characteristics of that data set. For example, when a data set is first created, the names of the source Hive table and the source Hive database are stored in the metadata for that data set. The metadata also includes the schemas of the data sets.

The DataSet Inventory contains an `ingestStatus` attribute for each data set, which indicates whether the data set has been completely provisioned (and therefore is ready to be added to a Studio project). The flag is set by the Workflow Manager after being notified by the Dgraph HDFS Agent on the completion of an ingest.

Language setting for attributes

During a normal Data Processing workflow, the language setting for all attributes is either a specific language (such as English or French) or `unknown` (which means a DP workflow does not use a language code for any specific language). The default language is set at install time for Studio and the DP CLI by the `LANGUAGE` property of the `bdd.conf` file. However, both Studio and the DP CLI can override the default language setting and specify a different language code for a workflow. For a list of supported languages, see *Supported languages on page 105*.

Workflow for loading new data

This topic discusses the workflow that runs inside Data Processing component of BDD when new data is loaded.

The Data Processing workflow shown in this topic is for loading data; it is one of many possible workflows. This workflow does not show updating data that has already been loaded. For information on running Refresh and Incremental update operations, see *Updating Data Sets on page 58*.

Loading new data includes these stages:

- Discovery of source data in Hive tables

- Loading and creating a sample of a data set

- Running a select set of enrichments on this data set (if so configured)

- Profiling the data

- Transforming the data set

- Exporting data from Big Data Discovery into Hadoop

You launch the Data Processing workflow for loading new data either from Studio (by creating a Hive table), or by running the Data Processing CLI (Command Line Interface) utility. As a Hadoop system administrator, you can control some steps in this workflow, while other steps run automatically in Hadoop.

The following diagram illustrates how the data processing workflow for loading new data fits within Big Data Discovery:

Data Processing workflow for loading new data
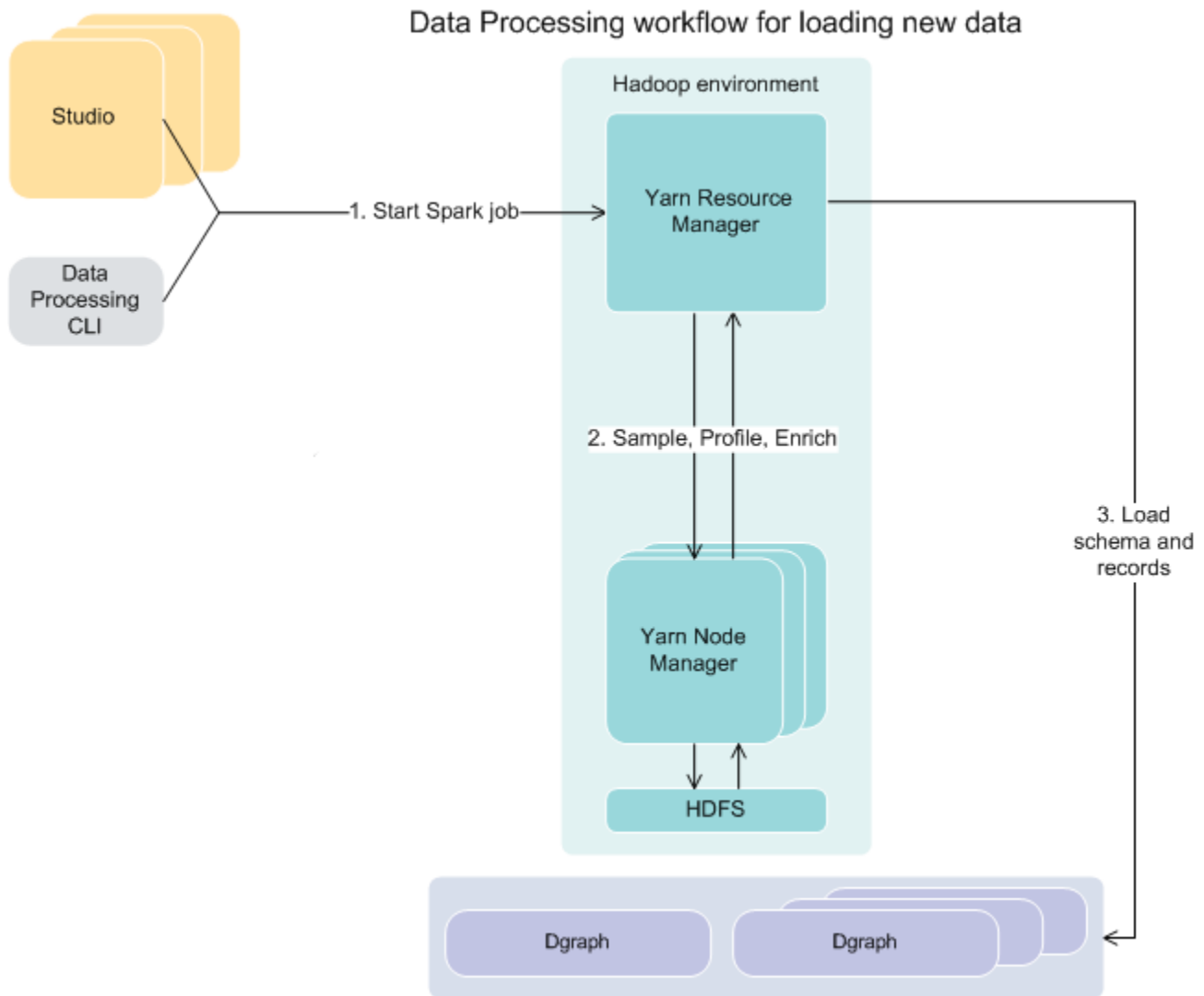
The steps in this diagram are:

1. The workflow for data loading starts either from Studio or the Data Processing CLI.

2. The Spark job is launched on Hadoop nodes that have Data Processing portion of Big Data Discovery installed on them.

3. The counting, sampling, discovery, and transformations take place and are processed on Hadoop nodes. The information is written to HDFS and sent back.

4. The data processing workflow launches the process of loading the records and their schema into the Dgraph, for each data set.

To summarize, during an initial data load, the Data Processing component of Big Data Discovery counts data in Hive tables, and optionally performs **data set sampling**. It then runs an initial data profiling, and applies some enrichments. These stages are discussed in this topic.

Sampling of a data set

If you work with a sampled subset of the records from large tables discovered in HDFS, you are using sample data as a proxy for the full tables. This lets you:

- Avoid latency and increase the interactivity of data analysis, in Big Data Discovery

- Analyze the data as if using the full set.

Data Processing does not always perform sampling; Sampling occurs only if a source data set contains more records than the default sample size used during BDD deployment. The default sample size used during deployment is 1 million records. When you subsequently run data processing workflow yourself, using the Command Line Interface (DP CLI), you can override the default sample size and specify your own.

> **Note:** If the number of records in the source data set is less than the value specified for the sample size, then no sampling takes place and Data Processing loads the source data in full.

Samples in BDD are taken as follows:

- Data Processing takes a random sample of the data, using either the default size sample, or the size you specify. BDD leverages the inbuilt Spark random sampling functionality.

- Based on the number of rows in the source data and the number of rows requested for the sample, BDD passes through the source data and, for each record, includes it in the sample with a certain (equal) probability. As a result, Data Processing creates a simple random sampling of records, in which:

    - Each element has the same probability of being chosen

    - Each subset of the same size has an equal probability of being chosen.

These requirements, combined with the large absolute size of the data sample, mean that samples taken by Big Data Discovery allow for making reliable generalizations on the entire corpus of data.

Profiling of a data set

Profiling is a process that determines the characteristics (columns) in the Hive tables, for each source Hive table discovered by the Data Processing in Big Data Discovery during data load.

Profiling is carried out by the data processing workflow for loading data and results in the creation of metadata information about a data set, including:

- Attribute value distributions

- Attribute type

- Topics

- Classification

For example, a specific data set can be recognized as a collection of structured data, social data, or geographic data.

Using **Explore** in Studio, you can then look deeper into the distribution of attribute values or types. Later, using **Transform**, you can change some of these metadata. For example, you can replace null attribute values with actual values, or fix other inconsistencies.

Enrichments

Enrichments are derived from a data set's additional information such as terms, locations, the language used, sentiment, and views. Big Data Discovery determines which enrichments are useful for each discovered data set, and automatically runs them on samples of the data. As a result of automatically applied enrichments, additional derived metadata (columns) are added to the data set, such as geographic data, a suggestion of the detected language, or positive or negative sentiment.

The data sets with this additional information appear in **Catalog** in Studio. This provides initial insight into each discovered data set, and lets you decide if the data set is a useful candidate for further exploration and analysis.

In addition to automatically-applied enrichments, you can also apply enrichments using **Transform** in Studio, for a project data set. From **Transform**, you can configure parameters for each type of enrichment. In this case, an enrichment is simply another type of available transformation.

Some enrichments allow you to add additional derived meaning to your data sets, while others allow you to address invalid or inconsistent values.

Transformations

Transformations are changes to a data set. Transformations allow you to perform actions such as:

- Changing data types
- Changing capitalization of values
- Removing attributes or records
- Splitting columns
- Grouping or binning values
- Extracting information from values

You can think of transformations as a substitute for an ETL process of cleaning your data before or during the data loading process. Use could transformations to overwrite an existing attribute, or create new attributes. Some transformations are enrichments, and as such, are applied automatically when data is loaded.

Most transformations are available directly as specific options in **Transform** in Studio. Once the data is loaded, you can use a list of predefined Transform functions, to create a transformation script.

For a full list of transformations available in BDD, including aggregations and joining of data sets, see the *Studio User's Guide*.

Exporting data from Big Data Discovery into HDFS

You can export the results of your analysis from Big Data Discovery into HDFS/Hive; this is known as **exporting to HDFS**.

From the perspective of Big Data Discovery, the process is about exporting the files from Big Data Discovery into HDFS/Hive. From the perspective of HDFS, you are importing the results of your work from Big Data Discovery into HDFS. In Big Data Discovery, the ***Dgraph HDFS Agent*** is responsible for exporting to HDFS and importing from it.

Working with Hive tables

Hive tables contain the data for the Data Processing workflows.

When processed, each Hive table results in the creation of a BDD data set, and that data set contains records from the Hive table. Note that a Hive table must contain at least one record in order for it to be processed. That is, Data Processing does not create a data set for an empty table.

Starting workflows

A Data Processing workflow can be started in one of two ways:

- A user in Studio invokes an operation that creates a new Hive table. After the Hive table is created, Studio starts the Data Processing process on that table.
- The DP CLI (Command Line Interface) utility is run.

The DP CLI, when run either manually or from a cron job, invokes the BDD Hive Table Detector, which can find a Hive table that does not already exist in the DataSet Inventory. A Data Processing workflow is then run on the table. For details on running the DP CLI, see *DP Command Line Interface Utility on page 42*.

New Hive table workflow and diagram

Both Studio and the DP CLI can be configured to launch a Data Processing workflow that does not use the Data Enrichment modules. The following high-level diagram shows a workflow in which the Data Enrichment modules are run:
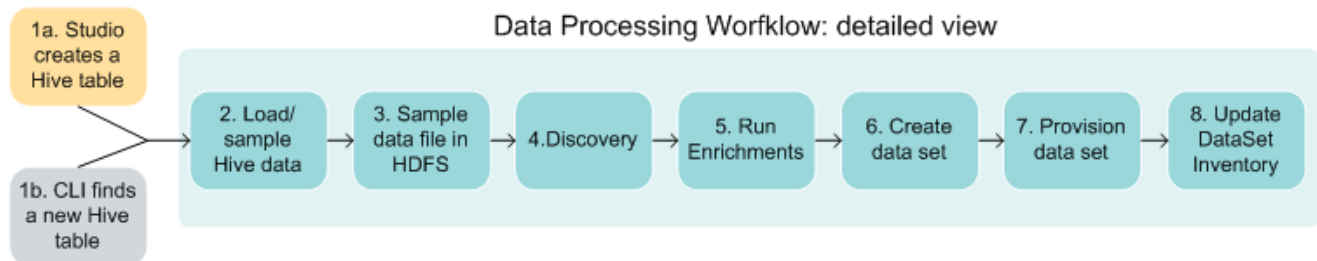
The steps in the workflow are:

1. The workflow is started for a single Hive table by Studio or by the DP CLI.

2. The job is started and the workflow is assigned to a Spark worker. Data is loaded from the Hive table's data files. The total number of rows in the table is counted, the data sampled, and a primary key is added. The number of processed (sampled) records is specified in the Studio or DP CLI configuration.

3. The data from step 2 is written to a Parquet file in HDFS. This file will remain in HDFS as long as the associated data set exists.

4. The data set schema and metadata are discovered. This includes discovering the data type of each column, such as long, geocode, and so on. (The DataSet Inventory is also updated with the discovered metadata. If the DataSet Inventory did not exist, it is created at this point.)

5. The Data Enrichment modules are run. A list of recommended enrichments is generated based on the results of the discovery process. The data is enriched using the recommended enrichments. If running enrichments is disabled in the configuration, then this step is skipped.

6. The data set is created in the Dgraph, using settings from steps 4 and 5. The DataSet Inventory is also updated to include metadata for the new data set.

7. The data set is provisioned (that is, HDFS files are written for ingest) and the Dgraph HDFS Agent is notified to pick up the HDFS files, which are sent to the Bulk Load Interface for ingesting into the Dgraph.

8. After provisioning has finished, Studio updates the `ingestStatus` attribute of the DataSet Inventory with the final status of the provisioning (ingest) operation.

Handling of updated Hive tables

Existing BDD data sets are not automatically updated if their Hive source tables are updated. For example, assume that a data set has been created from a specific Hive table. If that Hive table is updated with new data, the associated BDD data set is not automatically changed. This means that now the BDD data set is not in synch with its Hive source table.

To update the data set from the updated Hive table, you must run the DP CLI with either the `--refreshData` flag or the `--incrementalUpdate` flag. For details, see *Updating Data Sets on page 58*.

Handling of deleted Hive tables

BDD will never delete a Hive table, even if the associated BDD data set has been deleted from Studio. However, it is possible for a Hive administrator to delete a Hive table, even if a BDD data set has been created from that table. In this case, the BDD data set is not automatically deleted and will still be viewable in Studio. (A data set whose Hive source table was deleted is called an ***orphaned data set***.)

The next time that the DP CLI runs, it detects the orphaned data set and runs a Data Processing job that deletes the data set.

Handling of empty Hive tables

Data Processing does not process empty Hive tables. Instead, the Spark driver throws an `EmptyHiveTableException` when running against an empty Hive table. This causes the Data Processing job to not create a data set for the table. Note that the command may appear to have successfully finished, but the absence of the data set means the job ultimately failed.

Handling of Hive tables created with header/footer information

Data Processing does not support processing Hive tables that are based on files (such as CSV files) containing header/footer rows. In this case, the DP workflow will ignore the header and footer set on the Hive table using the `skip.header.line.count` and `skip.footer.line.count` properties. If a workflow on such a table does happen to succeed, the header/footer rows will get added to the resulting BDD data set as records, instead of being omitted.

Deletion of Studio projects

When a Studio user deletes a project, Data Processing is called and it will delete the transformed data sets in the project. However, it will not delete the data sets which have not been transformed.

Sampling and attribute handling

When creating a new data set, you can specify the maximum number of records that the Data Processing workflow should process from the Hive table.

The number of sampled records from a Hive table is set by the Studio or DP CLI configuration:

- In Studio, the `bdd.sampleSize` parameter in the **Data Processing Settings** page on Studio's Control Panel.

- In DP CLI, the `maxRecordsForNewDataSet` configuration parameter or the `--maxRecords` flag.

If the settings of these parameters are greater than the number of records in the Hive table, then all the Hive records are processed. In this case, the data set will be considered a full data set.

Discovery for attributes

The Data Processing discovery phase discovers the data set metadata in order to suggest a Dgraph attribute schema. For detailed information on the Dgraph schema, see *Dgraph Data Model on page 102*.

Record and value search settings for string attributes

When the DP data type discoverer determines that an attribute should be a string attributes, the settings for the record search and value search for the attribute are configured according to the settings of two properties in the `bdd.conf` file:

- The attribute is configured as record searchable if the average string length is greater than the `RECORD_SEARCH_THRESHOLD` property value.

- The attribute is configured as value searchable if the average string length is equal to or less than the `VALUE_SEARCH_THRESHOLD` property value.

In both cases, "average string length" refers to the average string length of the values for that column.

You can override this behavior by using the `--disableSearch` flag with the DP CLI. With this flag, the record search and value search settings for string attributes are set to false, regardless of the average String length of the attribute values. Note the following about using the `--disableSearch` flag:

- The flag can used only for provisioning workflows (when a new data set is created from a Hive table) and for refresh update workflows (when the DP CLI `--refreshData` flag is used). The flag cannot be used with any other type of workflow (for example, workflows that use the `--incrementalUpdate` flag are not supported with the `--disableSearch` flag).

- A disable search workflow can be run only with the DP CLI. This functionality is not available in Studio.

Effect of NULL values on column conversion

When a Hive table is being sampled, a Dgraph attribute is created for each column. The data type of the Dgraph attribute depends on how Data Processing interprets the values in the Hive column. For example, if the Hive column is of type String but it contains Boolean values only, the Dgraph attribute is of type `mdex:boolean`. NULL values are basically ignored in the Data Processing calculation that determines the data type of the Dgraph attribute.

Handling of Hive column names that are invalid Avro names

Data Processing uses Avro files to store data that should be ingested into the Dgraph (via the Dgraph HDFS Agent). In Avro, attribute names must start with an alphabetic or underscore character (that is, [A-Za-z_]), and the rest of the name can contain only alphanumeric characters and underscores (that is, [A-Za-z0-9_]).

Hive column names, however, can contain almost any Unicode characters, including characters that are not allowed in Avro attribute names. This format was introduced in Hive 0.13.0.

Because Data Processing uses Avro files to do ingest, this limits the names of Dgraph attributes to the same rules as Avro. This means that the following changes are made to column names when they are stored as Avro attributes:

- Any non-ASCII alphanumeric characters (in Hive column names) are changed to _ (the underscore).
- If the leading character is disallowed, that character is changed to an underscore and then the name is prefixed with "A_". As a result, the name would actually begin with "A__" (an A followed by two underscores).
- If the resulting name is a duplicate of an already-process column name, a number is appended to the attribute name to make it unique. This could happen especially with non-English column names.

For example:

```
Hive column name: @first-name

Changed name: A__first_name
```

In this example, the leading character (@) is not a valid Avro character and is, therefore, converted to an underscore (the name is also prefixed with "A_"). The hyphen is replaced with an underscore and the other characters are unchanged.

Attribute names for non-English tables would probably have quite a few underscore replacements and there could be duplicate names. Therefore, a non-English attribute name may look like this: A_____2

Data type discovery

When Data Processing retrieves data from a Hive table, the Hive data types are mapped to Dgraph data types when the data is ingested into the Dgraph.

The discovery phase of a workflow means that Data Processing discovers the data set metadata in order to determine the Dgraph attribute schema. Once Data Processing can ascertain what the data type is of a given Hive table column, it can map that Hive column data type to a Dgraph attribute data type.

For most types of workflows, the discovery phase is performed on the sample file. The exception is a Refresh update, which is a full data refresh on a BDD data set from the original Hive table.

Hive-to-Dgraph data conversions

When a Hive table is created, a data type is specified for each column (such as BOOLEAN or DOUBLE). During a Data Processing workflow, a Dgraph attribute is created for each Hive column. The Dgraph data type for the created attribute is based on the Hive column data type. For more information on the data model, including information about what are Dgraph records, and what are Dgraph attributes, see the section *Dgraph Data Model on page 102*.

This table lists the mappings for supported Hive data types to Dgraph data types. If a Hive data type is not listed, it is not supported by Data Processing and the data in that column will not be provisioned.

Hive Data Type	Hive Description	Dgraph Data Type Conversion
ARRAY<data_type>	Array of values of a Hive data type (such as, ARRAY<STRING>)	mdex:data_type-set where data_type is a Dgraph data type in this column. The -set data types are for multi-assign attributes (such as mdex:string-set).
BIGINT	8-byte signed integer.	mdex:long
BOOLEAN	Choice of TRUE or FALSE.	mdex:boolean
CHAR	Character string with a fixed length (maximum length is 255)	mdex:string
DATE	Represents a particular year/month/day, in the form: `YYYY-MM-DD` Date types do not have a time-of-day component. The range of values supported is 0000-01-01 to 9999-12-31.	mdex:dateTime
DECIMAL	Numeric with a precision of 38 digits.	mdex:double
DOUBLE	8-byte (double precision) floating point number.	mdex:double
FLOAT	4-byte (single precision) floating point number.	mdex:double
INT	4-byte signed integer.	mdex:long
SMALLINT	2-byte signed integer.	mdex:long
STRING	String values with a maximum of 32,767 bytes.	mdex:string A String column can be mapped as a Dgraph non-string data type if 100% of the values are actually in another data format, such as long, dateTime, and so on.
TIMESTAMP	Represents a point in time, with an optional nanosecond precision. Allowed date values range from 1400-01-01 to 9999-12-31.	mdex:dateTime

Hive Data Type	Hive Description	Dgraph Data Type Conversion
TINYINT	1-byte signed integer.	mdex:long
VARCHAR	Character string with a length specifier (between 1 and 65355)	mdex:string

Data type discovery for Hive string columns

If a Hive column is configured with a data type other than STRING, Data Processing assumes that the formats of the record values in that column are valid. In this case, a Dgraph attributes derived from the column automatically use the mapped Dgraph data type listed in the table above.

String columns, however, often store data that really is non-string data (for example, integers can be stored as strings). When it analyzes the content of Hive table string columns, Data Processing makes a determination as to what type of data is actually stored in each column, using this algorithm:

- If 100% of the column values are of a certain type, then the column values are ingested into the Dgraph as their Dgraph data type equivalents (see the table above).

- If the data types in the column are mixed (such as integers and dates), then the Dgraph data type for that column is string (mdex:string). The only exception to this rule is if the column has a mixture of integers and doubles (or floats); in this case, the data type maps to mdex:double (because an integer can be ingested as a double but not vice-versa).

For example, if the Data Processing discoverer concludes that a given string column actually stores geocodes (because 100% of the column values are proper geocodes), then those geocode values are ingested as Dgraph mdex:geocode data types. If however, 95% of the column values are geocodes but the other 5% are another data type, then the data type for the column defaults to the Dgraph mdex:string data type. Note, however, that double values that are in scientific notation (such as "1.4E-4") are evaluated as strings, not as doubles.

To take another example, if 100% of a Hive string column consists of integer values, then the values are ingested as Dgraph mdex:long data types. Any valid integer format is accepted, such as "10", "-10", "010", and "+10".

Space-padded values

Hive values that are padded with spaces are treated as follows:

- All integers with spaces are converted to strings (mdex:string)

- Doubles with spaces are converted to strings (mdex:string)

- Booleans with spaces are converted to strings (mdex:string)

- Geocodes are not affected even if they are padded with spaces.

- All date/time/timestamps are not affected even if they are padded with spaces.

Supported geocode formats

The following Hive geocode formats are supported during the discovery phase and are mapped to the Dgraph mdex:geocode data type:

```
Latitude Longitude
```

```
Latitude, Longitude
(Latitude Longitude)
(Latitude, Longitude)
```

For example:

```
40.55467767 -54.235
40.55467767, -54.235
(40.55467767 -54.235)
(40.55467767, -54.235)
```

Note that the comma-delimited format requires a space after the comma.

If Data Processing discovers any of these geocode formats in the column data, the value is ingested into the Dgraph as a geocode (`mdex:geocode`) attribute.

Supported date formats

Dates that are stored in Hive tables as `DATE` values are assumed to be valid dates for ingest. These `DATE` values are ingested as Dgraph `mdex:dateTime` data types.

For a date that is stored in a Hive table as a string, Data Processing checks it against a list of supported date formats. If the string date matches one of the supported date formats, then it is ingested as an `mdex:dateTime` data type. The date formats that are supported by Data Processing are listed in the `dateFormats.txt` file. Details on this file are provided in the topic *Date format configuration on page 36*.

In addition, Data Processing verifies that each date in a string column is a valid date. If a date is not valid, then the column is considered a string column, not a date column.

As an example of how a Hive column date is converted to a Dgraph date, a Hive date value of:

```
2013-10-23 01:23:24.1234567
```

will be converted to a Dgraph dateTime value of:

```
2013-10-23T05:23:24.123Z
```

The date will be ingested as a Dgraph `mdex:dateTime` data type.

Support of timestamps

Hive `TIMESTAMP` values are assumed to be valid dates and are ingested as Dgraph `mdex:dateTime` data types. Therefore, their format is not checked against the formats in the `dateFormats.txt` file.

When shown in Studio, Hive `TIMESTAMP` values will be formatted as "yyyy-MM-dd" or "yyyy-MM-dd HH:mm:ss" (depending on if the values in that column have times).

