Oracle® Big Data Discovery

Data Processing Guide

Version 1.0.0 • Revision A • March 2015

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 Hadoop to transform raw data into business insight in minutes, without the need to learn complex products or rely only on highly skilled resources.

About this guide

This guide describes the Data Processing component of Big Data Discovery (BDD). This guide provides a "behind the scenes" view of Big Data Discovery processes and logic used for various tasks within Data Processing, such as sampling and loading of data.

The Data Processing workflow is launched either from Studio, in which case it runs automatically, or you can control it through the command line interface (DP CLI). In either case, when this 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 workflow***. This guide describes these processes in detail.

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 within Hadoop. 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.

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

Who should use this guide

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.

Conventions used in this document

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 phrases 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.

Symbol conventions

The following table describes symbol conventions used in this document.

Symbol	Description	Example	Meaning
>	The right angle bracket, or greater-than sign, indicates menu item selections in a graphic user interface.	File > New > Project	From the File menu, choose New, then from the New submenu, choose Project.

Path variable conventions

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

Path variable	Meaning
`$MW_HOME`	Indicates the absolute path to your Oracle Middleware home directory, which is the root directory for your WebLogic installation.
`$DOMAIN_HOME`	Indicates the absolute path to your WebLogic domain home directory. For example, if `bdd_domain` is the domain name, then the `$DOMAIN_HOME` value is the `$MW_HOME/user_projects/domains/bdd_domain` directory.
`$BDD_HOME`	Indicates the absolute path to your Oracle Big Data Discovery home directory. For example, if `BDD1.0` is the name you specified for the Oracle Big Data Discovery installation, then the `$BDD_HOME` value is the `$MW_HOME/BDD1.0` directory.
`$DGRAPH_HOME`	Indicates the absolute path to your Dgraph home directory. For example, the `$DGRAPH_HOME` value might be the `$BDD_HOME/dgraph` directory.

Contacting Oracle Customer Support

Oracle Customer Support provides registered users with 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 topic provides a high-level introduction to the Data Processing component of Big Data Discovery.

About Integration with Hadoop

This topic discusses how BDD fits into the Hadoop environment.

Hadoop is a platform for storing, accessing, and analyzing all kinds of data: structured, unstructured and other, such as logs, and data from the Internet Of Things. Hadoop is broadly adopted by IT organizations; with lots of data sets being added to the Hadoop platform rapidly.

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 Hadoop environment in which the rest of your work takes place. 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 Hadoop, Oracle Big Data Discovery achieves data discovery for any data, at significantly-large scale, with high query-processing performance.

BDD inside the Hadoop Data 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 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 Cloudera Distribution for Hadoop (CDH) cluster at your site. Note that the existing CDH 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 enrichment takes 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 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 utilizes its access to Hadoop as an additional processing engine for data analysis.
  - It brings you insights without having to work for them — this is achieved by data discovery, sampling, profiling, and enrichments.
  - It lets you instantly join any combination of data sets.

Benefits of integration of BDD with Hadoop ecosystem

Big Data Discovery is deployed directly on a subset of nodes in the pre-existing CDH 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 Hadoop 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.

About Cloudera CDH

Big Data Discovery works with very large amounts of data which may already be stored within HDFS. A Hadoop distribution is a prerequisite for the product, and is critical for the functionality provided by the product. Cloudera CDH is the world's most complete, tested, and popular distribution of Apache Hadoop and related projects. CDH is 100% Apache-licensed open source and is the only Hadoop solution to offer unified batch processing, interactive SQL and interactive search, and role-based access controls.

CDH delivers the core elements of Hadoop — scalable storage and distributed computing — along with additional components, such as a user interface, plus necessary enterprise capabilities, such as security. In particular, BDD uses the HDFS, Hive, Oozie, and Spark components which are all packaged within the CDH distribution along with easy-to-use Web user interfaces.

Data Processing within Big Data Discovery

Data Processing collectively refers to a set of processes and jobs, all launched by Big Data Discovery once it is deployed. Many of these processes run in Hadoop and perform discovery, sampling, profiling, and enrichment of source data.