Note that if all values in a Hive timestamp column are not in the same format, then the time part in the Dgraph record becomes zero. For example, assume that a Hive column contains the following values:

```
2013-10-23 01:23:24
2012-09-22 02:24:25
```

Because both timestamps are in the same format, the corresponding values created in the Dgraph records are:

```
2013-10-23T01:23:24.000Z
2012-09-22T02:24:25.000Z
```

Now suppose a third row is inserted into that Hive table without the time part. The Hive column now has:

```
2013-10-23 01:23:24
```

```
2012-09-22 02:24:25
2007-07-23
```

In this case, the time part of the Dgraph records (the `mdex:dateTime` value) becomes zero:

```
2013-10-23T00:00:00.000Z
2012-09-22T00:00:00.000Z
2007-07-23T00:00:00.000Z
```

The reason is that if there are different date formats in the input data, then the Data Processing discoverer selects the more general format that matches all of the values, and as a result, the values that have more specific time information may end up losing some information.

To take another example, the pattern "yyyy-MM-dd" can parse both "2001-01-01" and "2001-01-01 12:30:23". However, a pattern like "yyyy-MM-dd hh:mm:ss" will throw an error when applied on the short string "2001-01-01". Therefore, the discoverer picks the best (longest possible) choice of "yyyy-MM-dd" that can match both "2001-01-01" and "2001-01-01 12:30:23". Because the picked pattern does not have time in it, there will be loss of precision.

Handling of unconvertible values

It is possible for your data to have column values that result in conversion errors (that is, where the original value cannot be converted to a Dgraph data type). Warnings are logged for the columns that contain conversion errors. For each column, one of the values that could not be converted is logged, as well as the total number of records that contained values that could not be converted. In addition, the values from the data set.

The following are examples of these log messages for unconvertible values:

```
[2016-03-16T16:01:43.315-04:00] [DataProcessing] [WARN] []
[com.oracle.endeca.pdi.logging.ProvisioningLogger] [tid:Driver] [userID:yarn]
Found 2 records containing unconvertible values (such as "2.718") for data source key type_tinyint.
These values could not be converted to type mdex:long and have been removed from the data set.

[2016-03-16T16:01:43.315-04:00] [DataProcessing] [WARN] []
[com.oracle.endeca.pdi.logging.ProvisioningLogger] [tid:Driver] [userID:yarn]
Found 4 records containing unconvertible values (such as "maybe") for data source key
type_string_as_boolean. These values could not be converted to type mdex:boolean and
have been removed from the data set.
```

Studio creation of Hive tables

Hive tables can be created from Studio.

The Studio user can create a Hive table by:

- Uploading data from a Microsoft Excel.

- Uploading data from delimited files, such as CSV, TSV, and TXT.

- Uploading data from compressed files, such as ZIP, GZ, and GZIP. A compressed file can include only one delimited file.

- Importing a JDBC data source.

- Exporting data from a Studio component.

- Transforming data in a data set and then creating a new data set from the transformed data.

After the Hive table is created, Studio starts a Data Processing workflow on the table. For details on these Studio operations, see the *Studio User's Guide*.

A Studio-created Hive table will have the `skipAutoProvisioning` property added at creation time. This property prevents the table from being processed again by the BDD Hive Table Detector.

Another table property will be `dataSetDisplayName`, which stores the display name for the data set. The display name is a user-friendly name that is visible in the Studio UI.

Chapter 3

Data Processing Configuration

This section describes configuration for the Workflow Manager and for attribute searchability. It also discusses how to add a SerDe JAR to the Data Processing workflows.

Workflow Manager configuration

Attribute searchability thresholds

Adding a SerDe JAR to DP workflows

Workflow Manager configuration

This topic describes how to modify three of the Workflow Manager configuration files.

The Workflow Manager `$BDD_HOME/workflowmanager/dp/config` directory contains these configuration files that you can modify to customize Data Processing workflows:

- The `edp.properties` file sets properties for DP workflows.

- The `dateFormats.txt` file provides the date formats supported by DP workflows.

- The `sparkContext.properties` file provides additional Spark configuration settings for DP workflows.

Note that the DP CLI also has a workflow configuration file (named `edp-cli.properties`) that is used in conjunction with the Workflow Manager `edp.properties` file. For details on the DP CLI `edp-cli.properties` file, see *DP CLI configuration on page 46*.

DP workflow properties

Date format configuration

Spark configuration

DP workflow properties

The Workflow Manager has a configuration file, `edp.properties`, that sets properties for Data Processing workflows.

The `edp.properties` file is located in the `$BDD_HOME/workflowmanager/dp/config` directory. Some of the default values for the properties are populated from the `bdd.conf` installation configuration file. After installation, you can change the workflow configuration parameters by opening the `edp.properties` file with a text editor.

> **Note:** The descriptions of these Workflow Manager configuration properties are provided only for informational purposes. Except for `extraJars`, you should not change these properties.

The DP CLI has a similar configuration file, named `edp-cli.properties`, that is also used for Provisioning and Refresh workflows launched by the DP CLI. For details on that file, see .

Data Processing defaults

The properties that set the Data Processing defaults are:

Data Processing Property	Description
`edpDataDir`	Specifies the location of the HDFS directory where data ingest and transform operations are processed. The default location is the `/user/bdd/<bdd-version>/edp/data` directory.
`notificationsServerUrl`	Specifies the URL of the Notification Service. This value is automatically set by the BDD installer and will have a value similar to this example: `http://web14.example.com:7003/bdd/v1/api`
`workflowManager`	Specifies the URL of the Workflow Manager. This value is automatically set by the BDD installer and will have a value similar to this example: `http://web14.example.com:7207/bdd.workflowmanager`

Dgraph Gateway connectivity settings

These properties are used to control access to the Dgraph Gateway that is managing the Dgraph nodes:

Dgraph Gateway Property	Description
`endecaServerHost`	The name of the host on which the Dgraph Gateway is running. The default name is specified in the `bdd.conf` configuration file.
`endecaServerPort`	The port on which Dgraph Gateway is listening. The default is 7003.
`endecaServerContextRoot`	The context root of the Dgraph Gateway when running on Managed Servers within the WebLogic Server. The value should be set to: `/endeca-server`

Kerberos credentials

The DP CLI is enabled for Kerberos support at installation time, if the `ENABLE_KERBEROS` property in the `bdd.conf` file is set to TRUE. The `bdd.conf` file also has parameters for specifying the name of the Kerberos principal, as well as paths to the Kerberos keytab file and the Kerberos configuration file. The installation script populates the `edp.properties` file with the properties in the following table.

Kerberos Property	Description
isKerberized	Specifies whether Kerberos support should be enabled. The default value is set by the ENABLE_KERBEROS property in the bdd.conf file.
localKerberosPrincipal	The name of the Kerberos principal. The default name is set by the KERBEROS_PRINCIPAL property in the bdd.conf file.
localKerberosKeytabPath	Path to the Kerberos keytab file on the WebLogic Admin Server. The default path is set by the KERBEROS_KEYTAB_PATH property in the bdd.conf file.
clusterKerberosPrincipal	The name of the Kerberos principal. The default name is set by the KERBEROS_PRINCIPAL property in the bdd.conf file.
clusterKerberosKeytabPath	Path to the Kerberos keytab file on the WebLogic Admin Server. The default path is set by the KERBEROS_KEYTAB_PATH property in the bdd.conf file.
krb5ConfPath	Path to the krb5.conf configuration file. This file contains configuration information needed by the Kerberos V5 library. This includes information describing the default Kerberos realm, and the location of the Kerberos key distribution centers for known realms. The default path is set by the KRB5_CONF_PATH property in the bdd.conf file. However, you can specify a local, custom location for the krb5.conf file.

For further details on these parameters, see the *Installation Guide*

Hadoop connectivity settings

The parameters that define connections to Hadoop environment processes and resources are:

Hadoop Parameter	Description
hiveServerHost	Name of the host on which the Hive server is running. The default value is set at the BDD installation time.
hiveServerPort	Port on which the Hive server is listening. The default value is set at the BDD installation time.
clusterOltHome	Path to the OLT directory on the Spark worker node. The default location is the $BDD_HOME/common/edp/olt directory.
oltHome	Both clusterOltHome and this parameter are required, and both must be set to the same value.

Hadoop Parameter	Description
hadoopClusterType	The installation type, according to the Hadoop distribution. The value is set by the INSTALL_TYPE property in the bdd.conf file.
hadoopTrustStore	Path to the directory on the install machine where the certificates for HDFS, YARN, Hive, and the KMS are stored. Required for clusters with TLS/SSL enabled. The default path is set by the HADOOP_CERTIFICATES_PATH property in the bdd.conf file.

Spark environment settings

These parameters define settings for interactions with Spark workers:

Spark Properties	Description
sparkMasterUrl	Specifies the master URL of the Spark cluster. In Spark-on-YARN mode, the ResourceManager's address is picked up from the Hadoop configuration by simply specifying yarn-cluster for this parameter. The default value is set at the BDD installation time.
sparkDynamicAllocation	Indicates if Data Processing will dynamically compute the executor resources or use static executor resource configuration: • If set to **false**, the values of the static resource parameters (sparkDriverMemory, sparkDriverCores, sparkExecutorMemory, sparkExecutorCores, and sparkExecutors) are required and are used. • If set to **true**, the values for the executor resources are dynamically computed. This means that the static resource parameters are not required and will be ignored even if specified. The default is set by the SPARK_DYNAMIC_ALLOCATION property in the bdd.conf file.
sparkDriverMemory	Amount of memory to use for each Spark driver process, in the same format as JVM memory strings (such as 512m, 2g, 10g, and so on). The default is set by the SPARK_DRIVER_MEMORY property in the bdd.conf file.
sparkDriverCores	Maximum number of CPU cores to use by the Spark driver. The default is set by the SPARK_DRIVER_CORES property in the bdd.conf file.

Spark Properties	Description
sparkExecutorMemory	Amount of memory to use for each Spark executor process, in the same format as JVM memory strings (such as 512m, 2g, 10g, and so on). The default is set by the SPARK_EXECUTOR_MEMORY property in the bdd.conf file.
	This setting must be less than or equal to Spark's **Total Java Heap Sizes of Worker's Executors in Bytes** (executor_total_max_heapsize) property in Cloudera Manager. You can access this property in Cloudera Manager by selecting **Clusters > Spark (Standalone)**, then clicking the **Configuration** tab. This property is in the **Worker Default Group** category (using the classic view).
sparkExecutorCores	Maximum number of CPU cores to use for each Spark executor. The default is set by the SPARK_EXECUTOR_CORES property in the bdd.conf file.
sparkExecutors	Total number of Spark executors to launch. The default is set by the SPARK_EXECUTORS property in the bdd.conf file.
yarnQueue	The YARN queue to which the Data Processing job is submitted. The default value is set by the YARN_QUEUE property in the bdd.conf file.
maxSplitSizeMB	The maximum partition size for Spark inputs, in MB. This controls the size of the blocks of data handled by Data Processing jobs. This property overrides the HDFS block size used in Hadoop.
	Partition size directly affects Data Processing performance — when partitions are smaller, more jobs run in parallel and cluster resources are used more efficiently. This improves both speed and stability.
	The default is set by the MAX_INPUT_SPLIT_SIZE property in the bdd.conf file (which is 32, unless changed by the user). The 32MB is amount should be sufficient for most clusters, with a few exceptions:
	• If your Hadoop cluster has a very large processing capacity and most of your data sets are small (around 1GB), you can decrease this value.
	• In rare cases, when data enrichments are enabled the enriched data set in a partition can become too large for its YARN container to handle. If this occurs, you can decrease this value to reduce the amount of memory each partition requires.
	If this property is empty, DP logs an error at start-up and uses a default value of 32MB.

Jar location settings

These properties specify the paths for jars used by workflows:

Jar Property	Description
sparkYarnJar	Path to JAR files used by Spark-on-YARN. The default path is set by the SPARK_ON_YARN_JAR property in the bdd.conf file. However, additional JARs (such as edpLogging.jar) are appended to the path by the installer.
bddHadoopFatJar	Path to the location of the Hadoop Shared Library (file name of bddHadoopFatJar.jar) on the cluster. The path is set by the installer. and is typically the $BDD_HOME/common/hadoop/lib directory. Note that the data_processing_CLI script has a BDD_HADOOP_FATJAR property that specifies the location of the Hadoop Shared Library on the local file system of the DP CLI client.
edpJarDir	Path to the directory where the Data Processing JAR files for Spark workers are located on the cluster. The default location is the $BDD_HOME/common/edp/lib directory.
extraJars	Path to any extra JAR files to be used by customers, such as the path to a custom SerDe JAR. The default path is set by the DP_ADDITIONAL_JARS property in the bdd.conf file.

Kryo serialization settings

These properties define the use of Kryo serialization:

Kryo Property	Description
kryoMode	Specifies whether to enable (true) or disable (false) Kryo for serialization. Make sure that this property is set to false because Kryo serialization is not supported in BDD.
kryoBufferMemSizeMB	Maximum object size (in MBs) to allow within Kryo. This property, like the kryoMode property, is not supported by BDD workflows.

Date format configuration

The `dateFormats.txt` file provides a list of date formats supported by Data Processing workflows. This topic lists the defaults used in this file. You can add or remove a date format from this file if you use the formats supported by it.

If a date in the Hive table is stored with a `DATE` data type, then it is assumed to be a valid date format and is not checked against the date formats in the `dateFormats.txt` file. Hive `TIMESTAMP` values are also assumed to be valid dates, and are also not checked against the `dateFormats.txt` formats.

However, if a date is stored in the Hive table within a column of type `STRING`, then Data Processing uses the `dateFormats.txt` to check if this date format is supported. Both dates and timestamps are then ingested into the Big Data Discovery as Dgraph `mdex:dateTime` data types.

The `dateFormats.txt` file is located in the `$BDD_HOME/workflowmanager/dp/config` directory.

> **Note:** The DP CLI configuration directory (`$BDD_HOME/dataprocessing/edp_cli/config`) has an identically-named file. However, only the file in the `$BDD_HOME/workflowmanager/dp/config` directory is used by the Workflow Manager. Therefore, make sure to modify the Workflow Manager file to change the supported date formats.

Default date formats

The default date formats that are supported and listed in the `dateFormats.txt` file are:

```
d/M/yy
d-M-yy
d.M.yy
M/d/yy
M-d-yy
M.d.yy
yy/M/d
yy-M-d
yy.M.d
MMM d, yyyy
EEE, MMM d, yyyy
yyyy-MM-dd HH:mm:ss
yyyy-MM-dd h:mm:ss a
yyyy-MM-dd'T'HH-mm-ssZ
yyyy-MM-dd'T'HH:mm:ss'Z'
yyyy-MM-dd'T'HH:mm:ss.SSS'Z'
yyyy-MM-dd HH:mm:ss.SSS
yyyy-MM-dd'T'HH:mm:ss.SSS
EEE d MMM yyyy HH:mm:ss Z
H:mm
h:mm a
H:mm:ss
h:mm:ss a
HH:mm:ss.SSS'Z'
d/M/yy HH:mm:ss
d/M/yy h:mm:ss a
d-M-yy HH:mm:ss
d-M-yy h:mm:ss a
d.M.yy HH:mm:ss
d.M.yy h:mm:ss a
M/d/yy HH:mm:ss
M/d/yy h:mm:ss a
M-d-yy HH:mm:ss
M-d-yy h:mm:ss a
M.d.yy HH:mm:ss
M.d.yy h:mm:ss a
yy/M/d HH:mm:ss
yy/M/d h:mm:ss a
```

```
yy.M.d HH:mm:ss
yy.M.d h:mm:ss a
```

For details on interpreting these formats, see
http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html

Modifying the dateFormats file

You can remove a date format from the file. If you remove a data format, Data Processing workflows will no longer support it.

You can also add date formats, as long as they conform to the formats in the `SimpleDateFormat` class. This class is described in the Web page accessed by the URL link listed above. Note that US is used as the locale.

Spark configuration

Data Processing uses a Spark configuration file, `sparkContext.properties`. This topic describes how Data Processing obtains the settings for this file and includes a sample of the file. It also describes options you can adjust in this file to tweak the amount of memory required to successfully complete a Data Processing workflow.

Data Processing workflows are run by Spark workers. When a Spark worker is started for a Data Processing job, it has a set of default configuration settings that can be overridden or added to by the `sparkContext.properties` file. This file is located in the `$BDD_HOME/workflowmanager/dp/config` directory.

> **Note:** The DP CLI configuration directory (`$BDD_HOME/dataprocessing/edp_cli/config`) has an identically-named `sparkContext.properties` file. However, only the file in the `$BDD_HOME/workflowmanager/dp/config` directory is used by the Workflow Manager. Therefore, make sure to modify the Workflow Manager file to change the Spark configuration.

The Spark configuration is very granular and needs to be adapted to the size of the cluster and also the data. In addition, the timeout and failure behavior may have to be altered. Spark offers an excellent set of configurable options for these purposes that you can use to configure Spark for the needs of your installation. For this reason, the `sparkContext.properties` is provided so that you can fine tune the performance of the Spark workers.

By default, the `sparkContext.properties` file is empty. However, you can add any Spark configuration property to the file. The properties that you specify will override all previously-set Spark settings. The documentation for the Spark properties is at: *https://spark.apache.org/docs/latest/configuration.html*

Keep in mind that the `sparkContext.properties` file can be empty. If the file is empty, a Data Processing workflow will still run correctly because the Spark worker will have a sufficient set of configuration properties to do its job.

> **Note:** Do not delete the `sparkContext.properties` file. Although it can be empty, a check is made for its existence and the Data Processing workflow will fail if the file is missing.

Spark default configuration

When started, a Spark worker gets its configuration settings in a three-tiered manner, in this order:

1. From the Hadoop default settings.

2. From the Data Processing configuration settings, which can either override the Hadoop settings, and/or provide additional settings. For example, the `sparkExecutorMemory` property (in the DP CLI configuration) can override the Hadoop `spark.executor.memory` property.

3. From the property settings in the `sparkContext.properties` file, which can either override any previous settings and/or provide additional settings.

If the `sparkContext.properties` file is empty, then the final configuration for the Spark worker is obtained from Steps 1 and 2.

Sample Spark configuration

The following is a sample `sparkContext.properties` configuration file:

```
###########################################################
# Spark additional runtime properties
###########################################################
spark.broadcast.compress=true
spark.rdd.compress=false
spark.io.compression.codec=org.apache.spark.io.LZFCompressionCodec
spark.io.compression.snappy.block.size=32768
spark.closure.serializer=org.apache.spark.serializer.JavaSerializer
spark.serializer.objectStreamReset=10000
spark.kryo.referenceTracking=true
spark.kryoserializer.buffer.mb=2
spark.broadcast.factory=org.apache.spark.broadcast.HttpBroadcastFactory
spark.broadcast.blockSize=4096
spark.files.overwrite=false
spark.files.fetchTimeout=false
spark.storage.memoryFraction=0.6
spark.tachyonStore.baseDir=System.getProperty("java.io.tmpdir")
spark.storage.memoryMapThreshold=8192
spark.cleaner.ttl=(infinite)
```

Configuring fail fast behavior for transforms

When a transform is committed, the ApplyTransformToDataSetWorkflow will not retry on failure. This workflow cannot safely be re-run after failure because the state of the data set may be out of sync with the state of the HDFS sample files. This non-retry behavior applies to all Hadoop environments.

Users can modify the `yarn.resourcemanager.am.max-attempts` setting on their cluster to prevent retries of any YARN job. If users do not do this, it may look like the workflow succeeded, but will fail on future transforms because of the inconsistent sample data files. Users do not have to set this property unless they want the fail fast behavior.

Enabling Spark event logging

You can enable Spark event logging with this file. At runtime, Spark internally compiles the DP workflow into multiple stages (a stage is usually defined by a set of Spark Transformation and bounded by Spark Action). The stages can be matched to the DP operations. The Spark event log includes the detailed timing information on a stage and all the tasks within the stage.

The following Spark properties are used for Spark event logging:

- `spark.eventLog.enabled` (which set to true) enables the logging of Spark events.

- `spark.eventLog.dir` specifies the base directory in which Spark events are logged.

- `spark.yarn.historyServer.address` specifies the address of the Spark history server (i.e., host.com:18080). The address should not contain a scheme (http://).

For example:

```
spark.eventLog.enabled=true
spark.eventLog.dir=hdfs://busj40CDH3-ns/user/spark/applicationHistory
spark.yarn.historyServer.address=busj40bda13.example.com:18088
```

Note that enabling Spark event logging should be done by Oracle Support personnel when trouble-shooting problems. Enabling Spark event logging under normal circumstances is not recommended as it can have an adverse performance impact on workflows.

Spark worker OutOfMemoryError

If insufficient memory is allocated to a Spark worker, an `OutOfMemoryError` may occur and the Data Processing workflow may terminate with an error message similar to this example:

```
java.lang.OutOfMemoryError: Java heap space
    at java.util.Arrays.copyOf(Arrays.java:2271)
    at java.io.ByteArrayOutputStream.grow(ByteArrayOutputStream.java:113)
    at java.io.ByteArrayOutputStream.ensureCapacity(ByteArrayOutputStream.java:93)
    at java.io.ByteArrayOutputStream.write(ByteArrayOutputStream.java:140)
    at java.io.BufferedOutputStream.flushBuffer(BufferedOutputStream.java:82)
    at java.io.BufferedOutputStream.write(BufferedOutputStream.java:126)
    at java.io.ObjectOutputStream$BlockDataOutputStream.drain(ObjectOutputStream.java:1876)
    at java.io.ObjectOutputStream$BlockDataOutputStream.setBlockDataMode(ObjectOutputStream.java:1785)
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1188)
    at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:347)
    at org.apache.spark.serializer.JavaSerializationStream.writeObject(JavaSerializer.scala:42)
    at org.apache.spark.serializer.SerializationStream$class.writeAll(Serializer.scala:102)
    at org.apache.spark.serializer.JavaSerializationStream.writeAll(JavaSerializer.scala:30)
    at org.apache.spark.storage.BlockManager.dataSerializeStream(BlockManager.scala:996)
    at org.apache.spark.storage.BlockManager.dataSerialize(BlockManager.scala:1005)
    at org.apache.spark.storage.MemoryStore.putValues(MemoryStore.scala:79)
    at org.apache.spark.storage.BlockManager.doPut(BlockManager.scala:663)
    at org.apache.spark.storage.BlockManager.put(BlockManager.scala:574)
    at org.apache.spark.CacheManager.getOrCompute(CacheManager.scala:108)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:227)
    at org.apache.spark.rdd.MappedRDD.compute(MappedRDD.scala:31)
    at org.apache.spark.rdd.RDD.computeOrReadCheckpoint(RDD.scala:262)
    at org.apache.spark.rdd.RDD.iterator(RDD.scala:229)
    at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:111)
    at org.apache.spark.scheduler.Task.run(Task.scala:51)
    at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:187)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:745)
```

The amount of memory required to successfully complete a Data Processing workflow depends on database considerations such as:

- The total number of records in each Hive table.
- The average size of each Hive table record.

It also depends on the DP CLI configuration settings, such as:

- `maxRecordsForNewDataSet`
- `runEnrichment`
- `sparkExecutorMemory`

If `OutOfMemoryError` instances occur, you can adjust the DP CLI default values, as well as specify `sparkContext.properties` configurations, to suit the provisioning needs of your deployment.

For example, Data Processing allows you to specify a `sparkExecutorMemory` setting, which is used to define the amount of memory to use per executor process. (This setting corresponds to the `spark.executor.memory` parameter in the Spark configuration.) The Spark `spark.storage.memoryFraction` parameter is another important option to use if the Spark Executors are having memory issues.

You should also check the "Tuning Spark" topic: *http://spark.apache.org/docs/latest/tuning.html*

Benign sparkDriver shutdown error

After a Spark job finishes successfully, you may see a sparkDriver shutdown ERROR message in the log, as in this abbreviated example:

```
...
11:11:42.828 Thread-2 INFO : Shutting down all executors
11:11:42.829 sparkDriver-akka.actor.default-dispatcher-19 INFO : Asking each executor to shut down
11:11:42.892 sparkDriver-akka.actor.default-dispatcher-17 ERROR: AssociationError [akka.tcp:/
/sparkDriver@10.152.110.203:62743] <- [akka.tcp://sparkExecutor
@atm.example.com:30203]: Error [Shut down address: akka.tcp://sparkExecutor
@bus00atm.us.oracle.com:30203] [
akka.remote.ShutDownAssociation: Shut down address: akka.tcp://sparkExecutor@atm.example.com:30203
Caused by: akka.remote.transport.Transport$InvalidAssociationException: The remote system terminated
the association because it is shutting down.
]
akka.event.Logging$Error$NoCause$
11:11:42.893 sparkDriver-akka.actor.default-dispatcher-19 INFO : Driver terminated or disconnected!
Shutting down. atm.example.com:30203
11:11:42.897 sparkDriver-akka.actor.default-dispatcher-19 INFO : MapOutputTrackerMasterEndpoint
stopped!
...
```

The actual Spark work is done successfully. However, the sparkDriver shutdown generates the error message. The log message is displayed by Spark (not the Data Processing code). The message is benign and there is no actual impact to functionality.

Note on differentiating job queuing and cluster locking

Sites that have a small and busy cluster may encounter problems with Spark jobs not running with a message similar to the following example:

```
[DataProcessing] [WARN] [] [org.apache.spark.Logging$class] [tid:Timer-0] [userID:yarn]
Initial job has not accepted any resources
; check your cluster UI to ensure that workers are registered
and have sufficient memory
```

The cause may be due to normal YARN job queuing rather than cluster locking. (Cluster locking is when a cluster is deadlocked by submitting many applications at once, and having all cluster resources taken up by the ApplicationManagers.) The appearance of the normal YARN job queuing is very similar to cluster locking, especially when there is a large YARN job taking excess time to run. To check on the status of jobs, use the Hadoop cluster manager for your Hadoop distribution.

The following information may help differentiate between job queuing and suspected cluster locking: Jobs are in normal queuing state unless there are multiple jobs in a RUNNING state, and you observe "Initial job has not accepted any resources" in the logs of *all these jobs*. As long as there is one job making progress where you usually see "Starting task X.X in stage X.X", those jobs are actually in normal queuing state. Also, when

checking Spark RUNNING jobs through ResourceManager UI, you should browse beyond the first page or use the Search box in the UI, so that no RUNNING applications are left out.

If your Hadoop cluster has a Hadoop version earlier than 2.6.0., it is recommended that the explicit setting is used to limit the ApplicationMaster share:

```
<queueMaxAMShareDefault>0.5</queueMaxAMShareDefault>
```

This property limits the fraction of the queue's fair share that can be used to run Application Masters.

Attribute searchability thresholds

Two properties in the `bdd.conf` file set the threshold that determines whether a string attribute should be record searchable and value searchable.

When you install BDD, the `bdd.conf` configuration file has these properties that control searchability for all string attributes:

- The `RECORD_SEARCH_THRESHOLD` property sets the threshold to enable record search for the attribute. The attribute is configured as record searchable if its average string length is greater than the threshold.

- The `VALUE_SEARCH_THRESHOLD` property sets the threshold to enable value search for the attribute. The attribute is configured as value searchable if its average string length is equal to or less than the threshold.

In both cases, "average string length" refers to the average string length of all the values in that attribute.

This topic describes how you can change the installed threshold values to new ones.

To configure the attribute searchability thresholds:

1. On the Admin Server, use a text editor to open `bdd.conf` in the `$BDD_HOME/BDD_manager/conf` directory.

    You may want to make a back up of the file before editing it.

2. Change the settings of the `RECORD_SEARCH_THRESHOLD` and/or `VALUE_SEARCH_THRESHOLD` properties. The value must be a Java integer.

    For example:

    ```
    ## The threshold length of a Dgraph attribute to enable record-search
    RECORD_SEARCH_THRESHOLD=250

    ## The threshold length of a Dgraph attribute to enable value-search
    VALUE_SEARCH_THRESHOLD=150
    ```

3. Go to `$BDD_HOME/BDD_manager/bin` and run this command:

    ```
    ./bdd-admin.sh publish-config bdd <path>
    ```

    where *<path>* is the absolute path to the `bdd.conf` file.

    The command will publish the `bdd.conf` file to all BDD nodes, and also to the HDFS `/user/$HDFS_DP_USER_DIR/edp/data` directory.

4. Restart your BDD cluster so the changes take effect in the BDD nodes:

    ```
    ./bdd-admin.sh restart
    ```

    Note that you do not have to restart your Hadoop cluster.

When a Data Processing workflow runs, it will read the `bdd.conf` in HDFS to obtain the attribute searchability thresholds for its discovery phase. If the workflow cannot find or read the file for any reason, then it will use a hardcoded value of 200 for both thresholds.

Adding a SerDe JAR to DP workflows

This topic describes the process of adding a custom Serializer-Deserializer (SerDe) to the Data Processing (DP) classpath.

When customers create a Hive table, they can specify a Serializer-Deserializer (SerDe) class of their choice. For example, consider the last portion of this statement:

```
CREATE TABLE samples_table(
    id INT,
    city STRING,
    country STRING,
    region STRING,
    population INT)
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.JsonSerde';
```

If that SerDe JAR is not packaged with the Data Processing package that is part of the Big Data Discovery, then a Data Processing run is unable to read the Hive table, which prevents the importing of the data into the Dgraph. To solve this problem, you can integrate your custom SerDe into the Data Processing workflow.

This procedure assumes this pre-requisite:

* Before integrating the SerDe JAR with Data Processing, the SerDe JAR should be present on the Hadoop cluster's HiveServer2 node and configured via the **Hive Auxiliary Jars Directory** property in the Hive service. To check this, you can verify that, for a table created with this SerDe, a `SELECT *` query on the table does not issue an error. This query should be verified to work from Hue and the Hive CLI to ensure the SerDe was added properly.

To integrate a custom SerDe JAR into the Data Processing workflow:

1. Copy the SerDe JAR into the same location on each cluster node.

    Note that this location can be the same one as used when adding the SerDe Jar to the HiveServer2 node.

2. Edit the Workflow Manager `edp.properties` file and add the path to the SerDe JAR to the `extraJars` property. This property should be a colon-separated list of paths to JARs. This will allow DP jobs from Studio and the CLI to pick up the SerDe JAR.

    The `edp.properties` file is in the `$BDD_HOME/workflowmanager/dp/config` directory.

As a result, the SerDe JAR is added in the Data Processing classpath. This means that the SerDe class will be used in all Data Processing workflows, whether they are initiated automatically by Studio or by running the Data Processing CLI.

Chapter 4

DP Command Line Interface Utility

This section provides information on configuring and using the Data Processing Command Line Interface utility.

DP CLI overview

DP CLI permissions and logging

DP CLI configuration

DP CLI flags

Using whitelists and blacklists

DP CLI cron job

DP CLI workflow examples

Processing Hive tables with Snappy compression

Changing Hive table properties

DP CLI overview

The DP CLI (Command Line Interface) shell utility is used to launch Data Processing workflows, either manually or via a cron job.

The Data Processing workflow can be run on an individual Hive table, all tables within a Hive database, or all tables within Hive. The tables must be of the auto-provisioned type (as explained further in this topic).

The DP CLI starts workflows that are run by Spark workers. The results of the DP CLI workflow are the same as if the tables were processed by a Studio-generated Data Processing workflow.

Two important use cases for the DP CLI are:

- Ingesting data from your Hive tables immediately after installing the Big Data Discovery (BDD) product. When you first install BDD, your existing Hive tables are not processed. Therefore, you must use the DP CLI to launch a first-time Data Processing operation on your tables.
- Invoking the BDD Hive Table Detector, which in turn can start Data Processing workflows for new or deleted Hive tables.

The DP CLI can be run either manually or from a cron job. The BDD installer creates a cron job as part of the installation procedure if the `ENABLE_HIVE_TABLE_DETECTOR` property is set to TRUE in the `bdd.conf` file.

DP CLI script

The DP CLI is run via the `data_processing_CLI` script. The script is located in the `$BDD_HOME/dataprocessing/edp_cli` directory. The directory also contains the `bdd-sp.sh` script, which is a symbolic link to the `data_processing_CLI` script. You can therefore run the DP CLI with either script.

The following abbreviated example shows an invocation of the DP CLI, which is being run against a Hive table name "warrantyclaims" in the "default" database:

```
./data_processing_CLI --database default --table warrantyclaims
...
[2016-09-29T13:10:17.702-04:00] [DataProcessing] [INFO] []
  [org.apache.hadoop.hive.metastore.HiveMetaStoreClient] [tid:main]
  [userID:fcalvin Connected to metastore.
New collection name = MdexCollectionIdentifier{
   databaseName=edp_cli_edp_37396cf4-fab1-40c8-bb08-d5bb09478d58,
   collectionName=edp_cli_edp_37396cf4-fab1-40c8-bb08-d5bb09478d58}
   jobId: 1fb5df0e-66cd-4ce5-8ff8-5b93e2ee2df0
data_processing_CLI finished with state SUCCESS
```

When creating a new data set (as in this example), the command output will list the names of the data set (in the `collectionName` field) and the Dgraph database (in the `databaseName` field), as well as the ID of the workflow job.

Note that the "finished with state SUCCESS" message means that the workflow was successfully launched; it does not mean that the workflow was successful. This means that is possible for a workflow to be launched successfully, but fail to finish. A notification of the workflow's final status is sent to Studio, as well as written to the logs.

Skipped and auto-provisioned Hive tables

From the point of view of Data Processing, there are two types of Hive tables: skipped tables and auto-provisioned tables. The table type depends on the presence of a special table property, `skipAutoProvisioning`. The `skipAutoProvisioning` property (when set to `true`) tells the BDD Hive Table Detector to skip the table for processing.

Skipped tables are Hive tables that have the `skipAutoProvisioning` table property present and set to `true`. Thus, a Data Processing workflow will never be launched for a skipped table (unless the DP CLI is run manually with the `--table` flag set to the table). This property is set in two instances:

• The table was created from Studio, in which case the `skipAutoProvisioning` property is always set at table creation time.

• The table was created by a Hive administrator and a corresponding BDD data set was provisioned from that table. Later, that data set was deleted from Studio. When a data set (from an admin-created table) is deleted, Studio modifies the underlying Hive table by adding the `skipAutoProvisioning` table property.

For information on changing the value of the `skipAutoProvisioning` property, see .

Auto-provisioned tables are Hive tables that were created by the Hive administrator and do not have a `skipAutoProvisioning` property. These tables can be provisioned by a Data Processing workflow that is launched by the BDD Hive Table Detector.

> ✏ **Note:** Keep in mind that when a BDD data set is deleted, its source Hive table is not deleted from the Hive database. This applies to data sets that were generated from either Studio-created tables or admin-created tables. The `skipAutoProvisioning` property ensures that the table will not be re-

provisioned when its corresponding data set is deleted (otherwise, the deleted data set would re-appear when the table was re-processed).

BDD Hive Table Detector

The BDD Hive Table Detector is a process that automatically keeps a Hive database in sync with the BDD data sets. The BDD Hive Table Detector has two major functions:

- Automatically checks all Hive tables within a Hive database:
  - For each auto-provisioned table that does not have a corresponding BDD data set, the BDD Hive Table Detector launches a new data provisioning workflow (unless the table is skipped via the blacklist).
  - For all skipped tables, such as, Studio-created tables, the BDD Hive Table Detector never provisions them, even if they do not have a corresponding BDD data set.
- Automatically launches the data set clean-up process if it detects that a BDD data set does not have an associated Hive table. (That is, an orphaned BDD data set is automatically deleted if its source Hive table no longer exists.) Typically, this scenario occurs when a Hive table (either admin-created or Studio-created) has been deleted by a Hive administrator.

The BDD Hive Table Detector detects empty tables, and does not launch workflows for those tables.

The BDD Hive Table Detector is invoked with the DP CLI, which has command flags to control the behavior of the script. For example, you can select the Hive tables you want to be processed. The `--whitelist` flag of the CLI specifies a file listing the Hive tables that should be processed, while the `--blacklist` flag controls a file with Hive tables that should be filtered out during processing.

DP CLI permissions and logging

This topic provides brief overviews of permissions and logging.

DP CLI permissions

The DP CLI script is installed with ownership permission for the person who ran the installer. These permissions can be changed by the owner to allow anyone else to run the script.

In addition, if the DP CLI is to be run by a non-install user, the non-install user must get additional permissions in one of two ways: either broadening permissions to HDFS or by impersonating the HDFS user.

- To broaden permissions for a given `edpDataDir=/user/bdd/edp/data`, you can grant everyone Write access:

```
hadoop fs -chmod -R 777 /user/bdd/edp/data
```

- Impersonating the install user does not require any permissions changes, and can be done by multiple different users. For example, while logged in as local user `j_jones`, you can use HDFS's impersonation feature to act as HDFS user `bdd`:

```
j_jones@node1:data-processing/edp_cli $
HADOOP_USER_NAME=bdd ./data_processing_CLI -t myTable
```

This is the same mechanism as used on the hadoop fs command line:

```
HADOOP_USER_NAME=hdfs hadoop fs -chmod -R 777 /user/bdd/edp/data
```

DP CLI logging

The DP CLI logs detailed information about its workflow into the log file defined in the `log4j.properties` file. This file is located in the `$BDD_HOME/dataprocessing/edp_cli/config` directory.

The implementation of the BDD Hive Table Detector is based on the DP CLI, so it uses the same logging properties as the DP CLI script. It also produces verbose outputs (on some classes) to stdout/stderr.

DP CLI configuration

The DP CLI has a configuration file, `edp-cli.properties`, that sets some Data Processing properties for Provisioning and Refresh update workflows.

The `edp-cli.properties` file is located in the `$BDD_HOME/dataprocessing/edp_cli/config` directory. Some of the default values for the properties are populated from the `bdd.conf` installation configuration file. After installation, you can change the CLI configuration parameters by opening the `edp-cli.properties` file with a text editor.

> **Note:** The Workflow Manager has a similar configuration file, named `edp.properties`, that is also used for all workflows launched by the DP CLI. For details on that file, see *DP workflow properties on page 30*.

Workflows that use edp-cli.properties

The following properties in the `edp-cli.properties` file are used by Provisioning and Refresh workflows:

- Provisioning workflows:
    - `maxRecordsForNewDataSet`
    - `runEnrichment`
    - `defaultLanguage`
    - `datasetAccessType`
- Refresh update workflows:
    - `maxRecordsForNewDataSet`
    - `datasetAccessType`

These workflows use a combination of the above properties from the `edp-cli.properties` file and the rest of the properties from the Workflow Manager's `edp.properties` file. Therefore, you can change the `edp.properties` file for other properties used by these workflows (such as Kerberos properties) and also for properties used by other types of workflows (such as Incremental update workflows).

Data Processing properties for Provisioning and Refresh workflows

These properties can be changed to affect the creation of Provisioning and Refresh data sets.

Data Processing Property	Description
maxRecordsForNewDataSet	Specifies the maximum number of records in the sample size of a new data set (that is, the number of sampled records from the source Hive table). In effect, this sets the maximum number of records in a BDD data set. Note that this setting controls the sample size for all new data sets and it also controls the sample size resulting from transform operations (such as during a Refresh update on a data set that contains a transformation script). The default is set by the MAX_RECORDS property in the bdd.conf file. The CLI --maxRecords flag can override this setting.
runEnrichment	Specifies whether to run the Data Enrichment modules. The default is set by the ENABLE_ENRICHMENTS property in the bdd.conf file. You can override this setting by using the CLI --runEnrichment flag. The CLI --excludePlugins flag can also be used to exclude some of the Data Enrichment modules.
defaultLanguage	Sets the language for all attributes in the created data set. The default is set by the LANGUAGE property in the bdd.conf file. For the supported language codes, see *Supported languages on page 105*.
datasetAccessType	Sets the access type for the data set, which determines which Studio users can access the data set in the Studio UI. This property takes one of these case-insensitive values: • public means that all Studio users can access the data set. This is the default. • private means that only designated Studio users and groups can access the data set. The users and groups are specified in attributes set in the data set's entry in the DataSet Inventory.

DP CLI flags

The DP CLI has a number of runtime flags that control its behavior.

You can list these flags if you use the --help flag. Most flags have a full name that begins with two dashes (such as --maxRecords) and an abbreviated version with one dash (such as -m). Some flags (such as get-active-jobs) do not use dashes as part of the flag name.

The --devHelp flag displays flags that are intended for use by Oracle internal developers and support personnel. These flags are therefore not documented in this guide.

Note: All flag names are case sensitive.

The CLI flags are:

CLI Flag	Description
-a, --all	Runs data processing on all Hive tables in all Hive databases.
-bl, --blackList *<blFile>*	Specifies the file name for the blacklist used to filter out Hive tables. The tables in this list are ignored and not provisioned. Must be used with the --database flag.
-d, --database *<dbName>*	Runs Data Processing using the specified Hive database. If this flag is not specified, the default Hive database is used. If a Hive table is not specified, runs on all Hive tables in the Hive database (note that tables with the skipAutoProvisioning property set to true will not be provisioned). For Refresh and Incremental updates, can be used to override the default database in the data set's metadata.
-devHelp, --devHelp	Displays usage information for flags intended to be used by Oracle support personnel.
-disableSearch, --disableSearch	Turns off Dgraph indexing for search. This means that DP Discovery disables record search and value search on all the attributes, irrespective of the average String length of the values. This flag can be used only for provisioning workflows (for new data sets created from Hive tables) and for refresh workflows (with the --refreshData flag). This flag cannot be used in conjunction with the --incrementalUpdate flag.
-e, --runEnrichment	Runs the Data Enrichment modules (except for the modules that never automatically run during the sampling phase). Overrides the runEnrichment property in the edp-cli.properties configuration file. You can also exclude some modules with the CLI --excludePlugins flag.
-ep, --excludePlugins *<exList>*	Specifies a list of Data Enrichment modules to exclude when Data Enrichments are run.

CLI Flag	Description
`-h, --help`	Displays usage information for flags intended to be used by customers.
`-health, --healthCheck`	Runs a health check on the components used by Data Processing.
`-incremental, --incrementalUpdate` `<logicalName> <filter>`	Performs an incremental update on a BDD data set from the original Hive table, using a filter predicate to select the new records. Optionally, can use the `--table` and `--database` flags.
`-m, --maxRecords <num>`	Specifies the maximum number of records in the sample size of a data set (that is, the number of sampled records from the source Hive table). In effect, this sets the maximum number of records in a BDD data set. Note that this setting controls the sample size for all new data sets and it also controls the sample size resulting from transform operations (such as during a Refresh update on a data set that contains a transformation script). Overrides the CLI `maxRecordsForNewDataSet` property in the `edp-cli.properties` configuration file.
`-mwt, --maxWaitTime <secs>`	Specifies the maximum waiting time (in seconds) for each table processing to complete. The next table is processed after this interval or as soon as the data ingesting is completed. This flag controls the pace of the table processing, and prevents Hadoop and Spark cluster nodes, as well as the Dgraph cluster nodes from being flooded with a large number of simultaneous requests.
`-ping, --pingCheck`	Ping checks the status of components that Data Processing needs.
`-refresh, --refreshData <logicalName>`	Performs a full data refresh on a BDD data set from the original Hive table. Optionally, you can use the `--table` and `--database` flags.

CLI Flag	Description
`-t, --table <tableName>`	Runs data processing on the specified Hive table. If a Hive database is not specified, assumes the default database. Note that the table is skipped in these cases: it does not exist, is empty, or has the table property `skipAutoProvisioning` set to `true`. For Refresh and Incremental updates, can be used to override the default source table in the data set's metadata.
`-v, --versionNumber`	Prints the version number of the current iteration of the Data Processing component within Big Data Discovery.
`-wl, --whiteList <wlFile>`	Specifies the file name for the whitelist used to select qualified Hive tables for processing. Each table on this list is processed by the Data Processing component and is ingested into the Dgraph as a BDD data set. Must be used with the `--database` flag.
`get-active-jobs`	Returns a list of all active Data Processing jobs, including their job IDs.
`get-job-status <jobId>`	Returns the status of a Data Processing job specified by its job ID. The job can be active (currently running) or inactive (finished, either successfully or not).
`cancel-job <jobId>`	Cancels an active Data Processing job specified by its job ID.
`UpgradeDatesetInventory <fromVersion>`	Upgrades the DataSet Inventory from a given BDD version to the latest version. Note that this subcommand is called by the upgrade script and should not be run interactively.
`UpgradeSampleFiles <fromVersion>`	Upgrades the sample files (produced as a result of a previous workflow) from a given BDD version to the latest version. Note that this subcommand is called by the upgrade script and should not be run interactively.

Using whitelists and blacklists

A whitelist specifies which Hive tables should be processed in Big Data Discovery, while a blacklist specifies which Hive tables should be ignored during data processing.

Default lists are provided in the DP CLI package:

- `cli_whitelist.txt` is the default whitelist name. The default whitelist is empty, as it does not select any Hive tables.

- `cli_blacklist.txt` is the default blacklist name. The default blacklist has one **.+** regex which matches all Hive table names (therefore all Hive tables are blacklisted and will not be imported).

Both files include commented-out samples of regular expressions that you can use as patterns for your tables.

To specify the whitelist, use this syntax:

```
--whiteList cli_whitelist.txt
```

To specify the blacklist, use this syntax:

```
--blackList cli_blacklist.txt
```

Both lists are optional when running the DP CLI. However, you use the `--database` flag if you want to use one or both of the lists.

If you manually run the DP CLI with the `--table` flag to process a specific table, the whitelist and blacklist validations will not be applied.

List syntax

The `--whiteList` and the `--blackList` flags take a corresponding text file as their argument. Each text file contains one or more regular expressions (regex). There should be one line per regex pattern in the file. The patterns are only used to match Hive table names (that is, the match is successful as long as there is one matched pattern found).

The default whitelist and blacklist contain commented-out sample regular expressions that you can use as patterns for your tables. You must edit the whitelist file to include at least one regular expression that specifies the tables to be ingested. The blacklist by default excludes all tables with the **.+** regex, which means you have to edit the blacklist if you want to exclude only specific tables.

For example, suppose you wanted to process any table whose name started with `bdd`, such as `bdd_sales`. The whitelist would have this regex entry:

```
^
bdd.*
```

You could then run the DP CLI with the whitelist, and not specify the blacklist.

List processing

The pattern matcher in Data Processing workflow uses this algorithm:

1. The whitelist is parsed first. If the whitelist is not empty, then a list of Hive tables to process is generated. If the whitelist is empty, then no Hive tables are ingested.

2. If the blacklist is present, the blacklist pattern matching is performed. Otherwise, blacklist matching is ignored.

To summarize, the whitelist is parsed first, which generates a list of Hive tables to process, and the blacklist is parsed second, which generates a list of skipped Hive table names. Typically, the names from the blacklist names modify those generated by the whitelist. If the same name appears in both lists, then that table is not processed, that is, the blacklist can, in effect, remove names from the whitelist.

Example

To illustrate how these lists work, assume that you have 10 Hive tables with sales-related information. Those 10 tables have a _bdd suffix in their names, such as `claims_bdd`. To include them in data processing, you create a `whitelist.txt` file with this regex entry:

```
^
.*_bdd$
```

If you then want to process all *_bdd tables except for the `claims_bdd` table, you create a `blacklist.txt` file with this entry:

```
claims_bdd
```

When you run the DP CLI with both the `--whiteList` and `--blackList` flags, all the *_bdd tables will be processed except for the `claims_bdd` table.

DP CLI cron job

You can specify that the BDD installer create a cron job to run the DP CLI.

By default, the BDD installer does not create a cron job for the DP CLI. To create the cron job, set the `ENABLE_HIVE_TABLE_DETECTOR` parameter to `TRUE` in the BDD installer's `bdd.conf` configuration file.

The following parameters in the `bdd.conf` configuration file control the creation of the cron job:

Configuration Parameter	Description
ENABLE_HIVE_TABLE_DETECTOR	When set to `TRUE`, creates a cron job, which automatically runs on the server defined by `DETECTOR_SERVER`. The default is `FALSE`.
DETECTOR_SERVER	Specifies the server on which the DP CLI will run.
DETECTOR_HIVE_DATABASE	The name of the Hive database that the DP CLI will run against.
DETECTOR_MAXIMUM_WAIT_TIME	The maximum amount of time (in seconds) that the Hive Table Detector waits between update jobs.
DETECTOR_SCHEDULE	A Cron format schedule that specifies how often the DP CLI runs. The default value is: `0 0 * * *` The default means the Hive Table Detector runs at midnight, every day of every month.

If the cron job is created, the default cron job definition settings (as set in the `crontab` file) are as follows:

```
0 0 * * * /usr/bin/flock -x -w 120 /localdisk/Oracle/Middleware/BDD/dataprocessing/edp_cli/work
/detector.lock
  -c "cd /localdisk/Oracle/Middleware/BDD/dataprocessing/edp_cli && ./data_processing_CLI
  -d default
  -wl /localdisk/Oracle/Middleware/BDD/dataprocessing/edp_cli/config/cli_whitelist.txt
  -bl /localdisk/Oracle/Middleware/BDD/dataprocessing/edp_cli/config/cli_blacklist.txt
  -mwt 1800 >> /localdisk/Oracle/Middleware/BDD/dataprocessing/edp_cli/work/detector.log 2>&1"
```

You can modify these settings, as shown in the following topic. In addition, be sure to monitor the size of the
`detector.log` file.

Modifying the DP CLI cron job

Modifying the DP CLI cron job

You can modify the crontab file to change settings for the cron job.

Some common changes include:

- Changing the schedule when the cron job is run.

- Changing which Hive database the DP CLI will run against. To do so, change the argument of the **-d** flag
  to specify another Hive database, such as **-d mytables** to process tables in the database named
  "mytables".