Data Processing Workflow

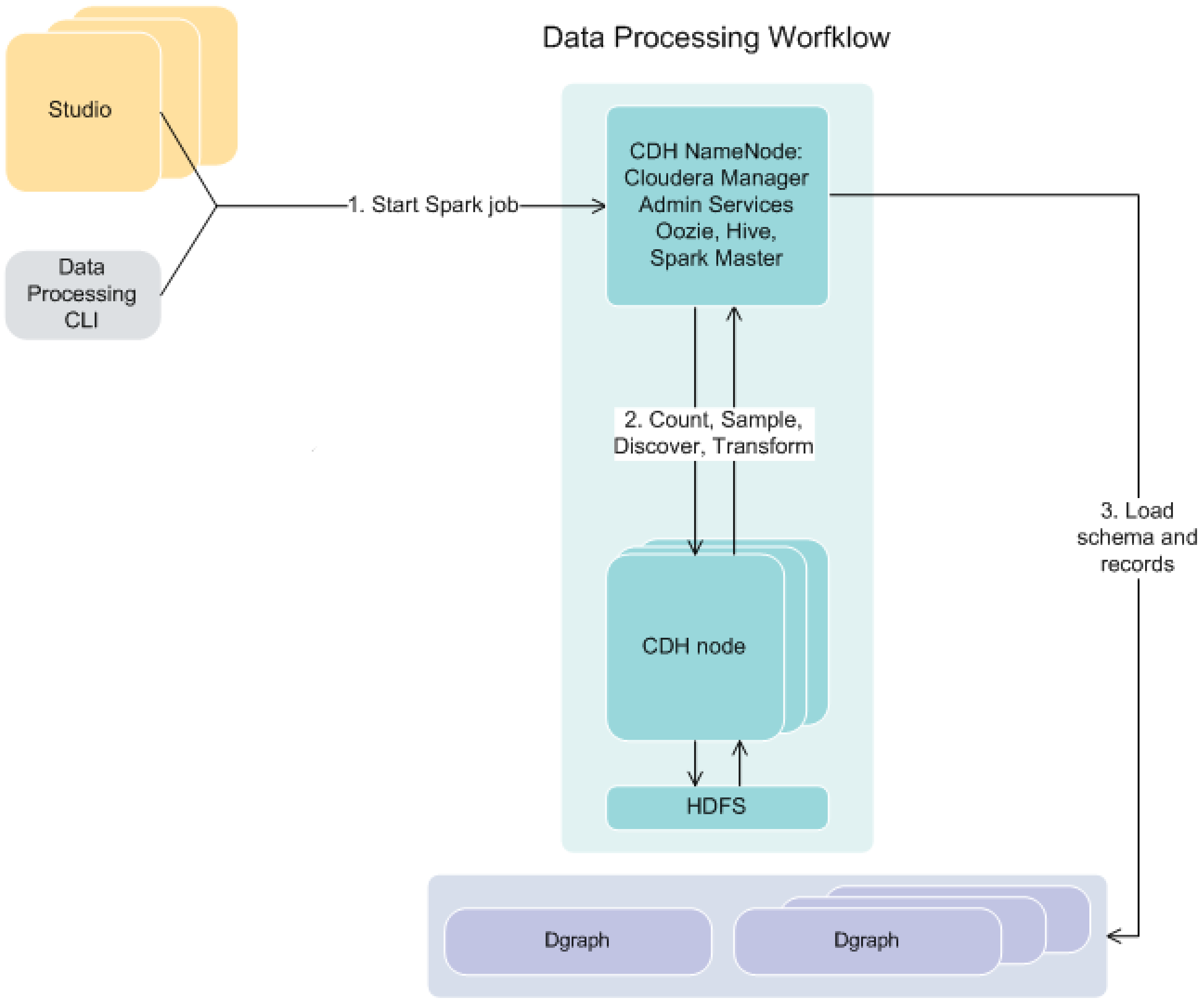
A ***Data Processing workflow*** is a stage in Big Data Discovery processing that includes:

- 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
- Profiling the data
- Transforming the data set
- Exporting data from Big Data Discovery into Hadoop

More information on some of these topics is found below.

You launch the data processing workflow 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 parts of the data processing workflow.