- Changing the amount of time (in seconds) that the Hive Table Detector waits between update jobs. To do
  so, change the argument of the **-mwt** flag to specify another time interval, such as **-mwt 2400** for an
  interval of 2400 seconds.

To modify the DP CLI cron job:

1. From the Linux command line, run the `crontab -e` command.

    The crontab file is opened in a text editor, such as the `vi` editor.

2. Make your changes to the crontab file in the editor.

3. Save the file.

The modified file may look like this:

```
30 4 * * * /usr/bin/flock -x -w 120 /localdisk/Oracle/Middleware/BDD/dataprocessing/edp_cli/work
/detector.lock
  -c "cd /localdisk/Oracle/Middleware/BDD/dataprocessing/edp_cli && ./data_processing_CLI
  -d mytables
  -wl /localdisk/Oracle/Middleware/BDD/dataprocessing/edp_cli/config/cli_whitelist.txt
  -bl /localdisk/Oracle/Middleware/BDD/dataprocessing/edp_cli/config/cli_blacklist.txt
  -mwt 2400 >> /localdisk/Oracle/Middleware/BDD/dataprocessing/edp_cli/work/detector.log 2>&1"
```

For the first few runs of the cron job, check the `detector.log` log file to verify that the cron jobs are running
satisfactorily.

DP CLI workflow examples

This topic shows some workflow examples using the DP CLI.

Excluding specific Data Enrichment modules

The `--excludePlugins` flag (abbreviated as `-ep`) specifies a list of Data Enrichment modules to exclude when enrichments are run. This flag should be used only enrichments are being run as part of the workflows (for example, with the `--excludePlugins` flag).

The syntax is:

```
./data_processing_CLI --excludePlugins <excludeList>
```

where *excludeList* is a space-separated string of one or more of these Data Enrichment canonical module names:

- `address_geo_tagger` (for the Address GeoTagger)
- `ip_geo_extractor` (for the IP Address GeoTagger)
- `reverse_geo_tagger` (for the Reverse GeoTagger)
- `tfidf_term_extractor` (for the TF.IDF Term extractor)
- `doc_level_sentiment_analysis` (for the document-level Sentiment Analysis module)
- `language_detection` (for the Language Detection module)

For example:

```
./data_processing_CLI --table masstowns --runEnrichment --excludePlugins reverse_geo_tagger
```

For details on the Data Enrichment modules, see *Data Enrichment Modules on page 85*.

Ping checking the DP components

The `--pingCheck` flag (abbreviated as `-ping`) ping checks the connection status of the components that Data Processing needs:

```
./data_processing_CLI --pingCheck
```

A successful result should be similar to this example:

```
Ping ok
data_processing_CLI finished with state SUCCESS
```

Running a DP health check

The `--healthCheck` flag (abbreviated as `-health`) returns the status of the components that are required by Data Processing:

```
./data_processing_CLI --healthCheck
```

A successful result should be similar to this example:

```
...
[2015-07-14T14:52:32.270-04:00] [DataProcessing] [INFO] []
[com.oracle.endeca.pdi.logging.ProvisioningLogger]
[tid:main] [userID:fcalvill] Ping check time elapsed: 7 ms
data_processing_CLI finished with state SUCCESS
```

A successful result should be similar to this example:

```
health=ok, v1="db:ok; yarn:ok; hdfs:ok"
data_processing_CLI finished with state SUCCESS
```

These status fields will report either `ok` (if a successful connection was made to the component) or `notok` (if the connection attempt to the component was unsuccessful):

- `db` is the connection to the Workflow Manager database.

- `yarn` is the connection to the YARN service.

- `hdfs` is the connection to HDFS.

- `auth` is the connection to a secure cluster. This field is displayed only if the cluster is configured as a secure cluster.

- `mode:shutdown` is reported only when Workflow Manager is in shutdown mode at the present time.

- `health` is the overall status of Data Processing. This status is `ok` if all the above connection statuses are `ok`.

Getting job status

The `get-job-status` flag returns the status of a completed or active Data Processing job:

```
./data_processing_CLI get-job-status <jobId>
```

When a Data Processing workflow is started from the DP CLI, its job ID is displayed, as in this example:

```
./data_processing_CLI -d default -t warrantyclaims
...
New collection name = MdexCollectionIdentifier{databaseName
=edp_cli_edp_997d2151-b694-4bd2-88be-732461731b6c, collectionName
=edp_cli_edp_997d2151-b694-4bd2-88be-732461731b6c}
jobId: 950b2d4a-20cf-4e9d-9f5f-5cf713ade145
```

Note that jobs started from Studio do not display a job ID, which means you cannot get their status.

You can then use that job ID to get its status:

```
./data_processing_CLI get-job-status 950b2d4a-20cf-4e9d-9f5f-5cf713ade145
```

The status for the job will be one of the following:

- `NOT_STARTED`

- `RUNNING`

- `SUCCEEDED`

- `NOTFOUND`

- `ABORTING`

- `ABORTED`

- `CANCELLING`

- `CANCELLED`

- `FAILED`

The following example shows the status of a job that has failed:

```
./data_processing_CLI get-job-status 950b2d4a-20cf-4e9d-9f5f-5cf713ade145
```

```
Job status for job id: 950b2d4a-20cf-4e9d-9f5f-5cf713ade145 is FAILED
```

Cancelling jobs

The `cancel-job` flag cancels an active Data Processing job:

```
./data_processing_CLI cancel-job <jobId>
```

The result output should be one of the following:

- Case 1: output when the job was not found:

  ```
  Could not cancel job with id: 123 because it could not be found.
  ```

- Case 2: output when the job succeeded already:

  ```
  Could not cancel job with id: 123 because it has already succeeded.
  ```

- Case 3: output when the job failed already:

  ```
  Could not cancel job with id: 123 because it has already failed.
  ```

- Case 4: output when the operation results in an error:

  ```
  Could not cancel job with id: 123 because of error: <error_message>
  ```

- Case 5: output when the job was successfully cancelled:

  ```
  Job status for job id: 123 is CANCELLED
  ```

Note that the output in Case 5 will be the same as a `get-job-status` command.

Processing Hive tables with Snappy compression

This topic explains how to set up the Snappy libraries so that the DP CLI can process Hive tables with Snappy compression.

By default, the DP CLI cannot successfully process Hive tables with Snappy compression. The reason is that the required Hadoop native libraries are not available in the library path of the JVM. Therefore, you must the Hadoop native libraries' path to the Workflow Manager's `sparkContext.properties` file, which is located in the `$BDD_HOME/workflowmanager/dp/config` directory. For information on this configuration file, see .

To configure workflows to use the Snappy libraries:

1. Locate the source directory for the Hadoop native libraries in your Hadoop installation. Typical locations are:
    - CDH:

      ```
      /opt/cloudera/parcels/CDH/lib/hadoop/lib/native/
      ```

    - MapR:

      ```
      /opt/mapr/hadoop/hadoop-2.7.0/lib/native
      ```

    - HDP:

      ```
      /usr/hdp/2.4.2.0-212/hadoop/lib/native
      ```

2. To make the native libraries available to Spark, the native libraries need to be added to the Workflow Manager's `sparkContext.properties` file, using these Spark properties:

```
spark.executorEnv.LD_LIBRARY_PATH=<snappypath>
spark.yarn.appMasterEnv.LD_LIBRARY_PATH=<snappypath>
```

CDH example:

```
spark.executorEnv.LD_LIBRARY_PATH=/opt/cloudera/parcels/CDH/lib/hadoop/lib/native
spark.yarn.appMasterEnv.LD_LIBRARY_PATH=/opt/cloudera/parcels/CDH/lib/hadoop/lib/native
```

Once the paths are added to the Workflow Manager's properties file, all subsequent DP workflows should be able to process Hive tables with Snappy compression.

Changing Hive table properties

This topic describes how to change the value of the `skipAutoProvisioning` property in a Hive table.

When a Hive table has a `skipAutoProvisioning` property set to `true`, the BDD Hive Table Detector will skip the table for data processing. For details, see .

You can change the value of `skipAutoProvisioning` property by issuing an SQL `ALTER TABLE` statement via the Cloudera Manager's Query Editor or as a Hive command.

To change the value of the `skipAutoProvisioning` property in a Hive table:

1. From the Cloudera Manager home page, click **Hue**.

2. From the Hue home page, click **Hue Web UI**.

3. From the Hue Web UI page, click **Metastore Manager**. As a result, you should see your Hive tables in the default database, as in this example:
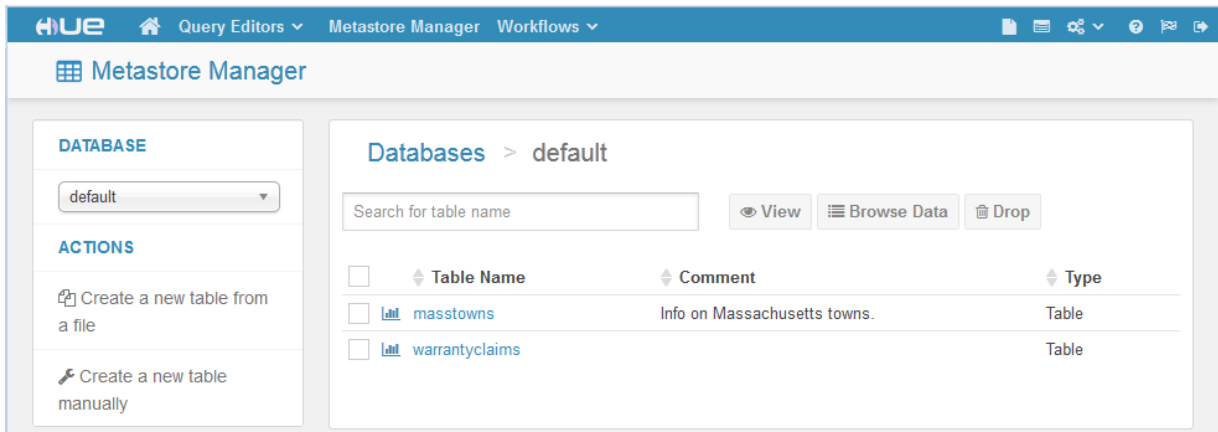
4. Verify that the table has the `skipAutoProvisioning` property:

   (a) Select the table you want to change and click **View**. The default **Columns** tab shows the table's columns.

   (b) Click the **Properties** tab.

   (c) In the **Table Parameters** section, locate the `skipAutoProvisioning` property and (if it exists) verify that its value is set to "true".

5. From the Metastore Manager page, click **Query Editors** and then **Hive**.

   The Query Editor page is displayed.

6. In the Query Editor, enter an `ALTER TABLE` statement similar to the following example (which is altering the warrantyclaims table) and click **Execute**.

7. From the Metastore Manager page, repeat Step 4 to verify that the value of the `skipAutoProvisioning` property has been changed..

An alternative to using the UI is to issue the `ALTER TABLE` statement as a Hive command:

```
hive -e "ALTER TABLE warrantyclaims SET TBLPROPERTIES('skipAutoProvisioning'='FALSE');"
```

Chapter 5

Updating Data Sets

This section describes how to run update operations on BDD data sets.

About data set updates

You can update data sets by running Refresh updates and Incremental updates with the DP CLI.

When first created, a BDD data set may be sampled, which means that the BDD data set has fewer records than its source Hive table. In addition, more records can be added to the source Hive table, and these new records will not be added to the data set by default.

Two DP CLI operations are available that enable the BDD administrator to synchronize a data set with its source Hive table:

- The `--refreshData` flag (abbreviated as `-refresh`) performs a full data refresh on a BDD data set from the original Hive table. This means that the data set will have all records from the source Hive table. If the data set had previously been sampled, it will now be a full data set. And as records get added to the Hive table, the Refresh update operation can keep the data set synchronized with its source Hive table.

- The `--incrementalUpdate` flag (abbreviated as `-incremental`) performs an incremental update on a BDD data set from the original Hive table, using a filter predicate to select the new records. Note that this operation can be run only after the data set has been configured for Incremental updates.

Note that the equivalent of a DP CLI Refresh update can done in Studio via the **Load Full Data Set** feature. However, Incremental Data updates can be performed only via the DP CLI, as Studio does not support this feature.

Re-pointing a data set

if you created a data set by uploading source data into Studio and want to run Refresh and Incremental updates, you should change the source data set to point to a new Hive table. (Note that this change is not required if the data set is based on a table created directly in Hive.) For information on this re-pointing operation, see the topic on converting a project to a BDD application in the *Studio User's Guide*.
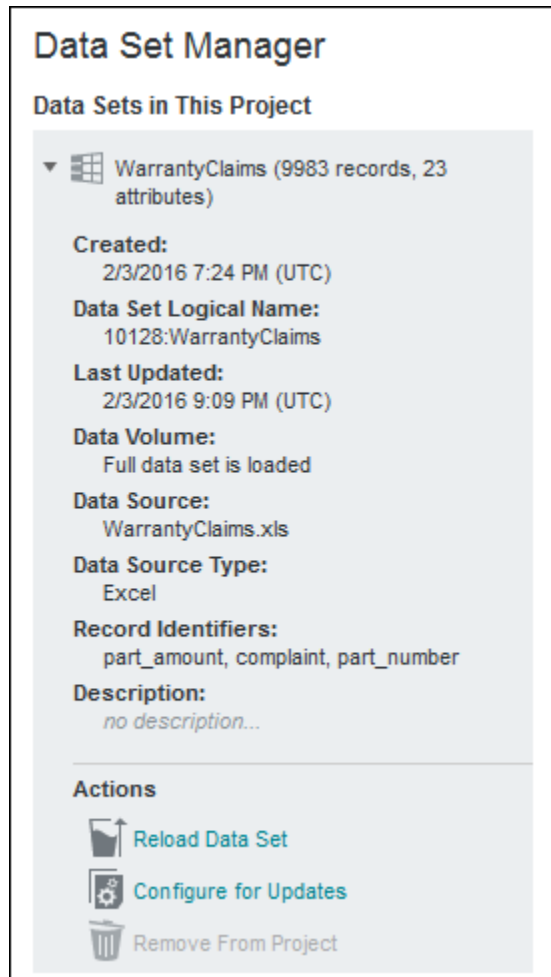
Obtaining the Data Set Logical Name

The Data Set Logical Name specifies the data set to be updated.

The Data Set Logical Name is needed as an argument to the DP CLI flags for the Refresh and Incremental update operations.

You can obtain the Data Set Logical Name in Studio from the **Data Set Manager** page in **Project Settings**.

The **Data Set Manager** page looks like this cropped example for the WarrantyClaims data set:

The **Data Set Logical Name** field lists the logical name of the data set. In this example, **10128:WarrantyClaims** is the Data Set Logical Name of this particular data set.

Refresh updates

A Refresh update replaces the schema and all the records in a project data set with the schema and records in the source Hive table.

The DP CLI `--refreshData` flag (abbreviated as `-refresh`) performs a full data refresh on a BDD data set from the original Hive table. The data set should be a project data set (that is, must added to a Studio project).

Loading the full data set affects only the data set in a specific project; it does not affect the data set as it displays in the Studio Catalog.

Running a Refresh update produces the following results:

- All records stored in the Hive table are loaded for that data set. This includes any table updates performed by a Hive administrator.

- If the data set was sampled, it is increased to the full size of the data set. That is, it is now a full data set.

- If the data set contains a transformation script, that script will be run against the full data set, so that all transformations apply to the full data set in the project.

- If the `--disableSearch` flag is also used, record search and value search will be disabled for the data set.

The equivalent of a DP CLI Refresh update can be done in Studio via the **Load Full Data Set** feature (although you cannot specify a different source table as with the DP CLI).

Note that you should not start a DP CLI Refresh update if a transformation on that data set is in progress. In this scenario, the Refresh update will fail and a notification will be sent to Studio:

```
Reload of <logical name> from CLI has failed. Please contact an administrator.
```

Schema changes

There are no restrictions on how the schema of the data set is changed due to changes in the schema and/or data of the source Hive table. This non-restriction is because the Refresh update operation uses a kill-and-fill strategy, in which the entire contents of the data set are removed and replaced with those in the Hive table.

Transformation scripts in Refresh updates

If the data set has an associated Transformation script, then the script will run against the newly-ingested attributes and data. However, some of the schema changes may prevent some of the steps of the script from running. For example:

- Existing columns in Hive table may be deleted. As a result, any Transformation script step that references the deleted attributes will be skipped.

- New columns can be added to the Hive table and they will result in new attributes in the data set. The Transformation script will not run on these new attributes as the script would not reference them.

- Added data to a Hive column may result in the attribute having a different data type (such as String instead of a previous Long). The Transformation script may or may not run on the changed attribute.

The following diagram illustrates the effects of a schema change on the Transformation script:
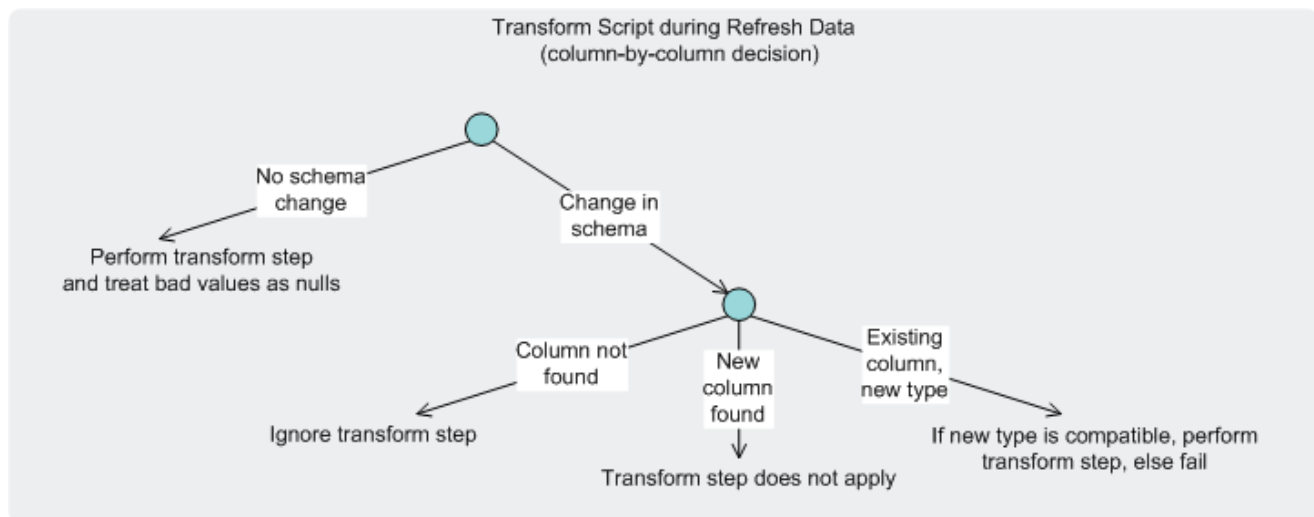
If the data set does not have an associated Transformation script and the Hive table schema has changed, then the data set is updated with the new schema and data.

Refresh flag syntax

Running a Refresh update

Refresh flag syntax

This topic describes the syntax of the `--refreshData` flag.

The DP CLI flag syntax for a Refresh update operation has one of the following syntaxes:

```
./data_processing_CLI --refreshData <logicalName>
```

or

```
./data_processing_CLI --refreshData <logicalName> --table <tableName>
```

or

```
./data_processing_CLI --refreshData <logicalName> --table <tableName> --database <dbName>
```

where:

- `--refreshData` (abbreviated as `-refresh`) is mandatory and specifies the logical name of the data set to be updated.

- `--table` (abbreviated as `-t`) is optional and specifies a Hive table to be used for the source data. This flag allows you to override the source Hive table that was used to create the original data set (the name of the original Hive table is stored in the data set's metadata).

- `--database` (abbreviated as `-d`) is optional and specifies the database of the Hive table specified with the `--table` flag. This flag allows you to override the database that was used to create the original data set). The `--database` flag can be used only if the `--table` flag is also used.

The *logicalName* value is available in the **Data Set Logical Name** property in Studio. For details, see *Obtaining the Data Set Logical Name on page 60*.

Use of the --table and --database flags

When a data set is first created, the names of the source Hive table and the source Hive database are stored in the DSI (DataSet Inventory) metadata for that data set. The `--table` flag allows you to override the default source Hive table, while the `--database` flag can override the database set in the data set's metadata.

Note that these two flags are ephemeral. That is, they are used only for the specific run of the operation and do not update the metadata of the data set.

If these flags are not specified, then the Hive table and Hive database that are used are the ones in the data set's metadata.

Use these flags when you want to temporarily replace the data in a data set with that from another Hive table. If the data change is permanent, it is recommended that you create a new data set from desired Hive table. This will also allow you to create a Transformation script that is exactly tailored to the new data set.

Running a Refresh update

This topic describes how to run a Refresh update operation.

This procedure assumes that:

* The data set has been created, either from Studio or with the DP CLI.

* The data set has been added to a Studio project.

To run a Refresh update on a data set:

1. Obtain the Data Set Logical Name of the data set you want to refresh:

    (a) In Studio, go to **Project Settings** and then **Data Set Manager**.

    (b) In the **Data Set Manager**, select the data set and expand the options next to its name.

    (c) Get the value from the **Data Set Logical Name** field.

2. From a Linux command prompt, change to the `$BDD_HOME/dataprocessing/edp_cli` directory.

3. Run the DP CLI with the `--refreshData` flag and the Data Set Logical Name. For example:

    ```
    ./data_processing_CLI --refreshData 10128:WarrantyClaims
    ```

If the workflow was successfully invoked, the DP CLI prints these messages at the end of the stdout output:

```
...
Refreshing existing collection: MdexCollectionIdentifier{
   databaseName=edp_cli_edp_db627d25-fe2f-4117-a068-b742bb699650,
   collectionName=edp_cli_edp_db627d25-fe2f-4117-a068-b742bb699650}
Collection key for new record:  MdexCollectionIdentifier{
   databaseName=refreshed_edp_9279101f-6c77-40f2-ab6d-5e4332a0be1c,
   collectionName=refreshed_edp_9279101f-6c77-40f2-ab6d-5e4332a0be1c}
   jobId: 06ea3ebd-1b31-4a74-84d1-db2dfb6299cd
data_processing_CLI finished with state SUCCESS
```

If the Refresh update operation successfully finished, the YARN Application Overview page should have a **State** of "FINISHED" and a **FinalStatus** of "SUCCEEDED". The **Name** field will have an entry similar to this example:

```
EDP: DatasetRefreshConfig{hiveDatabase=, hiveTable=,
collectionToRefresh=MdexCollectionIdentifier{
    databaseName=edp_cli_edp_db627d25-fe2f-4117-a068-b742bb699650,
    collectionName=edp_cli_edp_db627d25-fe2f-4117-a068-b742bb699650},
newCollectionId=MdexCollectionIdentifier{
```

```
    databaseName=refreshed_edp_9279101f-6c77-40f2-ab6d-5e4332a0be1c,
    collectionName=refreshed_edp_9279101f-6c77-40f2-ab6d-5e4332a0be1c},
    op=REFRESH_DATASET, jobId=06ea3ebd-1b31-4a74-84d1-db2dfb6299cd}
```

Note the following about the **Name** information:

- `hiveDatabase` and `hiveTable` are blank because the `--database` and `--table` flags were not used. In this case, the Refresh update operation uses the same Hive table and database that were used when the data set was first created.

- `collectionToRefresh` is name of the data set that was refreshed. This name is the same as the `Refreshing existing collection` field in the stdout listed above.

- `newCollectionId` is an internal name for the refreshed data set. This name will not appear in the Studio UI (the original Data Set Logical Name will continue to be used as it is a persistent name). This name is also the same as the `Collection key for new record` field in the stdout listed above.

You can also check the Dgraph HDFS Agent log for the status of the Dgraph ingest operation.

Note that future Refresh updates on this data set will continue to use the same Data Set Logical Name. You will also use this name if you set up a Refresh update cron job for this data set.

Incremental updates

An Incremental update adds new records to a project data set from a source Hive table.

The DP CLI `--incrementalUpdate` flag (abbreviated as `-incremental`) performs a partial update of a project data set by selecting adding new and modified records. The data set should be a project data set that is a full data set (i.e., is not a sample data set) and has been configured for incremental updates.

The Incremental update operation fetches a subset of the records in the source Hive table. The subset is determined by using a filtering predicate that specifies the Hive table column that holds the records and the value of the records to fetch. The records in the subset batch are ingested as follows:

- If a record is brand new (does not exist in the data set), it is added to the data set.

- If a record already exists in the data set but its content has been changed, it replaces the record in the data set.

The record identifier determines if a record already exists or is new.

Schema changes and disabling search

Unlike a Refresh update, an Incremental update has these limitations:

- An Incremental update cannot make schema changes to the data set. This means that no attributes in the data set will be deleted or added.

- An Incremental update cannot use the `--disableSearch` flag. This means that the searchability of the data set cannot be changed.

Transformation scripts in Incremental updates
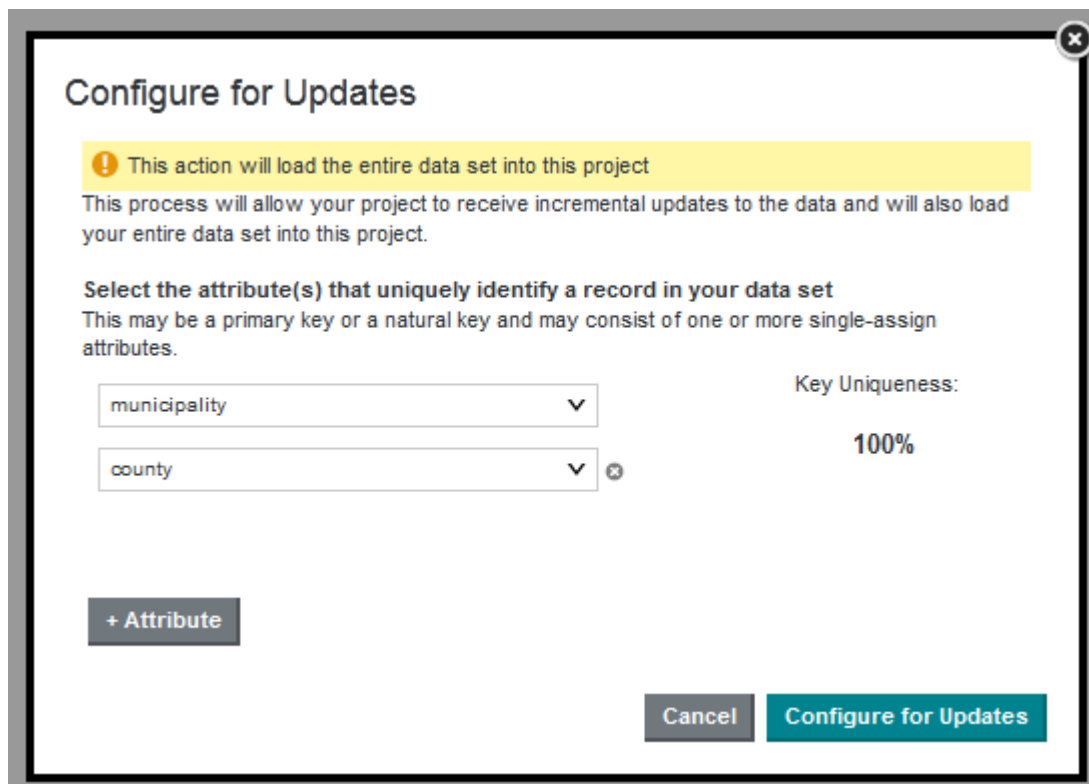
If the data set has an associated Transformation script, then the script will run against the new records and can transform them (if a transform step applies). Existing records in the data set are not affected.

Record identifier configuration

A data set must be configured for Incremental updates before you can run an Incremental update against it. This procedure must be done in Studio from the **Data Set Manager** page (under **Project Settings**).

The data set must be configured with a record identifier for determining the delta between records in the Hive table and records in the project data set. If columns have been added or removed from the Hive table, you should run a Refresh update to incorporate those column changes in the data set.

When selecting the attributes that uniquely identify a record, the uniqueness score must be 100%. If the record identifier is not 100% unique, the Data Processing workflow will fail and return an exception. In this example, the **Key Uniqueness** field shows a 100% figure:

After the data set is configured, its entry in the **Data Set Manager** page looks like this example:

Data Set Manager

Data Sets in This Project

▼ ⊞ masstowns (333 records, 6 attributes)

Created:
   2/2/2016 9:43 PM (UTC)

Data Set Logical Name:
   10129:masstowns

Last Updated:
   2/2/2016 9:43 PM (UTC)

Data Volume:
   Full data set is loaded

Data Source:
   default.masstowns

Data Source Type:
   Hive

Record Identifiers:
   municipality, county

Description:
   Info on Massachusetts towns.

─────────────────────

Actions

   ⬆ Reload Data Set

   ⚙ Configure for Updates

   🗑 Remove From Project

Note that the **Record Identifiers** field now lists the attributes that were selected in the **Configure for Updates** dialogue.

The Configure for Updates procedure is documented in the *Studio User's Guide*.

Error for non-configured data sets

If the data set has not been configured for Increment updates, the Incremental update fails with an error similar to this:

```
...
[2016-10-06T14:22:38.052-04:00] [DataProcessing] [WARN] []
  [com.oracle.endeca.pdi.client.EdpServiceClient] [tid:main] [userID:fcalvill]
  Exception in EdpServiceClient::invokeIncrementalUpdate
    com.oracle.bdd.services.workflowmanager.client.ApiException:
      {"type":null,"title":"submission not accepted.","status":500,"detail":null,
      "instance":null,"o:errorCode":null,"o:errorPath":null,"o:errorDetails":[]}
        at com.oracle.bdd.services.workflowmanager.client.ApiClient.invokeAPI(ApiClient.java:640)
        at
com.oracle.bdd.services.workflowmanager.api.WorkflowManagerApi.incrementalUpdate(WorkflowManagerApi.j
ava:535)
        at
com.oracle.endeca.pdi.client.EdpServiceClient.invokeIncrementalUpdate(EdpServiceClient.java:239)
        at com.oracle.endeca.pdi.EdpCli.runEdp(EdpCli.java:1103)
        at com.oracle.endeca.pdi.EdpCli.main(EdpCli.java:1225)
data_processing_CLI finished with state ERROR
Exception in thread "main" com.oracle.endeca.pdi.client.EdpExecutionException:
```

```
   EDP failed with exception: com.oracle.bdd.services.workflowmanager.client.ApiException:
      {"type":null,"title":"submission not accepted.","status":500,"detail":null,
      "instance":null,"o:errorCode":null,"o:errorPath":null,"o:errorDetails":[]}
         at
com.oracle.endeca.pdi.client.EdpServiceClient.invokeIncrementalUpdate(EdpServiceClient.java:242)
         at com.oracle.endeca.pdi.EdpCli.runEdp(EdpCli.java:1103)
         at com.oracle.endeca.pdi.EdpCli.main(EdpCli.java:1225)
Caused by: com.oracle.bdd.services.workflowmanager.client.ApiException:
      {"type":null,"title":"submission not accepted.","status":500,"detail":null,
      "instance":null,"o:errorCode":null,"o:errorPath":null,"o:errorDetails":[]}
         at com.oracle.bdd.services.workflowmanager.client.ApiClient.invokeAPI(ApiClient.java:640)
         at
com.oracle.bdd.services.workflowmanager.api.WorkflowManagerApi.incrementalUpdate(WorkflowManagerApi.j
ava:535)
         at
com.oracle.endeca.pdi.client.EdpServiceClient.invokeIncrementalUpdate(EdpServiceClient.java:239)
         ... 2 more
```

If this error occurs, configure the data set for Incremental updates and re-run the Incremental update operation.

[Incremental flag syntax](#)

[Running an Incremental update](#)

Incremental flag syntax

This topic describes the syntax of the `--incrementalUpdate` flag.

The DP CLI flag syntax for an Incremental update operation is one of the following:

```
./data_processing_CLI --incrementalUpdate <logicalName> <filter>
```

or

```
./data_processing_CLI --incrementalUpdate <logicalName> <filter> --table <tableName>
```

or

```
./data_processing_CLI --incrementalUpdate <logicalName> <filter> --table <tableName>
   --database <dbName>
```

where:

- `--incrementalUpdate` (abbreviated as `-incremental`) is mandatory and specifies the Data Set Logical Name (*logicalName*) of the data set to be updated. *filter* is a filter predicate that limits the records to be selected from the Hive table.

- `--table` (abbreviated as `-t`) is optional and specifies a Hive table to be used for the source data. This flag allows you to override the source Hive table that was used to create the original data set (the name of the original Hive table is stored in the data set's metadata).

- `--database` (abbreviated as `-d`) is optional and specifies the database of the Hive table specified with the `--table` flag. This flag allows you to override the database that was used to create the original data set). The `--database` flag can be used only if the `--table` flag is also used.

The *logicalName* value is available in the **Data Set Logical Name** property in Studio. For details, see *Obtaining the Data Set Logical Name on page 60*.

Filter predicate format

A filter predicate is mandatory and is one simple Boolean expression (not compounded), with this format:

```
"columnName operator filterValue"
```

where:

- `columnName` is the name of a column in the source Hive table.
- `operator` is one of the following comparison operators:
  - `=`
  - `<>`
  - `>`
  - `>=`
  - `<`
  - `<=`
- `filterValue` is a primitive value. Only primitive data types are supported, which are: integers (`TINYINT`, `SMALLINT`, `INT`, and `BIGINT`), floating point numbers (`FLOAT` and `DOUBLE`), Booleans (`BOOLEAN`), and strings (`STRING`). Note that expressions (such as "amount+1") are not supported.

You should enclose the entire filter predicate in either double quotes or single quotes. If you need to use quotes within the filter predicate, use the other quotation format. For example, if you use double quotes to enclose the filter predicate, then use single quotes within the predicate itself.

If `columnName` is configured as a `DATE` or `TIMESTAMP` data type, you can use the `unix_timestamp` date function, with one of these syntaxes:

```
columnName operator unix_timestamp(dateValue)

columnName operator unix_timestamp(dateValue, dateFormat)
```

If *dateFormat* is not specified, then the DP CLI uses one of two default data formats:

```
// date-time format:
yyyy-MM-dd HH:mm:ss

// time-only format:
HH:mm:ss
```

The date-time format is used for columns that map to Dgraph `mdex:dateTime` attributes, while the time-only format is used for columns that map to Dgraph `mdex:time` attributes.

If *dateFormat* is specified, use a pattern described here:
http://docs.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html

Note on data types in the filter predicate

You should pay close attention to the Hive column data types when constructing a filter for Incremental update, because the results of a comparison can differ. This is especially important for columns of type String, because results of String comparison are different from results of Number comparison.

Take, as an example, this filter that uses the "age" column in the Hive table:

```
./data_processing_CLI -incremental 10133:WarrantyClaims "age<18"
```

If the "age" column is a String column, then the results from the filter will be different than if "age" were a Number column (such as Int or Tinyint). The results would differ because:

- If "age" is a Number column, then "age < 18" means the column value must be numerically less than 18. The value 6, for example, is numerically less than 18.

- If "age" is a String column, then "age < 18" means the column value must be lexicographically less than 18. The value 6 is lexicographically more than 18.

Therefore, the number of filtered records will differ depending on the data type of the "age" column.

Also keep in mind that if the data set was originally created using File Upload in Studio, then the underlying Hive table for that data set will have all columns of type String.

Examples

Example 1: If the Hive "birthyear" column contains a year of birth for a person, then the command can be:

```
./data_processing_CLI --incrementalUpdate 10133:WarrantyClaims "claimyear > 1970"
```

In the example, only the records of claims made after 1970 are processed.

Example 2: Using the `unix_timestamp` function with a supplied date-time format:

```
./data_processing_CLI --incrementalUpdate 10133:WarrantyClaims
 "factsales_shipdatekey_date >= unix_timestamp('2006-01-01 00:00:00', 'yyy-MM-dd HH:mm:ss')"
```

Example 3: Another example of using the `unix_timestamp` function with a supplied date-time format:

```
./data_processing_CLI --incrementalUpdate 10133:WarrantyClaims
"creation_date >= unix_timestamp('2015-06-01 20:00:00', 'yyyy-MM-dd HH:mm:ss')"
```

Example 4: An invalid example of using the `unix_timestamp` function with a date that does not contain a time:

```
./data_processing_CLI --incrementalUpdate 10133:WarrantyClaims
"claim_date >= unix_timestamp('2000-01-01')"
```

The error will be:

```
16:41:29.375 main ERROR: Failed to parse date
/ time value '2000-01-01' using the format 'yyyy-MM-dd HH:mm:ss'
```

Running an Incremental update

This topic describes how to run an Incremental update operation.

This procedure assumes that the data set has been configured for Incremental updates (that is, a record identifier has been configured).

Note that the example in the procedure does not use the `--table` and `--database` flags, which means that the command will run against the original Hive table from which the data set was created.

To run an Incremental update on a data set:

1. Obtain the Data Set Logical Name of the data set you want to incrementally update:

   (a) In Studio, go to **Project Settings** and then **Data Set Manager**.

   (b) In the **Data Set Manager**, select the data set and expand the options next to its name.

   (c) Get the value from the **Data Set Logical Name** field.

2. From a Linux command prompt, change to the `$BDD_HOME/dataprocessing/edp_cli` directory.

3. Run the DP CLI with the `--incrementalUpdate` flag, the Data Set Logical Name, and the filter predicate. For example:

```
./data_processing_CLI --incrementalUpdate 10801:WarrantyClaims "claim_amount > 1000"
```

If the workflow was successfully invoked, the DP CLI prints these messages at the end of the stdout output:

```
...
jobId: b6a9fab0-7ca0-4d35-9950-a066520dd948
data_processing_CLI finished with state SUCCESS
```

The YARN Application Overview page should have a **State** of "FINISHED" and a **FinalStatus** of "SUCCEEDED". The **Name** field will have an entry similar to this example:

```
EDP: IncrementalUpdateConfig{collectionId=MdexCollectionIdentifier{
    databaseName=default_edp_e6bfc4c3-24cc-4141-96fe-53cb808b788d,
    collectionName=default_edp_e6bfc4c3-24cc-4141-96fe-53cb808b788d},
    jobId=b6a9fab0-7ca0-4d35-9950-a066520dd948,
    whereClause=claim_amount > 1000}
```

Note the following about the **Name** information:

- `IncrementalUpdateConfig` is the name of the type of Incremental workflow.
- `whereClause` lists the filter predicate used in the command.

You can also check the Dgraph HDFS Agent log for the status of the Dgraph ingest operation.

If the Incremental update determines that there are no records that fit the filter predicate criteria, a Dgraph ingest operation is not performed and the DP log will have this entry:

```
Nothing to be done here since no records are returned for the filter predicate.
```

Note that future Incremental updates on this data set will continue to use the same Data Set Logical Name. You will also use this name if you set up a Incremental update `cron` job for this data set.

Creating cron jobs for updates

You can create `cron` jobs to run your Refresh and Incremental updates.

You can use the Linux `crontab` command to create cron jobs for your Refresh and Incremental updates. A `cron` job will run the DP CLI (with one of the update flags) at a specific date and time.

The `crontab` file will have one or more `cron` jobs. Each job should take up a single line. The job command syntax is:

```
schedule path/to/command
```

The command begins with a five-field *schedule* of when the command will run. A simple version of the time fields in is:

```
minute hour dayOfMonth month dayOfWeek
```

where:

- `minute` is 0-59.
- `hour` is 0-23 (0 = midnight).
- `dayOfMonth` is 1-31 or * for every day of the month.

- `month` is 1-12 or * for every month.

- `dayOfWeek` is 0-6 (0 - Sunday) or * for every day of the week.

path/to/command is the path (including the command name) of the DP CLI update to run, including the appropriate flag and argument.

An example would be:

```
0 0 2 * * /localdisk/Oracle/Middleware/BDD/dataprocessing/edp_cli
/data_processing_CLI --refresh 10133:WarrantyClaims
```

The job would run every day at 2am.

To set up a cron job for updates:

1. From the Linux command line, use the `crontab` command with the `e` flag to open the `crontab` file for editing:

    `crontab -e`

2. Enter the job command line, as in the above example.

3. Save the file.

You can also use the Hive Table Detector `cron` job as a template, as it uses the Linux `flock` command and generates a log file. For details, see .

Chapter 6

Data Processing Logging

This section describes logging for the Data Processing component of Big Data Discovery.

DP logging overview

This topic provides an overview of the Data Processing logging files.

Location of the log files

Each run of Data Processing produces one or more log files on each machine that is involved in the Data Processing job. The log files are in these locations:

- On the Data Processing nodes, the location of the DP log files is set by the `log4j.appender.edpMain.Path` property in the `log4j.properties` configuration file in the `$BDD_HOME/workflowmanager/dp/config` directory. The default location of the file is the `/tmp/edp/log` directory. These DP log files apply to workflows initiated by both Studio and the DP CLI.

- On the client machine, the Workflow Manager workflow log files are written to the `$BDD_HOME/logs/workflowmanager` directory. These log files apply to workflows initiated by both Studio and the DP CLI.

- On the Hadoop nodes, logs are generated by the Spark processes. The contents of these logs are also returned in the DP log files.

Local log files

The Data Processing log files (in the `/tmp/edp/log` directory) are named `edpLog*.log`. The naming pattern is set in the `logging.properties` configuration.

The default naming pattern for each log file is

```
edp_%timestamp_%unique.log
```

where:

- `%timestamp` provides a timestamp in the format: yyyyMMddHHmmssSSS

- `%unique` provides a uniquified string

For example:

```
edp_20150728100110505_0bb9c1a2-ce73-4909-9de0-a10ec83bfd8b.log
```

The `log4j.appender.edpMain.MaxSegmentSize` property sets the maximum size of a log file, which is 100MB by default. Logs that reach the maximum size roll over to the next log file. The maximum amount of disk space used by the main log file and the logging rollover files is about 1GB by default.

DP logging properties file

Data Processing has a default Log4j configuration file that sets its logging properties.

The file is named `log4j.properties` and is located in the `$BDD_HOME/workflowmanager/dp/config` directory.

> **Note:** The DP CLI has a similar configuration file (also named `log4j.properties`) that is located in the `$BDD_HOME/dataprocessing/edp_cli/config` directory. If you are changing the DP logging properties, make sure you edit the file in the `$BDD_HOME/workflowmanager/dp/config` directory. Although the DP CLI version is used at the start of a DP workflow launched by the DP CLI, the Workflow Manager version is used by the Spark worker and thus controls the configuration of the DP output log.

Default DP logging properties file

The default DP logging properties file in the `$BDD_HOME/workflowmanager/dp/config` directory looks like this:

```
############################################################
# Global properties
############################################################

log4j.rootLogger = INFO, console, edpMain

############################################################
# Handler specific properties.
############################################################

log4j.appender.console = org.apache.log4j.ConsoleAppender

############################################################
# EdpODPFormatterAppender is a custom log4j appender that gives two new optional
# variables that can be added to the log4j.appender.*.Path property and are
# filled in at runtime:
# %timestamp  - provides a timestamp in the format: yyyyMMddHHmmssSSS
# %unique -  provides a uniquified string
############################################################

log4j.appender.edpMain = com.oracle.endeca.pdi.logging.EdpODLFormatterAppender
log4j.appender.edpMain.ComponentId = DataProcessing
log4j.appender.edpMain.Path = /tmp/edp/logs/edp_%timestamp_%unique.log
log4j.appender.edpMain.Format = ODL-Text
log4j.appender.edpMain.MaxSegmentSize = 100000000
log4j.appender.edpMain.MaxSize = 1000000000
log4j.appender.edpMain.Encoding = UTF-8

############################################################
# Formatter specific properties.
############################################################
```

```
log4j.appender.console.layout = org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern
= [%d{yyyy-MM-dd'T'HH:mm:ss.SSSXXX}] [DataProcessing] [%p] [] [%C] [tid:%t] [userID:${user.name}]
%m%n

#############################################################
# Facility specific properties.
#############################################################

# These loggers from dependency libraries are explicity set to different logging levels.
# They are known to be very noisy and obsure other log statements.
log4j.logger.org.eclipse.jetty = WARN
log4j.logger.org.apache.spark.repl.SparkIMain$exprTyper = INFO
log4j.logger.org.apache.spark.repl.SparkILoop$SparkILoopInterpreter = INFO
```

The file has the following properties:

Logging property	Description
log4j.rootLogger	The level of the root logger is defined as INFO and attaches the console and edpMain handlers to it.
log4j.appender.console	Defines console as a Log4j ConsoleAppender.
log4j.appender.edpMain	Defines edpMain as EdpODPFormatterAppender (a custom Log4j appender).
log4j.appender.edpMain.ComponentId	Sets DataProcessing as the name of the component that generates the log messages.
log4j.appender.edpMain.Path	Sets the path for the DP output log files. The default path is to the /tmp/edp/log directory. Each log file is named: `edp_%timestamp_%unique.log` See the comments in the log file for the definitions of the %timestamp and %unique variables.
log4j.appender.edpMain.Format	Sets ODL-Text as the formatted string as specified by the conversion pattern.
log4j.appender.edpMain.MaxSegmentSize	Sets the maximum size (in bytes) of a log file. When the file reaches this size, a rollover file is created. The default is 100000000 (about 100 MB).
log4j.appender.edpMain.MaxSize	Sets the maximum amount of disk space to be used by the main log file and the logging rollover files. The default is 1000000000 (about 1GB).

Logging property	Description
`log4j.appender.edpMain.Encoding`	Sets character encoding for the log file. The default `UTF-8` value prints out UTF-8 characters in the file.
`log4j.appender.console.layout`	Sets the `PatternLayout` class for the `console` layout.
`log4j.appender.console.layout.ConversionPattern`	Defines the log entry conversion pattern as: • **%d** is the date of the logging event, in the specified format. • **%p** outputs the priority of the logging event. • **%C** outputs the category of the logging event. • **%m** outputs the application-supplied message associated with the logging event while **%n** is the platform-dependent line separator character. For other conversion characters, see: *https://logging.apache.org/log4j/1.2/apidocs /org/apache/log4j/PatternLayout.html*

`log4j.logger.org.eclipse.jetty`
 	
`log4j.logger.org.apache.spark.repl.SparkIMain$ exprTyper`
 	
`log4j.logger.org.apache.spark.repl.SparkILoop$ SparkILoopInterpreter`
	Sets the default logging level for the Spark and Jetty loggers.

Logging levels

The logging level specifies the amount of information that is logged. The levels (in descending order) are:

- `SEVERE` — Indicates a serious failure. In general, `SEVERE` messages describe events that are of considerable importance and which will prevent normal program execution.

- `WARNING` — Indicates a potential problem. In general, `WARNING` messages describe events that will be of interest to end users or system managers, or which indicate potential problems.

- `INFO` — A message level for informational messages. The `INFO` level should only be used for reasonably significant messages that will make sense to end users and system administrators.

- `CONFIG` — A message level for static configuration messages. `CONFIG` messages are intended to provide a variety of static configuration information, and to assist in debugging problems that may be associated with particular configurations.

- `FINE` — A message level providing tracing information. All options, `FINE`, `FINER`, and `FINEST`, are intended for relatively detailed tracing. Of these levels, `FINE` should be used for the lowest volume (and most important) tracing messages.

- `FINER` — Indicates a fairly detailed tracing message.

- `FINEST` — Indicates a highly detailed tracing message. `FINEST` should be used for the most voluminous detailed output.

- `ALL` — Enables logging of all messages.

These levels allow you to monitor events of interest at the appropriate granularity without being overwhelmed by messages that are not relevant. When you are initially setting up your application in a development environment, you might want to use the `FINEST` level to get all messages, and change to a less verbose level in production.

DP log entry format

DP log levels

DP log entry format

This topic describes the format of Data Processing log entries, including their message types and log levels.

The following is an example of a NOTIFICATION message resulting from the part of the workflow where DP connects to the Hive Metastore:

```
[2016-09-30T10:04:49.200-04:00] [DataProcessing] [NOTIFICATION] [] [hive.metastore]
   [host: web09.example.com] [nwaddr: 10.152.105.219] [tid: Driver] [userId: yarn]
   [ecid: 0000LTvQG_h7ADkpSw4Eyc1Nvb3b000001,0:1] Connected to metastore.
```

The format of the DP log fields (using the above example) and their descriptions are as follows:

Log entry field	Description	Example
Timestamp	The date and time when the message was generated. This reflects the local time zone.	`[2016-09-30T10:04:49.200-04:00]`
Component ID	The ID of the component that originated the message. "DataProcessing" is hard-coded for the DP component.	`[DataProcessing]`
Message Type	The type of message (log level): • INCIDENT_ERROR • ERROR • WARNING • NOTIFICATION • TRACE • UNKNOWN	`[NOTIFICATION]`

Log entry field	Description	Example
Message ID	The message ID that uniquely identifies the message within the component. The ID may be null.	`[]`
Module ID	The Java class that prints the message entry.	`[hive.metastore]`
Host name	The name of the host where the message originated.	`[host: web09.example.com]`
Host address	The network address of the host where the message originated	`[nwaddr: 10.152.105.219]`
Thread ID	The ID of the thread that generated the message.	`[tid: Driver]`
User ID	The name of the user whose execution context generated the message.	`[userId: yarn]`
ECID	The Execution Context ID (ECID), which is a global unique identifier of the execution of a particular request in which the originating component participates. Note that	`[ecid: 0000LTvQG_h7ADkpSw4Eyc1Nvb3b000001,0:1]`
Message Text	The text of the log message.	`Connected to metastore.`

DP log levels

This topic describes the log levels that can be set in the DP `log4j.properties` file.