The following diagram illustrates how the data processing workflow fits within the larger picture of Big Data Discovery:


Data Processing Worfklow
Studio
Data Processing CLI
1. Start Spark job
CDH NameNode: Cloudera Manager Admin Services Oozie, Hive, Spark Master
2. Count, Sample, Discover, Transform
CDH node
HDFS
3. Load schema and records
Dgraph
Dgraph

The steps in this diagram are:

1. The data processing workflow starts either from Studio (automatically), or when you run the Data Processing CLI.
2. The Spark job is launched on those CDH NameNodes on which Big Data Discovery is installed.
3. The counting, sampling, discovery and transformations take place and are processed on CDH nodes. The information is written to HDFS and sent back.
4. Next, the data processing workflow launches the process of loading the records and their schema into the Dgraph, for each discovered source data set.

Sampling of a data set

During data processing, Big Data Discovery discovers data in Hive tables, and performs ***data set sampling*** and initial data profiling using enrichments.

Working with data at very large scales causes latency and reduces the interactivity of data analysis. To avoid these issues in Big Data Discovery, you work with a sampled subset of the records from large tables discovered in HDFS. Using sample data as a proxy for the full tables, you can analyze the data as if using the full set.

During data processing, a random sample of the data is taken. The default sample size is 1 million records. Administrators can adjust the sample size.

Samples in BDD are taken as follows. 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. The result is that a simple random sampling of records is created, in which:

- Every element has the same probability of being chosen, and
- 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 to 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 Big Data Discovery during data processing.

Profiling is carried out by the processing workflow 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

Transformations can be thought of as a substitute for an ETL process of cleaning your data before or during the data loading process. Transformations can be used to overwrite an existing attribute, or create new attributes.

Most transformations are available directly as specific options in **Transform** in Studio. Some transformations are enrichments.

The custom transformation option lets you use the Groovy scripting language and a list of custom, predefined Groovy functions available in Big Data Discovery, to create a transformation formula.

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.

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, if you choose to do so.

Note that after a data set is created in Big Data Discovery, you can manipulate the contents of the data set by using **Transform** and its 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 skipped (that is, will not be ingested into the Dgraph). 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 character is detected, an exception is thrown with this error message:

```
Character <c> is not legal in XML 1.0
```

The record with that character is rejected.

Fixing date formats

Ingested date values originate 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 24*.

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.

Chapter 2

Data Processing Workflows

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

About workflows

This topic provides an overview of Data Processing workflows.

A Data Processing (DP) workflow 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. Data Processing workflows are launched from Studio or by running the DP CLI (command line interface) utility.

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.

A Data Processing job is run by a Spark worker that has been assigned by Oozie. 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.

DataSet Inventory

The ***DataSet Inventory*** is an internal structure that lets Data Processing keep track of the available data sets. The DataSet Inventory metadata 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 Dgraph HDFS Agent to denote the completion of an ingest.

Language setting for attributes

During a normal Data Processing workflow, the default language setting for all attributes is `unknown` (which means a DP workflow does not use a language code for any specific language). Both Studio and the DP Command Line Interface utility can be configured with a specific language code to be used for a workflow.

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 generates an error when an empty Hive table is found, and no data set is created for that empty table.

Starting workflows

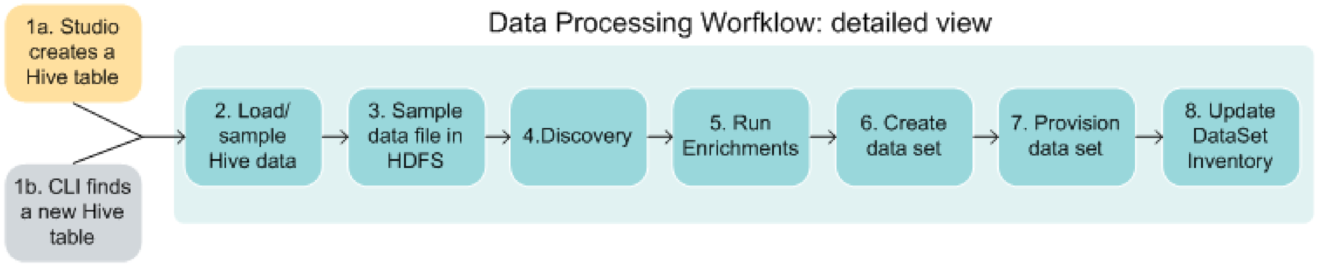
A Data Processing workflow can be started in one of two ways:

- 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 28*.