The Data Processing logger is configured with the type of information written to log files, by specifying the log level. When you specify the type, the DP logger returns all messages of that type, as well as the messages that have a higher severity. For example, if you set the message type to `WARN`, messages of type `FATAL` and `ERROR` are also returned.

The DP `log4j.properties` file lists these four packages for which you can set a logging level:

- `log4j.rootLogger`
- `log4j.logger.org.eclipse.jetty`
- `log4j.logger.org.apache.spark.repl.SparkIMain$exprTyper`
- `log4j.logger.org.apache.spark.repl.SparkILoop$SparkILoopInterpreter`

You can change a log level by opening the properties file in a text editor and changing the level of any of the four packages. You use a Java log level from the table below. :

This example shows how you can manually change a log level setting:

```
log4j.rootLogger = FATAL, console, edpMain
```

In the example, the log level for the main logger is set to FATAL.

Logging levels

The log levels (in decreasing order of severity) are:

Java Log Level	ODL Log Level	Meaning
OFF	N/A	Has the highest possible rank and is used to turn off logging.
FATAL	INCIDENT_ERROR	Indicates a serious problem that may be caused by a bug in the product and that should be reported to Oracle Support. In general, these messages describe events that are of considerable importance and which will prevent normal program execution.
ERROR	ERROR	Indicates a serious problem that requires immediate attention from the administrator and is not caused by a bug in the product.
WARN	WARNING	Indicates a potential problem that should be reviewed by the administrator.
INFO	NOTIFICATION	A message level for informational messages. This level typically indicates a major lifecycle event such as the activation or deactivation of a primary sub-component or feature. This is the default level.
DEBUG	TRACE	Debug information for events that are meaningful to administrators, such as public API entry or exit points. You should not use this level in a production environment, as performance for DP jobs will be slower.

These levels allow you to monitor events of interest at the appropriate granularity without being overwhelmed by messages that are not relevant. When you are initially setting up your application in a development environment, you might want to use the DEBUG level to get most of the messages, and change to a less verbose level in production.

Example of DP logs during a workflow

This example gives an overview of the various DP logs that are generated when you run a workflow with the DP CLI.

The example assumes that the Hive administrator has created a table named **warrantyclaims** (which contains information about customer claims made against automobile warranties). The workflow will be run with the DP CLI, which is described in *DP Command Line Interface Utility on page 42*.

The DP CLI command line is:

```
./data_processing_CLI --database default --table warrantyclaims --maxRecords 1000
```

The `--table` flag specifies the name of the Hive table, the `--database` flag states that the table in is the Hive database named "default", and the `--maxRecords` flag sets the sample size to be a maximum of 1,000 records.

Command stdout

The DP CLI first prints out the configuration with which it is running, which includes the following:

```
./data_processing_CLI --database default --table warrantyclaims --maxRecords 1000
...
[2016-09-15T14:06:32.803-04:00] [DataProcessing] [INFO] []
   [org.apache.hadoop.hive.metastore.HiveMetaStoreClient]
   [tid:main] [userID:tbrady] Connected to metastore.
New collection name = MdexCollectionIdentifier{
   databaseName=edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3,
   collectionName=edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3}
   jobId: 86e1a9e5-138a-491b-b006-944ed955ca39
data_processing_CLI finished with state SUCCESS
```

The **MdexCollectionIdentifier** field lists the names of the Dgraph database and data set (the data set name in Studio be in a user-friendly operation type. The jobId field lists the job ID of the Data Processing workflow.

$BDD_HOME/logs/workflowmanager logs

In this example, the logs in the `$BDD_HOME/logs/workflowmanager` directory are updated. The `diagnostic.log` is of special interest as it contains details of the workflow.

YARN logs

If you use the YARN **ResourceManager Web UI** link, the **All Applications** page shows the Spark applications that have run. In our example, the job name is:

```
ProvisionDataSetFromHiveConfig{hiveDatabaseName=default, hiveTableName=warrantyclaims,
   newCollectionId=MdexCollectionIdentifier{
   databaseName=edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3,
   collectionName=edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3},
   jobId=86e1a9e5-138a-491b-b006-944ed955ca39}
```

The **Name** field shows these characteristics about the job:

- `ProvisionDataSetFromHiveConfig` lists the type of DP workflow that was run. In this example, the workflow is ProvisionDataSetFromHiveConfig, which means that it will create a new BDD data set from a Hive table.

- `hiveDatabaseName` lists the name of the Hive database (**default** in this example).

- `hiveTableName` lists the name of the Hive table that was provisioned (**warrantyclaims** in this example).

- `newCollectionId` lists the name of the new data set and its Dgraph database (both names are the same).

- jobId lists the job ID of the workflow.

Clicking on **History** in the **Tracking UI** field displays the job history. The information in the Application Overview panel includes the name of the name of the user who ran the job, the final status of the job, and the elapsed time of the job. FAILED jobs will have error information in the **Diagnostics** field.

Clicking on **logs** in the **Logs** field displays the `stdout` and `stderr` output. The `stderr` output will be especially useful for FAILED jobs. In addition, the `stdout` section has a link (named **Click here for the full log**) that displays more detailed output information.

Dgraph HDFS Agent log

When the DP workflow finishes, the Dgraph HDFS Agent fetches the DP-created files and sends them to the Dgraph for ingest. The log messages for the Dgraph HDFS Agent component for the ingest operation will be similar to the following entries (note that the message details are not shown):

```
New import request received: MdexCollectionIdentifier{
   databaseName=edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3,
   collectionName=edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3},
   ...
   requestOrigin: FROM_DATASET
Received request for database edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3
About to send event
Event class Event {[[
    jobId: 86e1a9e5-138a-491b-b006-944ed955ca39
    timestamp: 1473962829706
    eventName: INGEST_JOB_START_EVENT
    properties: {}
} sent successfully
]]
Starting ingest for: MdexCollectionIdentifier{
   databaseName=edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3,
   collectionName=edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3},
   ...
   requestOrigin: FROM_DATASET
Finished reading 1020 records for MdexCollectionIdentifier{
   databaseName=edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3,
   collectionName=edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3},
   ...
   requestOrigin: FROM_DATASET
createBulkIngester edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3
sendRecordsToIngester 1020
closeBulkIngester
Ingest finished with 1020 records committed and 0 records rejected.
   Status: INGEST_FINISHED.
   Request info: MdexCollectionIdentifier{
      databaseName=edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3,
      collectionName=edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3},
      ...
      requestOrigin: FROM_DATASET
About to send event
Terminating
Event class Event {[[
    jobId: 86e1a9e5-138a-491b-b006-944ed955ca39
    timestamp: 1473962830989
    eventName: INGEST_JOB_COMPLETED_EVENT
    properties: {}
} sent successfully
]]
Terminating
```

The ingest operation is complete when the final **Status: INGEST_FINISHED** message is written to the log.

Dgraph out log

As a result of the ingest operation for the data set, the Dgraph out log (`dgraph.out`) will have these bulk_ingest messages:

```
Start ingest for collection: edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3
   for database edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3
Starting a bulk ingest operation for database edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3
batch 0 finish BatchCommit status Success
   for database edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3
Ending bulk ingest at client's request
   for database edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3 - finalizing changes
Bulk ingest completed: Added 1020 records and rejected 0 records,
```

```
   for database edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3
Ingest end - 1.285MB in 1.013sec = 1.268MB/sec
   for database edp_cli_edp_3cec5f0b-d682-434e-851c-9a869ccfc7d3
```

At this point, the data set records are in the Dgraph and the data set can be viewed in Studio.

Accessing YARN logs

When a client (Studio or the DP CLI) launches a Data Processing workflow, a Spark job is created to run the actual Data Processing job.

This Spark job is run by an arbitrary node in the Hadoop cluster (node is chosen by YARN). To find the Data Processing logs, use Cloudera Manager (for CDH jobs), Ambari (for HDP jobs), or MapR Control System (for MapR jobs).

To access the YARN logs on CDH or HDP:

1. Use the appropriate Web UI:

    - From the Cloudera Manager home page, click **YARN (MR2 Included)**.

    - From the Ambari home page, click **YARN**.

2. In the YARN menu, click the **ResourceManager Web UI** quick link.

3. The All Applications page lists the status of all submitted jobs. Click on the **ID** field to list job information. Note that failed jobs will list exceptions in the **Diagnostics** field.

4. To show log information, click on the appropriate log in the **Logs** field at the bottom of the Applications page.

To view logs for completed MapR applications (assuming you have enabled the YARN Log Aggregation option):

1. Log on to the MapR Control System.

2. In the Navigation Pane, click **JobHistoryServer**.

3. Click the **Job ID** link for the job that you want to view the logs.

4. In the Logs column of the Application Master section, click the **logs** link.

The Data Processing log also contains the locations of the Spark worker STDOUT and STDERR logs. These locations are listed in the "YARN executor launch context" section of the log. Search for the "SPARK_LOG_URL_STDOUT" and "SPARK_LOG_URL_STDERR" strings, each of which will have a URL associated with it. The URLs are for the worker logs.

Also note that if a workflow invoked the Data Enrichment modules, the YARN logs will contain the results of the enrichments, such as which columns were created.

Workflow Manager logs

The Workflow Manager Service generates five types of logs.

The Workflow Manager logs are stored in the $BDD_HOME/logs/workflowmanager directory.

The logs are:

- diagnostic.log logs workflow events (see below).

- `<date>.request.log` logs requests to Workflow Manager from clients (see below).

- `<date>.native-staderrout.log` captures the JVM native stderr and stdout.

- `<date>.stderrout.log` captures Java System.out and System.err.

- `sql.log` is a log file generated by the EclipseLink part of Workflow Manager.

Diagnostic log

The Workflow Manager diagnostic log contains details of the process and status of workflows. For example, a request for a new data set and the resulting of the submitted job to Spark would be:

```
...
ProvisionDatasetFromHive::processEvent Start{1fb5df0e-66cd-4ce5-8ff8-5b93e2ee2df0}
jobState: JobState:{ProvisionDatasetFromHive, 1fb5df0e-66cd-4ce5-8ff8-5b93e2ee2df0, 0, 1}
state: {1} START
TRANSITION: START  -- Start -->  EXECUTE YARN APPLICATION
stateN: {2} EXECUTE YARN APPLICATION
beginTransaction: {
  state:    2
  input:    {"properties":{"$AT$":"1475169019913","$NV$":"1"}}
  output:   {"properties":{}}
} commit
beginTransaction: {
  input:    {"properties":{"$AT$":"1475169019913","$NV$":"1","$CK$":"0"}}
  output:   {"properties":{"DATABASE_NAME":"edp_cli_edp_37396cf4-fab1-40c8-bb08-d5bb09478d58",
            "COLLECTION_NAME":"edp_cli_edp_37396cf4-fab1-40c8-bb08-d5bb09478d58"}}
} commit
new spark job submission.
waiting for 120000 ms. jobId: 1fb5df0e-66cd-4ce5-8ff8-5b93e2ee2df0
...
Connecting to ResourceManager at web2.example.com/10.152.105.219:8032
Requesting a new application from cluster with 1 NodeManagers
...
Submitting application 1 to ResourceManager
Submitted application application_1475159335282_0001
Application report for application_1475159335282_0001 (state: ACCEPTED)
[[
     client token: N/A
     diagnostics: N/A
     ApplicationMaster host: N/A
     ApplicationMaster RPC port: -1
     queue: root.fcalvin
     start time: 1475169024978
     final status: UNDEFINED
     tracking URL: http://web2.example.com:8088/proxy/application_1475159335282_0001/
     user: fcalvin
]]
...
Application report for application_1475159335282_0001 (state: RUNNING)
[[
     client token: N/A
     diagnostics: N/A
     ApplicationMaster host: 10.152.105.219
     ApplicationMaster RPC port: 0
     queue: root.fcalvin
     start time: 1475169024978
     final status: UNDEFINED
     tracking URL: http://web2.example.com:8088/proxy/application_1475159335282_0001/
     user: fcalvin
]]
...
Application report for application_1475159335282_0001 (state: FINISHED)
[[
     client token: N/A
     diagnostics: N/A
```

```
      ApplicationMaster host: 10.152.105.219
      ApplicationMaster RPC port: 0
      queue: root.fcalviln
      start time: 1475169024978
      final status: SUCCEEDED
      tracking URL: http://web2.example.com:8088/proxy/application_1475159335282_0001/A
      user: fcalvin
]]
MdexCollectionIdentifier{
    databaseName=edp_cli_edp_37396cf4-fab1-40c8-bb08-d5bb09478d58,
    collectionName=edp_cli_edp_37396cf4-fab1-40c8-bb08-d5bb09478d58}:
    agent response for ingest request: Request accepted}
...
publish: IngestStartedEvent{1fb5df0e-66cd-4ce5-8ff8-5b93e2ee2df0}
...
publish: IngestCompletedEvent{1fb5df0e-66cd-4ce5-8ff8-5b93e2ee2df0}
...
notification: class Notification {[[
  workflowName: CLIDataLoad
  sourceDatabaseName: null
  sourceDatasetKey: null
  targetDatabaseName: edp_cli_edp_37396cf4-fab1-40c8-bb08-d5bb09478d58
  targetDatasetKey: edp_cli_edp_37396cf4-fab1-40c8-bb08-d5bb09478d58
  ecid: 1.0000LTqvDmK7ADkpSw4Eyc1NvKfs000000;kXjE
  status: SUCCEEDED
  startTime: 1475169014628
  timestamp: 1475169109265
  progressPercentage: 100.0
  errorMessage: null
  trackingUrl: http://web2.example.com:8088/proxy/application_1475159335282_0001
  properties: {dataSetDisplayName=WarrantyClaims, isCli=true, wm.jobId
=1fb5df0e-66cd-4ce5-8ff8-5b93e2ee2df0}
}
]]
```

Request log

Requests to the Workflow Manager are logged in the `<date>.request.log`. A sample request would be:

```
"POST /bdd.workflowmanager/v1/ProvisionDataSetFromHiveWorkflow HTTP/1.1" 202 48 "-" "Swagger-Codegen
/1.0.0/java" 31458
```

In this example, the request is to start a provisioning workflow for a new data set.

The request entries are prefixed with the IP address from which the requests originated and the time stamp of the request.

Transform Service log

The Transform Service processes transformations on data sets, and also provides previews of the effects of the transformations on the data sets.

The Transform Service logs are stored in the `$BDD_HOME/logs/transformservice` directory. When the Transform Service receives a request to preview a data set, it logs the schema of that data set, as shown in this abbreviated example:

```
16/09/14 20:40:55.732 - INFO  [GridPreviewRunner@37]:- Start processing preview request
for MdexCollectionIdentifier{databaseName=edp_cli_edp_4dd5ac28-2e85-4efc-a3c2-391b6a78f69c,
collectionName=edp_cli_edp_4dd5ac28-2e85-4efc-a3c2-391b6a78f69c}
16/09/14 20:40:55.732 - INFO  [GridPreviewRunner@38]:- class TransformConfig {
    schema: [class Column {
        name: production_country
        type: STRING
```

```
        isSingleAssign: true
        isRecordSearchable: false
        isValueSearchable: true
        language: en
    }, class Column {
        name: dealer_geocode
        type: GEOCODE
        isSingleAssign: true
        isRecordSearchable: false
        isValueSearchable: false
        language: en
...
    }, class Column {
        name: labor_description
        type: STRING
        isSingleAssign: true
        isRecordSearchable: false
        isValueSearchable: true
        language: en
    }]
    transformList: [class PutColumnTransform {
        class TransformInfo {
            transformType: null
        }
        column: class Column {
            name: sentiments
            type: STRING
            isSingleAssign: true
            isRecordSearchable: null
            isValueSearchable: null
            language: null
        }
        exceptionAction: class SetNullAction {
            class TransformExceptionAction {
                actionType: null
            }
            actionType: null
        }
        transformType: null
        script: getSentiment(complaint)
    }]
    resultRowCount: 50
    sort: null
    filter: null
    databaseName: edp_cli_edp_4dd5ac28-2e85-4efc-a3c2-391b6a78f69c
    collectionName: edp_cli_edp_4dd5ac28-2e85-4efc-a3c2-391b6a78f69c
    optimization: null
```

Note that the `transformList` section lists the contents of the transformation script (if one exists). In this example, the Transform `getSentiment` function is used on the `complaint` attribute.


Logging the configuration

When it receives its first preview or transformation request, the Transform Service logs the system, Spark, and Hadoop configurations that it is using. An abbreviated configuration entry is as follows:

```
Number of processors available: 2
Total available memory: 226 MB
Free memory: 170 MB
Maximum available memory: 3403 MB

16/06/29 14:51:37.807 - INFO  [LocalSparkClient@50]:- Spark configuration:
spark.externalBlockStore.folderName = spark-78c17408-b81f-4d0e-a4ac-f06174e67c42
spark.driver.cores = 4
spark.io.compression.codec = lzf
```

```
spark.lib = /localdisk/Oracle/Middleware/BDD-1.3.0.35.999/transformservice/bddservices/spark_lib
/spark-assembly.jar
spark.app.name = transformservice
spark.executor.memory = 4g
spark.master = local[8]
spark.driver.host = 10.152.105.219
spark.executor.id = driver
spark.app.id = local-1467226296747
spark.driver.port = 50018
spark.local.dir = /localdisk/Oracle/Middleware/BDD-1.3.0.35.999/transformservice/tmp
spark.fileserver.uri = http://10.152.105.219:13880
spark.ui.enabled = false
spark.driver.maxResultSize = 4g

16/06/29 14:51:37.966 - INFO  [LocalSparkClient@59]:- Hadoop configuration:
s3.replication = 3
mapreduce.output.fileoutputformat.compress.type = BLOCK
mapreduce.jobtracker.jobhistory.lru.cache.size = 5
hadoop.http.filter.initializers = org.apache.hadoop.http.lib.StaticUserWebFilter
...
yarn.resourcemanager.system-metrics-publisher.enabled = false
mapreduce.client.output.filter = FAILED
```

If you are reporting a Transform Service problem to Oracle Support, make sure you include the Transform Service log when you report the problem.

Chapter 7
Data Enrichment Modules

This section describes the Data Enrichment modules of Big Data Discovery.

About the Data Enrichment modules

The Data Enrichment modules increase the usability of your data by discovering value in its content.

Bundled in the Data Enrichment package is a collection of modules along with the logic to associate these modules with a column of data (for example, an address column can be detected and associated with a GeoTagger module).

During the sampling phase of the Data Processing workflow, some of the Data Enrichment modules run automatically while others do not. If you run a workflow with the DP CLI, you can use the `--excludePlugins` flag to specify which modules should not be run.

After a data set has been created, you can run any module from Studio's **Transform** page.

Pre-screening of input

When Data Processing is running against a Hive table, the Data Enrichment modules that run automatically obtain their input pre-screened by the sampling stage. For example, only an IP address is ever passed to the IP Address GeoTagger module.

Attributes that are ignored

All Data Enrichment modules ignore both the primary-key attribute of a record and any attribute whose data type is inappropriate for that module. For example, the Entity extractor works only on string attributes, so that numeric attributes are ignored. In addition, multi-assign attributes are ignored for auto-enrichment.

Sampling strategy for the modules

When Data Processing runs (for example, during a full data ingest), each module runs only under the following conditions during the sampling phase:

- Entity: never runs automatically.

- TF-IDF: runs only if the text contains between 35 and 30,000 tokens.

- Sentiment Analysis (both document level and sub-document level) : never runs automatically

- Address GeoTagger: runs only on well-formed addresses. Note that the GeoTagger sub-modules (City/Region/Sub-Region/Country) never run automatically.

- IP Address GeoTagger: runs only on IPV4 type addresses (does not run on private IP addresses and does not run on automatically on IPV6 type addresses).

- Reverse GeoTagger: only runs on valid geocode formats.

- Boilerplate Removal: never runs automatically.

- Tag Stripper: never runs automatically.

- Phonetic Hash: never runs automatically.

- Language Detection: runs only if the input text is at least 30 words long. This module is enabled for tokens in the range 30 to 30,000 tokens.

Note that when the Data Processing workflow finishes, you can manually run any of these modules from **Transform** in Studio.

Supported languages

The supported languages are specific to each module. For details, see the topic for the module.

Output attribute names

The types and names of output attributes are specific to each module. For details on output attributes, see the topic for the module.

Data Enrichment logging

If Data Enrichment modules are run in a workflow, they are logged as part of the YARN log. The log entries described which module was run and the columns (attributes) created by the modules.

For example, a data set that contains many geocode values can be produce the following log entries:

```
Running enrichments (if any)..
generate plugin recommendations and auto enrich transform script
TOTAL AVAILABLE PLUGINS: 12
SampleValuedRecommender::Registering Plugin: AddressGeoTaggerUDF
SampleValuedRecommender::Registering Plugin: IPGeoExtractorUDF
SampleValuedRecommender::Registering Plugin: ReverseGeoTaggerUDF
```

```
SampleValuedRecommender::Registering Plugin: LanguageDetectionUDF
SampleValuedRecommender::Registering Plugin: DocLevelSentimentAnalysisUDF
SampleValuedRecommender::Registering Plugin: BoilerPlateRemovalUDF
SampleValuedRecommender::Registering Plugin: TagStripperUDF
SampleValuedRecommender::Registering Plugin: TFIDFTermExtractorUDF
SampleValuedRecommender::Registering Plugin: EntityExtractionUDF
SampleValuedRecommender::Registering Plugin: SubDocLevelSentimentAnalysisUDF
SampleValuedRecommender::Registering Plugin: PhoneticHashUDF
SampleValuedRecommender::Registering Plugin: StructuredAddressGeoTaggerUDF
valid input string count=0, total input string count=101, success ratio=0.0
AddressGeotagger won't be invoked since the success ratio is < 80%
SampleValuedRecommender: --- [ReverseGeoTaggerUDF] plugin RECOMMENDS column: [latlong] for
Enrichment, based on 101 samples
SampleValuedRecommender: --- new enriched column 'latlong_geo_city' will be created from 'latlong'
SampleValuedRecommender: --- new enriched column 'latlong_geo_country' will be created from 'latlong'
SampleValuedRecommender: --- new enriched column 'latlong_geo_postcode' will be created from
'latlong'
SampleValuedRecommender: --- new enriched column 'latlong_geo_region' will be created from 'latlong'
SampleValuedRecommender: --- new enriched column 'latlong_geo_subregion' will be created from
'latlong'
SampleValuedRecommender: --- new enriched column 'latlong_geo_regionid' will be created from
'latlong'
SampleValuedRecommender: --- new enriched column 'latlong_geo_subregionid' will be created from
'latlong'
```

In the example, the Reverse GeoTagger created seven columns.

Entity extractor

The Entity extractor module extracts the names of people, companies, and places from the input text inside records in source data.

The Entity extractor locates and classifies individual elements in text into the predefined categories, which are PERSON, ORGANIZATION, and LOCATION.

The Entity extractor supports only English input text.

Configuration options

This module does not automatically run during the sampling phase of a Data Processing workflow, but you can launch it from **Transform** in Studio.

Output

For each predefined category, the output is a list of names which are ingested into the Dgraph as a multi-assign string Dgraph attribute. The names of the output attributes are:

- `<attribute>_entity_person`
- `<attribute>_entity_loc`
- `<attribute>_entity_org`

In addition, the Transform API has a `getEntities` function that wraps the Name Entity extractor to return single values from the input text.

Example

Assume the following input text:

```
While in New York City, Jim Davis bought 300 shares of Acme Corporation in 2012.
```

The output would be:

```
location: New York City
organization: Acme Corporation
person: Jim Davis
```

Noun Group extractor

This plugin extracts noun groups from the input text.

The Noun Group extractor retrieves noun groups from a string attribute in each of the supported languages. The extracted noun groups are sorted by C-value and (optionally) truncated to a useful number, which is driven by the size of the original document and how many groups are extracted. One use of this plugin is in tag cloud visualization to find the commonly occurring themes in the data.

A typical noun group consists of a determiner (the head of the phrase), a noun, and zero or more dependents of various types. Some of these dependents are:

- noun adjuncts
- attribute adjectives
- adjective phrases
- participial phrases
- prepositional phrases
- relative clauses
- infinitive phrases

The allowability, form, and position of these elements depend on the syntax of the language being used.

Design

This plugin works by applying language-specific phrase grouping rules to an input text. A phrase grouping rule consists of sequences of lexical tests that apply to the tokens in a sentence, identifying a grouping action. The action of a grouping rule is a single part of speech with a weight value, which can be negative or positive integers, followed by optional component labels and positions. The POS (part of speech) for noun groups will use the noun POS. The components must either be head or mod, and the positions are zero-based index into the pattern, excluding the left and right context (if exists).

Configuration options

There are no configuration options.

Note that this plugin is not run automatically during the Data Processing sampling phase (i.e., when a new or modified Hive table is sampled).

Output

The output of this plugin is an ordered list of phrases (single- or multi-word) that are ingested into the Dgraph as a multi-assign, string attribute.

The name of the output attributes is `<colname>_ noun_groups`.

In addition, the Transform API has the `extractNounGroups` function that is a wrapper around the Name Group extractor to return noun group single values from the input text.

Example

The following sentence provides a high-level illustration of noun grouping:

```
The quick brown fox jumped over the lazy dog.
```

From this sentence, the extractor would return two noun groups:

- The quick brown fox

- the lazy dog

Each noun group would be ingested into the Dgraph as a multi-assign string attribute.

TF.IDF Term extractor

This module extracts key words from the input text.

The TF.IDF Term module extracts key terms (salient terms) using a predictable, statistical algorithm. (TF is "term frequency" while IDF is "inverse document frequency".)

The TF.IDF statistic is a common tool for the purpose of extracting key words from a document by not only considering a single document but all documents from the corpus. For the TF.IDF algorithm, a word is important for a specific document if it shows up relatively often within that document and rarely in other documents of the corpus.

The number of output terms produced by this module is a function of the TF.IDF curve. By default, the module stops returning terms when the score of a given term falls below ~68%.

The TF.IDF Term extractor supports these languages:

- English (UK/US)

- French

- German

- Italian

- Portuguese (Brazil)

- Spanish

Configuration options

During a Data Processing sampling operation, this module runs automatically on text that contains between 30 and 30,000 tokens. However, there are no configuration options for such an operation.

In Studio, the Transform API provides a language argument that specifies the language of the input text, to improve accuracy.

Output

The output is an ordered list of single- or multi-word phrases which are ingested into the Dgraph as a multi-assign string Dgraph attribute. The name of the output attribute is `<attribute>_key_phrases`.

Sentiment Analysis (document level)

The document-level Sentiment Analysis module analyzes a piece of text and determines whether the text has a positive or negative sentiment.

It supports any sentiment-bearing text (that is, texts which are not too short, numeric, include only a street address, or an IP address). This module works best if the input text is over 40 characters in length.

This module supports these languages:

- English (US and UK)
- French
- German
- Italian
- Spanish

Configuration options

This module never runs automatically during a Data Processing workflow.

In addition, the Transform API has a `getSentiment` function that wraps this module.

Output

The default output is a single text that is one of these values:

- POSITIVE
- NEGATIVE

Note that NULL is returned for any input which is either null or empty.

The output string is subsequently ingested into the Dgraph as a single-assign string Dgraph attribute. The name of the output attribute is `<attribute>_doc_sent`.

Sentiment Analysis (sub-document level)

The sub-document-level Sentiment Analysis module returns a list of sentiment-bearing phrases which fall into one of the two categories: positive or negative.

The SubDocument-level Sentiment Analysis module obtains the sentiment opinion at a sub-document level. This module returns a list of sentiment-bearing phrases which fall into one of the two categories: positive or negative. Note that this module uses the same Sentiment Analysis classes as the document-level Sentiment Analysis module.

This module supports these languages:

- English (US and UK)

- French

- German

- Italian

- Spanish

Configuration options

Because this module never runs automatically during a Data Processing sampling operation, there are no configuration options for such an operation.

Output

For each predefined category, the output is a list of names which are ingested into the Dgraph as a multi-assign string Dgraph attribute. The names of the output attributes are:

- `<attribute>_sub_sent_neg` (for negative phrases)

- `<attribute>_sub_sent_pos` (for positive phrases)

Address GeoTagger

The Address GeoTagger returns geographical information for a valid global address.

The geographical information includes all of the possible administrative divisions for a specific address, as well as the latitude and longitude information for that address. The Address GeoTagger only runs on valid, unambiguous addresses which correspond to a city. In addition, the length of the input text must be less than or equal to 350 characters.

For triggering on auto-enrichment, the Address GeoTagger requires two or more match points to exist. For a postcode to match, it must be accompanied by a country.

Some valid formats are:

- City + State

- City + State + Postcode

- City + Postcode

- Postcode + Country

- City + State + Country

- City + Country (if the country has multiple cities of that name, information is returned for the city with the largest population)

For example, these inputs generate geographical information for the city of Boston, Massachusetts:

- Boston, MA (or Boston, Massachusetts)

- Boston, Massachusetts 02116

- 02116 US

- Boston, MA US

- Boston US

The final example ("Boston US") returns information for Boston, Massachusetts because even though there are several cities and towns named "Boston" in the US, Boston, Massachusetts has the highest population of all the cities named "Boston" in the US.

Note that for this module to run automatically, the minimum requirement is that the city plus either a state or a postcode are specified.

Keep in mind that regardless of the input address, the geographical resolution does not get finer than the city level. For example, this module will not resolve down to the street level if given a full address. In other words, this full address input:

```
400 Oracle Parkway, Redwood City, CA 94065
```

produces the same results as supplying only the city and state:

```
Redwood City, CA
```

GeoNames data

The information returned by this geocode tagger comes from the GeoNames geographical database, which is included as part of the Data Enrichment package in Big Data Discovery.

Configuration options

This module is run (on well-formed addresses) during a Data Processing sampling operation. However, there are no configuration options for such an operation.

Output

The output information includes the latitude and longitude, as well as all levels of administrative areas.

Depending on the country, the output attributes consist of these administrative divisions, as well as the geocode of the address:

- `<attribute>_geo_geocode` — the latitude and longitude values of the address (such as "42.35843 - 71.05977").

- `<attribute>_geo_city` — corresponds to a city (such as "Boston").

- `<attribute>_geo_country` — the country code (such as "US").

- `<attribute>_geo_postcode` — corresponds to a postcode, such as a zip code in the US (such as "02117").

- `<attribute>_geo_region` — corresponds to a geographical region, such as a state in the US (such as "Massachusetts").

- `<attribute>_geo_regionid` — the ID of the region in the GeoNames database (such as "6254926" for Massachusetts).

- `<attribute>_geo_subregion` — corresponds to a geographical sub-region, such as a county in the US (such as "Suffolk County").

- `<attribute>_geo_subregionid` — the ID of the sub-region in the GeoNames database (such as "4952349" for Suffolk County in Massachusetts).

All are output as single-assign string (`mdex:string`) attributes, except for `Geocode` which is a single-assign geocode (`mdex:geocode`) attribute.

Note that if an invalid input is provided (such as a zip code that is not valid for a city and state), the output may be NULL.

Examples

The following output might be returned for the "Boston, Massachusetts USA" address:

```
ext_geo_city            Boston
ext_geo_country         US
ext_geo_geocode         42.35843 -71.05977
ext_geo_postcode        02117
ext_geo_region          Massachusetts
ext_geo_regionid        6254926
ext_geo_subregion       Suffolk Country
ext_geo_subregionid     4952349
```

This sample output is for the "London England" address:

```
ext_geo_city            City of London
ext_geo_country         GB
ext_geo_geocode         51.51279 -0.09184
ext_geo_postcode        ec4r
ext_geo_region          England
ext_geo_regionid        6269131
ext_geo_subregion       Greater London
ext_geo_subregionid     2648110
```

IP Address GeoTagger

The IP Address GeoTagger returns geographical information for a valid IP address.

The IP Address GeoTagger is similar to the Address GeoTagger, except that it uses IP addresses as its input text. This module is useful IP addresses are present in the source data and you want to generate geographical information based on them. For example, if your log files contain IP addresses as a result of people coming to your site, this module would be most useful for visualization where those Web visitors are coming from.

Note that when given a string that is not an IP address, the IP Address GeoTagger returns NULL.

GeoNames data

The information returned by this geocode tagger comes from the GeoNames geographical database, which is included as part of the Data Enrichment package in Big Data Discovery.

Configuration options

There are no configuration options for a Data Processing sampling operation.

Output

The output of this module consists of the following attributes:

- `<attribute>_geo_geocode` — the latitude and longitude values of the address (such as "40.71427 -74.00597 ").

- `<attribute>_geo_city` — corresponds to a city (such as "New York City").

- `<attribute>_geo_region` — corresponds to a region, such as a state in the US (such as "New York").

- `<attribute>_geo_regionid` — the ID of the region in the GeoNames database (such as "5128638 " for New York).

- `<attribute>_geo_postcode` — corresponds to a postcode, such as a zip code in the US (such as "02117").

- `<attribute>_geo_country` — the country code (such as "US").

Example

The following output might be returned for the 148.86.25.54 IP address:

```
ext_geo_city          New York City
ext_geo_country       US
ext_geo_geocode       40.71427 -74.00597
ext_geo_postcode      10007
ext_geo_region        New York
ext_geo_regionid      5128638
```

Reverse GeoTagger

The Reverse GeoTagger returns geographical information for a valid geocode latitude/longitude coordinates that resolve to a metropolitan area.

The purpose of the Reverse GeoTagger is, based on a given latitude and longitude value, to find the closest place (city, state, country, postcode, etc) with population greater than 5000 people. The location threshold for this module is 100 nautical miles. When the given location exceeds this radius and the population threshold, the result is NULL.

The syntax of the input is:

```
<double>separator<double>
```

where:

- The first double is the latitude, within the range of -90 to 90 (inclusive).

- The second double is the longitude, within the range of -180 to 180 (inclusive).

- The separator is any of these characters: whitespace, colon, comma, pipe, or a combination of whitespaces and one the other separator characters.

For example, this input:

```
42.35843 -71.05977
```

returns geographical information for the city of Boston, Massachusetts.

However, this input:

```
39.30 89.30
```

returns NULL because the location is in the middle of the Gobi Desert in China.

GeoNames data

The information returned by this geocode tagger comes from the GeoNames geographical database, which is included as part of the Data Enrichment package in Big Data Discovery.

Configuration options

There are no configuration options for a Data Processing sampling operation.

In Studio, the **Transform** area includes functions that return only a specified piece of the geographical results, such as only a city or only the postcode.

Output

The output of this module consists of these attribute names and values:

- `<attribute>_geo_city` — corresponds to a city (such as "Boston").

- `<attribute>_geo_country` — the country code (such as "US").

- `<attribute>_geo_postcode` — corresponds to a postcode, such as a zip code in the US (such as "02117").

- `<attribute>_geo_region` — corresponds to a geographical region, such as a state in the US (such as "Massachusetts").

- `<attribute>_geo_regionid` — the ID of the region in the GeoNames database (such as "6254926" for Massachusetts).

- `<attribute>_geo_subregion` — corresponds to a geographical sub-region, such as a county in the US (such as "Suffolk County").

- `<attribute>_geo_subregionid` — the ID of the sub-region in the GeoNames database (such as "4952349" for Suffolk County in Massachusetts).

Tag Stripper

The Tag Stripper module removes any HTML, XML and XHTML markup from the input text.

Configuration options

This module never runs automatically during a Data Processing sampling operation.

When you run it from within **Transform** in Studio, the module takes only the input text as an argument.

Output

The output is a single text which is ingested into the Dgraph as a single-assign string Dgraph attribute. The name of the output attribute is `<attribute>_html_strip`.

Phonetic Hash

The Phonetic Hash module returns a string attribute that contains the hash value of an input string.

A word's phonetic hash is based on its pronunciation, rather than its spelling. This module uses a phonetic coding algorithm that transforms small text blocks (names, for example) into a spelling-independent hash comprised of a combination of twelve consonant sounds. Thus, similar-sounding words tend to have the same hash. For example, the term "purple" and its misspelled version of "pruple" have the same hash value (PRPL).

Phonetic hashing can used, for example, to normalize data sets in which a data column is noisy (for example, misspellings of people's names).

This module works only with whitespace languages.

Configuration options

This module never runs automatically during a Data Processing sampling operation and therefore there are no configuration options.

In Studio, you can run the module within **Transform**, but it does not take any arguments other than the input string.

Output

The module returns the phonetic hash of a term in a single-assign Dgraph attribute named `<attribute>_phonetic_hash`. The value of the attribute is useful only as a grouping condition.

Language Detection

The Language Detection module can detect the language of input text.

The Language Detection module can accurately detect and report primary languages in a plain-text input, even if it contains more than one language. The size of the input text must be between 35 and 30,000 words for more than 80% of the values sampled.

The Language Detection module can detect all languages supported by the Dgraph. The module parses the contents of the specified text field and determines a set of scores for the text. The supported language with the highest score is reported as the language of the text.

If the input text of the specified field does not match a supported language, the module outputs "Unknown" as the language value. If the value of the specified field is NULL, or consists only of white spaces or non-alphabetic characters, the component also outputs "Unknown" as the language.

Configuration options

There are no configuration options for this module, both when it is run as part of a Data Processing sampling operation and when you run it from **Transform** in Studio.

Output

If a valid language is detected, this module outputs a separate attribute with the ISO 639 language code, such as "en" for English, "fr" for French, and so on. There are two special cases when NULL is returned:

- If the input is NULL, the output is NULL.
- If there is a valid input text but the module cannot decide on a language, then the output is NULL.

The name of the output attribute is `<attribute>_lang`.

Updating models

You can update three of the models used by Data Enrichment modules.

You can update these models:

- Sentiment Analysis model, used by the two Sentiment Analysis modules
- TF.IDF model
- GeoTagger model, used by the three GeoTaggers.

Each model is updated by running the `bdd-admin` script with the `update-model` command and the model type argument. More information is available in the topics below.

Updating Sentiment Analysis models

Updating TF.IDF models

Updating GeoTagger models

Updating Sentiment Analysis models

This topic describes how to set up and update the two Sentiment Analysis models with new training data.

The training data sets for the Sentiment Analysis modules consist of two input files with these names:

- `<lang>_pos.txt` contains text with positive sentiment.
- `<lang>_neg.txt` contains text with negative sentiment.

<lang> is a supported country code: `en` (UK/US English), `fr` (French), `de` (German), `it` (Italian), or `es` (Spanish).

The text files should have one sentence per line. You must train your sentiment model against examples of the type of data that you are going to see when you use your model. For example, if you are trying to determine the sentiment of tweets, you will need to obtain examples of tweet review entries. You can either provide your own data or buy it. For a good model, you will need at least several hundred examples, if not thousands.

Each language-specific set of training files must reside in a directory whose name corresponds to the language of the files. The directory names are:

- `american`
- `french`
- `german`

- `italian`

- `spanish`

The suggested naming structure of the entire directory is:

```
/<root>/models/sentiment/<language>
```

where *<language>* is one or more of the above names.

Create a *<language>* directory only if you intend to build models for that language. For example, you can have these two language directories:

```
/share/models/sentiment/american
/share/models/sentiment/french
```

The `american` directory would have the `en_pos.txt` and `en_neg.txt` files, while the `french` directory would have the `fr_pos.txt` and `fr_neg.txt` files.

To update the Sentiment Analysis model:

1. Create the directory structure (explained above) for the Sentiment Analysis training files, with a separate sub-directory for each language version.

    In our example, following directory will be used for the American English version of the training files:

    ```
    /share/models/sentiment/american
    ```

2. Copy the `en_pos.txt` and `en_neg.txt` files into the `/american` directory.

3. Run the `bdd-admin` script with the `update-model` command, the `sentiment` model-type argument, and the absolute path to the `/sentiment` directory:

    ```
    ./bdd-admin.sh update-model sentiment /share/models/sentiment
    ```

If successful, the command prints these messages:

```
[2016/07
/17 15:35:02 -0400] [Admin Server] Generating the sentiment model file using new model
file...Success!
[2016/07/17 15:35:55 -0400] [Admin Server] Publishing the sentiment model file...
[2016/07/17 15:36:07 -0400] [Admin Server] Successfully published the model file.
```

The operation replaces the Sentiment Analysis model's current JAR on the YARN worker nodes with the new one.

You can revert the model by running the command without the path argument:

```
./bdd-admin.sh update-model sentiment
```

This reverts the Sentiment Analysis model to the original, shipped version.

Updating TF.IDF models

This topic describes how to set up and update the TF.IDF model with new training data.

For the TF.IDF training data, you provide one or more language-specific `<lang>_abstracts.zip` files, where *<lang>* is a supported country code:

- `de` (German)

- `en` (US English)

- `es` (Spanish)

- `fr` (French)

- `gb` (UK English)

- `is` (Icelandic)

- `it` (Italian)

- `pt` (Portuguese)

Each ZIP file contains a large number of language training model files that can be any text that's in the given language. You can use a variety of corpora, such as these two widely-used versions:

- Brown corpus, available for download: *http://www.nltk.org/nltk_data/packages/corpora/brown.zip*

- text8 corpus, available for download at: *https://cs.fit.edu/~mmahoney/compression/text8.zip*

All the ZIP files must be in the same directory, which can have any name of your choosing. The example below assumes this directory structure:

```
/share/models/tfidf/en_abstracts.zip
```

The following procedure assumes that you have downloaded a corpus ZIP file and renamed it to `en__abstracts.zip`.

To update the TF.IDF model:

1. Create the directory structure (explained above) for the TF.IDF training files, with one directory for the ZIP files.

2. Copy the `en__abstracts.zip` training file into the `/share/models/tfidf` directory.

3. Run the `bdd-admin` script with the `update-model` command, the `tdidf` model-type argument, and the absolute path to the `/tfidf` directory:

```
./bdd-admin.sh update-model tfidf /share/models/tfidf
```

If successful, the command prints these messages:

```
[2016/07
/15 11:21:42 -0400] [Admin Server] Generating the tfidf model file using new model file...Success!
[2016/07/15 11:24:45 -0400] [Admin Server] Publishing the tfidf model file...
[2016/07/15 11:24:57 -0400] [Admin Server] Successfully published the model file.
```

The operation replaces the TF.IDF model's current JAR on the YARN worker nodes with the new one.

You can revert the model by running the command without the path argument:

```
./bdd-admin.sh update-model tfidf
```

This reverts the TF.IDF model to the original, shipped version.

Updating GeoTagger models

You can update the geographical data sources for the geonames model that is used by the GeoTagger modules.

Before running the procedure below, you must build two scripts:

- `geodb-download-world.sh` is used to download the latest geonames data from the *http://www.geonames.org/* site.

- `<country_code>_stripnames` is a text file that is used by the geonames builder to strip out words from the geonames data that otherwise could yield missing results from the GeoTaggers. The *<country_code>* prefix is the two-character (**ISO 3166-1 alpha-2**) uppercase code for a country. For example "US" for the United States, which results in a `US_stripnames` file name.

The `geodb-download-world.sh` script should look like this example:

```
#!/bin/bash

set -x

wget -V >/dev/null 2>&1 || { echo >&2 "I require 'wget' but it's not installed.  Aborting."; exit 1
; }

rm allCountries_nozip.zip
rm allCountries.zip
wget http://download.geonames.org/export/dump/allCountries.zip
mv allCountries.zip allCountries_geonames.zip

rm allCountries.zip
rm allCountries_zip.zip
wget http://download.geonames.org/export/zip/allCountries.zip
mv allCountries.zip allCountries_postalCode.zip

rm admin1CodesASCII.txt
wget http://download.geonames.org/export/dump/admin1CodesASCII.txt

rm admin2Codes.txt
wget http://download.geonames.org/export/dump/admin2Codes.txt

rm countryInfo.txt
wget http://download.geonames.org/export/dump/countryInfo.txt

rm cities1000.zip
wget http://download.geonames.org/export/dump/cities1000.zip

rm cities5000.zip
wget http://download.geonames.org/export/dump/cities5000.zip

rm GeoLite2-City.mmdb
wget http://geolite.maxmind.com/download/geoip/database/GeoLite2-City.mmdb.gz
gunzip GeoLite2-City.mmdb.gz
```

Note that the `rm` commands allow you to run the script without having to manually delete existing files.

The content of the `<country_code>_stripnames` file should resemble this `US_stripnames` example:

```
 (historical)
township of
city of
town of
 mobile home park
 home park
village of
borough of
, city of
 trailer park
 election precinct
 mining district
 estates mobile home park
unorganized territory of
 trailer court
 census designated place
 designated place
 village mobile home park
 mobile park
 mobile estates
```

In the stripnames file, you should put verbatim patterns (not regular expressions) that will be stripped when they match the beginning or the end of a city name. For example, adding the line:

```
city of
```

causes the geonames raw data for "city of Attleboro" to be represented in the GeoTagger model as "Attleboro" because "city of" matched the prefix of the geonames raw data (this is because "city of" would cause a mismatch for the GeoTaggers). Note that the stripnames file is not trimmed, which allows matching to include the space character.

To update the GeoTagger model:

1.   Create a directory for the GeoTagger files.

     In our example, the following directory structure is used:

     ```
     /share/models/geotagger
     ```

2.   Copy the `geodb-download-world.sh` script and your `<country_code>_stripnames` file into the `/geotagger` directory.

3.   Run the `geodb-download-world.sh` script.

     When the script finishes, the `/geotagger` directory should contain these new files:

     • `admin1CodesASCII.txt`

     • `admin2Codes.txt`

     • `allCountries_geonames.zip`

     • `allCountries_postalCode.zip`

     • `cities1000.zip`

     • `cities5000.zip`

     • `countryInfo.txt`

     • `GeoLite2-City.mmdb`

4.   Run the `bdd-admin` script with the `update-model` command, the `geonames` model-type argument, and the absolute path to the `/geotagger` directory:

     ```
     ./bdd-admin.sh update-model geonames /share/models/geotagger
     ```

If successful, the command prints these messages:

```
[2016/06
/10 16:15:00 -0400] [Admin Server] Generating the geonames model file using new model file...Success!
[2016/06/10 16:33:38 -0400] [Admin Server] Publishing the geonames model file...
[2016/06/10 16:33:49 -0400] [Admin Server] Successfully published the model file.
```

The operation replaces the GeoTagger model's current JAR on the YARN worker nodes with the new one.

You can revert the model by running the command without the path argument:

```
./bdd-admin.sh update-model geonames
```

This reverts the GeoTagger model to the original, shipped version.

Chapter 8

Dgraph Data Model

This section introduces basic concepts associated with the schema of records in the Dgraph, and describes how data is structured and configured in the Dgraph data model. When a Data Processing workflow runs, a resulting data set is created in the Dgraph. The records in this data set, as well as their attributes, are discussed in this section.

About the data model

The data model in the Dgraph consists of data sets, records, and attributes.

- Data sets contain records.

- Records are the fundamental units of data.

- Attributes are the fundamental units of the schema. For each attribute, a record may be assigned zero, one, or more attribute values.

Data records

Records are the fundamental units of data in the Dgraph.

Dgraph records are processed from rows in a Hive table that have been sampled by a Data Processing workflow in Big Data Discovery.

Source information that is consumed by the Dgraph, including application data and the data schema, is represented by records. Data records in Big Data Discovery are the business records that you want to explore and analyze using Studio. A specific record belongs to only one specific data set.

Attributes

An **attribute** is the basic unit of a record schema. Assignments from attributes (also known as **key-value pairs**) describe records in the Dgraph.

For a data record, an assignment from an attribute provides information about that record. For example, for a list of book records, an assignment from the Author attribute contains the author of the book record.

Each attribute is identified by a unique name within each data set. Because attribute names are scoped within their own data sets, it is possible for two attributes to have the same name, as long as each belongs to a different data set.

Each attribute on a data record is itself represented by a record that describes this attribute. Following the book records example, there is a record that describes the Author attribute. A set of these records that describe attributes forms a schema for your records. This set is known as system records. Each attribute in a record in the schema controls an aspect of the attribute on a data record. For example, an attribute on any data record can be searchable or not. This fact is described by an attribute in the schema record.

Assignments on attributes

Attribute data types

Assignments on attributes

Records are assigned values from attributes. An ***assignment*** indicates that a record has a value from an attribute.

A record typically has assignments from multiple attributes. For each assigned attribute, the record may have one or more values. An assignment on an attribute is known as a ***key-value pair (KVP)***.

Not all attributes will have an assignment for every record. For example, for a publisher that sells both books and magazines, the ISBNnumber attribute would be assigned for book records, but not assigned (empty) for most magazine records.

Attributes may be single-assign or multi-assign:

- A single-assign attribute is an attribute for which each record can have at most one value. For example, for a list of books, the ISBN number would be a single-assign attribute. Each book only has one ISBN number.

- A multi-assign attribute is an attribute for which a single record can have more than one value. For the same list of books, because a single book may have multiple authors, the Author attribute would be a multi-assign attribute.

At creation time, the Dgraph attribute is configured to be either single-assign or multi-assign.

Attribute data types

The attribute type identifies the type of data allowed for the attribute value (key-value pair).

The Dgraph supports the following attribute data types:

Attribute type	Description
mdex:string	XML-valid character strings.
mdex:int	A 32-bit signed integer. Although the Dgraph supports mdex:int attributes, they are not used by Data Processing workflows.
mdex:long	A 64-bit signed integer. mdex:long values accepted by the Dgraph can be up to the value of 9,223,372,036,854,775,807.

Attribute type	Description
mdex:double	A floating point value.
mdex:time	Represents the hour and minutes of an instance of time, with the optional specification of fractional seconds. The time value can be specified as a universal (UTC) date time or as a local time plus a UTC time zone offset.
mdex:dateTime	Represents the year, month, day, hour, minute, and seconds of a time point, with the optional specification of fractional seconds. The dateTime value can be specified as a universal (UTC) date time or as a local time plus a UTC time zone offset.
mdex:duration	Represents a duration of the days, hours, and minutes of an instance of time. Although the Dgraph supports mdex:duration attributes, they are not used by Data Processing workflows.
mdex:boolean	A Boolean. Valid Boolean values are true (or 1, which is a synonym for true) and false (or 0, which is a synonym for false).
mdex:geocode	A latitude and longitude pair. The latitude and longitude are both double-precision floating-point values, in units of degrees.

During a Data Processing workflow, the created Dgraph attributes are derived from the columns in a Hive table. For information on the mapping of Hive column data types to Dgraph attribute data types, see *Data type discovery on page 24*.

Supported languages

The Dgraph uses a language code to identify a language for a specific attribute.

Language codes must be specified as valid RFC-3066 language code identifiers. The supported languages and their language code identifiers are:

Afrikaans: af	Danish: da	Indonesian: id	Norwegian Bokmal: nb	Spanish, Latin American: es_lam
Albanian: sq	Divehi: nl	Italian: it	Norwegian Nynorsk: nn	Spanish, Mexican: es_mx
Amharic: am	Dutch: nl	Japanese: ja	Oriya: or	Swahili: sw
Arabic: ar	English, American: en	Kannada: kn	Persian: fa	Swedish: sv
Armenian: hy	English, British: en_GB	Kazakh, Cyrillic: kk	Persian, Dari: prs	Tagalog: tl
Assamese: as	Estonian: et	Khmer: km	Polish: pl	Tamil: ta

Azerbaijani: `az`	Finnish: `fi`	Korean: `ko`	Portuguese: `pt`	Telugu: `te`
Bangla: `bn`	French: `fr`	Kyrgyz: `ky`	Portuguese, Brazilian: `pt_BR`	Thai: `th`
Basque: `eu`	French, Canadian: `fr_ca`	Lao: `lo`	Punjabi: `pa`	Turkish: `tr`
Belarusian: `be`	Galician: `gl`	Latvian: `lv`	Romanian: `ro`	Turkmen: `tk`
Bosnian: `bs`	Georgian: `ka`	Lithuanian: `lt`	Russian: `ru`	Ukrainian: `uk`
Bulgarian: `bg`	German: `de`	Macedonian: `mk`	Serbian, Cyrillic: `sr_Cyrl`	Urdu: `ur`
Catalan: `ca`	Greek: `el`	Malay: `ms`	Serbian, Latin: `sr_Latn`	Uzbek, Cyrillic: `uz`
Chinese, simplified: `zh_CN`	Gujarati: `gu`	Malayalam: `ml`	Sinhala: `si`	Uzbek, Latin: `uz_latin`
Chinese, traditional: `zh_TW`	Hebrew: `he`	Maltese: `mt`	Slovak: `sk`	Valencian: `vc`
Croatian: `hr`	Hungarian: `hu`	Marathi: `mr`	Slovenian: `sl`	Vietnamese: `vn`
Czech: `cs`	Icelandic: `is`	Nepali: `ne`	Spanish: `es`	unknown (i.e., none of the above languages): `unknown`

The language codes are case insensitive.

Note that an error is returned if you specify an invalid language code.

With the language codes, you can specify the language of the text to the Dgraph during a record search or value search query, so that it can correctly perform language-specific operations.

How country locale codes are treated

A country locale code is a combination of a language code (such as `es` for Spanish) and a country code (such as `MX` for Mexico or `AR` for Argentina). Thus, the `es_MX` country locale means Mexican Spanish while `es_AR` is Argentinian Spanish.

If you specify a country locale code for a `Language` element, the software ignores the country code but accepts the language code part. In other words, a country locale code is mapped to its language code and only that part is used for tokenizing queries or generating search indexes. For example, specifying `es_MX` is the same as specifying just `es`. The exceptions to this rule are the codes listed above (such as `pt_BR`).

Note, however, that if you create a Dgraph attribute and specify a country locale code in the `Language` field, the attribute will be tagged with the country locale code, even though the country code will be ignored during indexing and querying.

Language-specific dictionaries and Dgraph database

The Dgraph has two spelling correction engines:

- If the `Language` property in an attribute is set to `en`, then spelling correction will be handled through the English spelling engine (and its English spelling dictionary).

- If the `Language` property is set to any other value, then spelling correction will use the non-English spelling engine (and its language-specific dictionaries).

All dictionaries are generated from the data records in the Dgraph, and therefore require that the attribute definitions be tagged with a language code.

A data set's dictionary files are stored in the Dgraph database directory for that data set.

Specifying a language for a data set

When creating a data set, you can specify the language for all attributes in that data set, as follows:

- Studio: When uploading a file in via the Data Set Creation Wizard, the **Language** field in the **Preview** page (under **Advanced Settings**) lets you select a language.

- DP CLI: The `defaultLanguage` property in the `edp-cli.properties` configuration file sets the language.

Note that you cannot set languages on a per-attribute basis.

Chapter 9

Dgraph HDFS Agent

This section describes the role of the Dgraph HDFS Agent in the exporting and ingesting of data.

About the Dgraph HDFS Agent

Importing records from HDFS for ingest

Exporting data from Studio

Dgraph HDFS Agent logging

About the Dgraph HDFS Agent

The Dgraph HDFS Agent acts as a data transport layer between the Dgraph and an HDFS environment.

The Dgraph HDFS Agent plays two important roles:

- Takes part in the ingesting of records into the Dgraph. It does so by first reading records from HDFS that have been output by a Data Processing workflow and then sending the records to the Dgraph's Bulk Load interface.

- Takes part in the exporting of data from Studio back into HDFS. The exported data can be in the form of either a local file or an HDFS Avro file that can be used to create a Hive table.

Importing records from HDFS for ingest

The Dgraph HDFS Agent plays a major part in the loading of data from a Data Processing workflow into the Dgraph.

The Dgraph HDFS Agent's role in the ingest procedure is to read the output Avro files from the Data Processing workflow, format them for ingest, and send them to the Dgraph.

Specifically, the high-level, general steps in the ingest process are:

1. A Data Processing workflow finishes by writing a set of records in Avro files in the output directory.

2. The Spark client then locates the Dgraph leader node and the Bulk Load port for the ingest, based on the data set name. The Dgraph that will ingest the records must be a leader within the Dgraph cluster, within the BDD deployment. The leader Dgraph node is elected and determined automatically by Big Data Discovery.

3. The Dgraph HDFS Agent reads the Avro files and prepares them in a format that the Bulk Load interface of the Dgraph can accept.

4. The Dgraph HDFS Agent sends the files to the Dgraph via the Bulk Load interface's port.

5. When a job is successfully completed, the files holding the initial data are deleted.

The ingest of data sets is done with a round-robin, multiplexing algorithm. The Dgraph HDFS Agent divides the records from a given data set into batches. Each batch is processed as a complete ingest before the next batch is processed. If two or more data sets are being processed, the round-robin algorithm alternates between sending record batches from each source data set to the Dgraph. Therefore, although only one given ingest operation is being processed by the Dgraph at any one time, this multiplexing scheme does allow all active ingest operations to be scheduled in a fair fashion.

Note that if Data Processing writes a NULL or empty value to the HDFS Avro file, the Dgraph HDFS Agent skips those values when constructing a record from the source data for the consumption by the Bulk Load interface.

Updating the spelling dictionaries

When the Dgraph HDFS Agent sends the ingest request to the Dgraph, it also sets the `updateSpellingDictionaries` flag in the bulk load request. The Dgraph thus updates the spelling dictionaries for the data set from the data corpus. This operation is performed after every successful ingest. The operation also enables spelling correction for search queries against the data set.

Post-ingest merge operation

After sending the record files to the Dgraph for ingest, the Dgraph HDFS Agent also requests a full merge of all generations of the Dgraph database files.

The merge operation consists of two actions:

1. The Dgraph HDFS Agent sends a URL merge request to the Dgraph.

2. If it successfully receives the request, the Dgraph performs the merge.

The final results of the merge are logged to the Dgraph out log.

Exporting data from Studio

The Dgraph HDFS Agent is the conduit for exporting data from a Studio project.

From within a project in Studio, you can export data as a new Avro file (`.avro` extension), CSV file (`.csv` extension), or text file (`.txt` extension). Files can be exported to either an external directory on your computer, or to HDFS. For details on the operation, see the *Studio User's Guide*.

When a user exports a data set to a file in HDFS from Studio, the exported file's owner will always be the owner of HDFS agent process (or the HDFS agent principal owner in a Kerberized cluster). That is, the Dgraph HDFS Agent uses the username from the export request to create a FileSystem object. That way, BDD can guarantee that a file will not be created if the user does not have permissions, and if the file it created, it is owned by that user. The group is assign automatically by Hadoop.

As part of the export operation, the user specifies the delimiter to be used in the exported file:

• If the delimiter is a comma, the export process creates a `.csv` file.

• If the delimiter is anything except a comma, the export process creates a `.txt` file.

If you export to HDFS, you also have the option of creating a Hive table from the data. After the Hive table is created, a Data Processing workflow is launched to create a new data set.

The following diagram illustrates the process of exporting data from Studio into HDFS:

In this diagram, the following actions take place:

1. From **Transform** in Studio, you can select to export the data into HDFS. This sends an internal request to export the data to the Dgraph.

2. The Dgraph communicates with the Dgraph HDFS Agent, which launches the data exporting process and writes the file to HDFS.

3. Optionally, you can choose to create a Hive table from the data. If you do so, the Hive table is created in HDFS.

Errors that may occur during the export are entered into the Dgraph HDFS Agent's log.

Dgraph HDFS Agent logging

The Dgraph HDFS Agent writes its stdout/stderr output to a log file.

The Dgraph HDFS Agent `--out` flag specifies the file name and path of the Dgraph HDFS Agent's stdout/stderr log file. This log file is used for both import (ingest) and export operations.

The name and location of the output log file is set at installation time via the `AGENT_OUT_FILE` parameter of the `bdd.conf` configuration file. Typically, the log name is `dgraphHDFSAgent.out` and the location is the `$BDD_HOME/logs` directory.

The Dgraph HDFS Agent log is especially important to check if you experience problems with loading records at the end of a Data Processing workflow. Errors received from the Dgraph (such as rejected records) are logged here.

Ingest operation messages

The following are sample messages for a successful ingest operation for a data set. The messages have been edited for readability:

```
New import request received: MdexCollectionIdentifier{
   databaseName=edp_cli_edp_534a06d4-2b73-4f81-88ea-e8deaa89a2d6,
   collectionName=edp_cli_edp_534a06d4-2b73-4f81-88ea-e8deaa89a2d6},
   ...
   requestOrigin: FROM_DATASET
Received request for database edp_cli_edp_534a06d4-2b73-4f81-88ea-e8deaa89a2d6
About to send event
Event class Event {[[
    jobId: c407c7e7-eb63-4f80-98a0-a39816da95aa
    timestamp: 1473856704007
    eventName: INGEST_JOB_START_EVENT
    properties: {}
} sent successfully
]]
Starting ingest for: MdexCollectionIdentifier{
   databaseName=edp_cli_edp_534a06d4-2b73-4f81-88ea-e8deaa89a2d6,
   collectionName=edp_cli_edp_534a06d4-2b73-4f81-88ea-e8deaa89a2d6},
   ...
```

```
    requestOrigin: FROM_DATASET
Finished reading 500 records for MdexCollectionIdentifier{
   databaseName=edp_cli_edp_534a06d4-2b73-4f81-88ea-e8deaa89a2d6,
   collectionName=edp_cli_edp_534a06d4-2b73-4f81-88ea-e8deaa89a2d6},
   ...
   requestOrigin: FROM_DATASET
createBulkIngester edp_cli_edp_534a06d4-2b73-4f81-88ea-e8deaa89a2d6
sendRecordsToIngester 500
closeBulkIngester
Ingest finished with 500 records committed and 0 records rejected.
   Status: INGEST_FINISHED.
   Request info: MdexCollectionIdentifier{
      databaseName=edp_cli_edp_534a06d4-2b73-4f81-88ea-e8deaa89a2d6,
      collectionName=edp_cli_edp_534a06d4-2b73-4f81-88ea-e8deaa89a2d6}
   ...
   requestOrigin: FROM_DATASET
About to send event
Terminating
Event class Event {[[
    jobId: c407c7e7-eb63-4f80-98a0-a39816da95aa
    timestamp: 1473856705917
    eventName: INGEST_JOB_COMPLETED_EVENT
    properties: {}
} sent successfully
]]
Terminating
```

Some events in the sample log are:

1. The Data Processing workflow has written a set of Avro files in the `/user/bdd/edp/data/.dataIngestSwamp` directory in HDFS.

2. The Dgraph HDFS Agent starts an ingest operation for the data set.

3. The `createBulkIngester` operation is used to instantiate a Bulk Load ingester instance for the data set.

4. The Dgraph HDFS Agent reads 9983 records from the Avro files.

5. The `sendRecordsToIngester` operation sends the 500 records to the Dgraph's ingester.

6. The Bulk Load instance is closed with the `closeBulkIngester` operation.

7. The `Status: INGEST_FINISHED` message signals the end of the ingest operation. The message also lists the number of successfully committed records and the number of rejected records. In addition, the Dgraph HDFS Agent notifies the client that the ingest has finished, at which point Workflow Manager updates the `status` attribute of the DataSet Inventory with the final status of the ingest operation. The status should be `FINISHED` for a successful ingest or `ERROR` if an error occurred.

Note that throughout the workflow, Dgraph HDFS Agent constantly sends notification updates to Workflow Manager.

Rejected records

It is possible for a certain record to contain data which cannot be ingested or can even crash the Dgraph. Typically, the invalid data will consist of invalid XML characters. In this case, the Dgraph cannot remove or cleanse the invalid data, it can only skip the record with the invalid data. The interface rejects non-XML 1.0 characters upon ingest. That is, a valid character for ingest must be a character according to production 2 of the XML 1.0 specification. If an invalid character is detected, the record with the invalid character is rejected with this error message in the Dgraph HDFS Agent log:

```
Received error message from server: Record rejected: Character <c> is not legal in XML 1.0
```

A source record can also be rejected if it is too large. There is a limit of 128MB on the maximum size of a source record. An attempt to ingest a source record larger than 128MB fails and an error is returned (with the primary key of the rejected record), but the bulk load ingest process continues after that rejected record.

Logging for new and deleted attributes

The Dgraph HDFS Agent logs the names of attributes being created or deleted as result of transforms. For example:

```
Finished reading 499 records for Collection name: default_edp_2a0122f2-4d15-46bf-9669-21333442f10b
Adding attributes to collection: default_edp_2a0122f2-4d15-46bf-9669-21333442f10b
  [NumInStock]
Added attributes to collection: default_edp_2a0122f2-4d15-46bf-9669-21333442f10b
...
Deleting attributes from collection: default_edp_2a0122f2-4d15-46bf-9669-21333442f10b
  [OldPrice2]
Deleted attributes from collection: default_edp_2a0122f2-4d15-46bf-9669-21333442f10b
```

In the example, the NumInStock attribute was added to the data set and the OldPrice2 attribute was deleted.

Log entry format

Logging properties file

Configuring daily log rotation

Log entry format

This topic describes the format of Dgraph HDFS Agent log entries, including their message types and log levels.

The following is an example of a NOTIFICATION message:

```
[2016-09-14T08:38:24.869-04:00] [DgraphHDFSAgent] [NOTIFICATION] []
[com.endeca.dgraph.hdfs.agent.importer.RecordsConsumer]
[host: web05.example.com] [nwaddr: 10.152.104.14]
[tid: edp_cli_edp_534a06d4-2b73-4f81-88ea-e8deaa89a2d6_RecordsConsumer]
[userId: tjones] [ecid: 0000LSch1TM7y0kpSwXBic1NqKIX000000,0:1:2]
sendRecordsToIngester 500
```

The format of the Dgraph HDFS Agent log fields (using the above example) and their descriptions are as follows:

Log entry field	Description	Example
Timestamp	The date and time when the message was generated. This reflects the local time zone.	`[2016-09-14T08:38:24.869-04:00]`
Component ID	The ID of the component that originated the message. "DgraphHDFSAgent" is hard-coded for the Dgraph HDFS Agent.	`[DgraphHDFSAgent]`

Log entry field	Description	Example
Message Type	The type of message (log level): • INCIDENT_ERROR • ERROR • WARNING • NOTIFICATION • TRACE • UNKNOWN	`[NOTIFICATION]`
Message ID	The message ID that uniquely identifies the message within the component. Currently is left blank.	`[]`
Module ID	The Java class that prints the message entry.	`[com.endeca.dgraph.hdfs.agent.importer.RecordsConsumer]`
Host name	The name of the host where the message originated.	`[host: web05.example.com]`
Host address	The network address of the host where the message originated	`[nwaddr: 10.152.104.14]`
Thread ID	The ID of the thread that generated the message.	`[tid: edp_cli_edp_534a06d4-2b73-4f81-88ea-e8deaa89a2d6_RecordsConsumer]`
User ID	The name of the user whose execution context generated the message.	`[userId: tjones]`
ECID	The Execution Context ID (ECID), which is a global unique identifier of the execution of a particular request in which the originating component participates.	`[ecid: 0000LSch1TM7y0kpSwXBic1NqKIX000000,0:1:2]`
Message Text	The text of the log message.	`sendRecordsToIngester 500`

Logging properties file

The Dgraph HDFS Agent has a default Log4j configuration file that sets its logging properties.

The file is named `log4j.properties` and is located in the `$DGRAPH_HOME/dgraph-hdfs-agent/lib` directory.

The log file is a rolling log file. The default version of the file is as follows:

```
log4j.rootLogger=INFO, ROLLINGFILE
#
# Add ROLLINGFILE to rootLogger to get log file output
#    Log DEBUG level and above messages to a log file
log4j.appender.ROLLINGFILE=oracle.core.ojdl.log4j.OracleAppender
log4j.appender.ROLLINGFILE.ComponentId=DgraphHDFSAgent
log4j.appender.ROLLINGFILE.Path=${logfilename}
log4j.appender.ROLLINGFILE.Format=ODL-Text
log4j.appender.ROLLINGFILE.MaxSegmentSize=10485760
log4j.appender.ROLLINGFILE.MaxSize=1048576000
log4j.appender.ROLLINGFILE.Encoding=UTF-8
log4j.appender.ROLLINGFILE.layout = org.apache.log4j.PatternLayout
log4j.appender.ROLLINGFILE.layout.ConversionPattern
= %-d{yyyy-MM-dd HH:mm:ss}  [ %t:%r ] - [ %p ]  %m%n
```

The file defines the `ROLLINGFILE` appenders for the root logger and also sets the log level for the file.

The file has the following properties:

Logging property	Description
`log4j.rootLogger`	The level of the root logger is defined as `INFO` and attaches the `ROLLINGFILE` appender to it. You can change the log level, but do not change the `ROLLINGFILE` appender.
`log4j.appender.ROLLINGFILE`	Sets the appender to be `OracleAppender`. This defines the ODL (Oracle Diagnostics Logging) format for the log entries. Do not change this property.
`log4j.appender.ROLLINGFILE.ComponentId`	Sets `DgraphHDFSAgent` as the name of the component that generates the log messages. Do not change this property.
`log4j.appender.ROLLINGFILE.Path`	Sets the path for the log files. The `${logfilename}` variable picks up the path from the Dgraph HDFS Agent `--out` flag used at start-up time. Do not change this property.
`log4j.appender.ROLLINGFILE.Format`	Sets `ODL-Text` as the formatted string as specified by the conversion pattern. Do not change this property.
`log4j.appender.ROLLINGFILE.MaxSegmentSize`	Sets the maximum size (in bytes) of the log file. When the `dgraphHDFSAgent.out` file reaches this size, a rollover file is created. The default is 10485760 (about 10 MB).

Logging property	Description
log4j.appender.ROLLINGFILE.MaxSize	Sets the maximum amount of disk space to be used by the dgraphHDFSAgent.out file and the logging rollover files. The default is 1048576000 (about 1GB).
log4j.appender.ROLLINGFILE.Encoding	Sets character encoding for the log file. The default UTF-8 value prints out UTF-8 characters in the file.
log4j.appender.ROLLINGFILE.layout	Sets the org.apache.log4j.PatternLayout class for the layout.
log4j.appender.ROLLINGFILE.layout.ConversionPattern	Defines the log entry conversion pattern. For the conversion characters, see: https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/PatternLayout.html

Logging levels

You can change the log level by opening the properties file in a text editor and changing the level for the log4j.rootLogger property to a Java log level from the table below. This example shows how you can change the log level setting to ERROR:

```
log4j.rootLogger=ERROR
```

When writing log messages, however, the logging system converts the Java level to an ODL equivalent level. The table below The log levels (in decreasing order of severity) are:

Java Log Level	ODL Log Level	Meaning
OFF	N/A	Has the highest possible rank and is used to turn off logging.
FATAL	INCIDENT_ERROR	Indicates a serious problem that may be caused by a bug in the product and that should be reported to Oracle Support. In general, these messages describe events that are of considerable importance and which will prevent normal program execution.
ERROR	ERROR	Indicates a serious problem that requires immediate attention from the administrator and is not caused by a bug in the product.
WARN	WARNING	Indicates a potential problem that should be reviewed by the administrator.

Java Log Level	ODL Log Level	Meaning
INFO	NOTIFICATION	A message level for informational messages. This level typically indicates a major lifecycle event such as the activation or deactivation of a primary sub-component or feature. This is the default level.
DEBUG	TRACE	Debug information for events that are meaningful to administrators, such as public API entry or exit points.

These levels allow you to monitor events of interest at the appropriate granularity without being overwhelmed by messages that are not relevant. When you are initially setting up your application in a development environment, you might want to use the INFO level to get most of the messages, and change to a less verbose level in production.

Configuring daily log rotation

This topic describes how to configure the Agent log for daily rotation.

By default, the Dgraph HDFS Agent log does not support daily rotation. However, you can modify the `log4j.properties` file for this feature.

To configure the Dgraph HDFS Agent log for daily rotation.

1. Open the `log4j.properties` file in a text editor.

2. Add the following line to the file:

   ```
   log4j.appender.ROLLINGFILE.RotationFrequency=1440
   ```

3. Remove these two lines:

   ```
   log4j.appender.ROLLINGFILE.MaxSegmentSize=10485760
   log4j.appender.ROLLINGFILE.MaxSize=1048576000
   ```

4. Save the file.

5. Re-start the Dgraph HDFS Agent:

   ```
   ./bdd-admin.sh restart -c agent
   ```

Keep in mind that you can force rotate the logs by running the `bdd-admin` script with the `rotate-logs` command, as in this example:

```
./bdd-admin.sh rotate-logs -c agent -n web009.us.example.com
```

As a result of this example, the `dgraphHDFSAgent.out` log is renamed to `dgraphHDFSAgent.out-1438022767` and an empty `dgraphHDFSAgent.out` log is created.

For information on the `rotate-logs` command, see the *Administrator's Guide*.

Index

A

Address GeoTagger 92

assignments 104

attributes
    data types 104
    multi-assign 104
    setting searchability thresholds 41
    single-assign 104

B

blacklists, CLI 51

boolean attribute type 105

C

Cleaning the source data 12

CLI, DP
    about 43
    configuration 46
    cron job 52
    flags 47
    --incrementalUpdate flag 67
    logging 46
    permissions 45
    --refreshData flag 62
    running Incremental updates 69
    running Refresh updates 63
    white- and blacklists 51

configuration
    date formats 36
    Dgraph HDFS Agent logging 113
    DP CLI 46
    DP logging 73
    DP workflows 30
    Spark worker 37

cron job
    Hive Table Detector 52
    Refresh and Incremental updates 70

D

Data Enrichment modules
    about 86
    Entity extractor 88
    excluding from workflows 54
    IP Address GeoTagger 94
    Language Detection 97
    Noun Group extractor 89
    Phonetic Hash 97
    Reverse GeoTagger 95
    Sentiment Analysis, document 91
    Sentiment Analysis, sub-document 91
    Tag Stripper 96

    TF.IDF Term extractor 90

data model, Dgraph 103

Data Processing workflows
    about 15
    excluding enrichments 54
    Kerberos support 10
    logging 72
    processing Hive tables 21
    sampling 23

Data Set Logical Name for updates 60

data type conversions from Hive to Dgraph 24

date formats, supported 36

dateTime attribute type 105

Dgraph
    attributes 103
    data model 103
    Kerberos support 11
    record assignments 104
    supported languages 105

Dgraph HDFS Agent
    about 108
    daily log rotation 116
    exporting data from Studio 109
    ingesting records 108
    Kerberos support 11
    logging 110
    logging configuration 113

disabling record and value search 23

double attribute type 105

E

enrichments 20

Entity extractor 88

F

flags, CLI 47

G

geocode attribute type 105

GeoTagger model, updating 100

H

Hadoop integration with BDD 8

HDFS Data at Rest Encryption for BDD 12

health check for DP components 54

Hive tables
    created from Studio 28
    ingesting 21