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:


Data Processing Worfklow: detailed view
1a. Studio creates a Hive table
1b. CLI finds a new Hive table
2. Load/ sample Hive data
3. Sample data file in HDFS
4.Discovery
5. Run Enrichments
6. Create data set
7. Provision data set
8. Update DataSet Inventory

The steps in the workflow are:

1. The workflow is started for a single Hive table by Studio or by the DP CLI.
2. An Oozie job is started, which in turn assigns the workflow 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 an Avro 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, the Dgraph HDFS Agent 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 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 changed. This means that now the BDD data set is not in synch with its Hive source table.

If you want the updated Hive table to be processed (i.e., create a new data set), do the following:

1. From Studio, delete the data set. As part of the delete operation, the Data Processing workflow adds the `skipAutoProvisioning` property to the Hive table from which the data set was sourced. This property will prevent Data Processing from processing the table.
2. From the Hive environment, remove the `skipAutoProvisioning` property from the Hive table.
3. Run the CLI, which will launch a Data Processing workflow for the table.

Data Processing creates a new data set representing the newer version of the Hive table.

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 handle empty Hive tables. Instead, Data Processing throws an `EmptyHiveTableException` when running against an empty Hive table. This causes the DP Oozie job to fail.

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

Data Processing samples (processes) a number of records from a Hive table.

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

- In Studio, the `bdd.maxRecordsToProcess` parameter in the **Data Processing Settings** panel on Studio's Control Panel.
- In DP CLI, `maxRecordsProcessed` configuration parameter or the `--maxRecords` flag.

The sampled records comprise the records in a data set.

Discovery for attributes

The Data Processing discovery phase discovers the DataSet metadata in order to suggest a Dgraph attribute schema. For detailed information on the Dgraph schema, see *Data Model in Big Data Discovery on page 55*.

Record and value search settings for string attributes

When the Data Processing data type discoverer determines that an attribute should be a string attribute, the settings for the record search and value search for this attribute are configured as follows:

- The attribute is configured as value-searchable if the average string length is equal or less than 200 characters.
- The attribute is configured as record-searchable if the average string length is greater than 200 characters.

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

Effect of NULL values on column conversion

When a Hive table is being sampled, a Dgraph tribute 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-processed 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 DataSet 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.

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 *Data Model in Big Data Discovery on page 55*.

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. These `-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`

Hive data type	Hive description	Dgraph data type conversion
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 Note that 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
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.

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 24*.

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 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.

Studio creation of Hive tables

Hive tables can be created from Studio.

The Studio user can create a Hive table by:

- Uploading data from an Excel or CSV file.
- 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 *Data Exploration and Analysis 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.

Creation of a search interface

Data Processing creates a search interface for each data set.

A search interface controls record search behavior for groups of one or more string attributes from the same data set. Each string attribute that has been configured to be record-searchable and is longer than 200 characters is added as a member of the search interface. Each search interface is named **All** (there is no naming conflict among the search interfaces for different data sets as each one is scoped to a different data set).

Snippeting

Snippeting is also enabled for each search interface attribute, with a value of 10 for the snippet size. This means that a snippet can contain a maximum of 10 words.

When the Studio user performs a record search query, Big Data Discovery returns an excerpt from a record. This is called ***snipetting***. A ***snippet*** contains the search terms that the end user provided, along with a portion of the term’s surrounding content to provide context. With the added context, users can more quickly choose the individual records they are interested in.

Chapter 3

Data Processing Configuration

This section describes configuration for date formats and configuration for Spark. It also discusses how to add a SerDe JAR to the Data Processing 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.

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'
EEE d MMM yyyy HH:mm:ss Z
H:mm
h:mm a
```

```
H:mm:ss
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 in this topic. 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.

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.

The `sparkContext.properties` file is located in the `$CLI_HOME/edp_cli/config` directory. As shipped, the 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/1.1.0/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 not run 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 Cloudera CDH default settings.
2. From the Data Processing configuration settings, which can either override the Cloudera settings, and/or provide additional settings. For example, the `sparkExecutorMemory` property (in the DP CLI configuration) can override the CDH `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)
```

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:

- `maxRecordsProcessed` (default is 10000)
- `runEnrichment` (default is `false`)
- `sparkExecutorMemory` (default is 10GB)

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 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*

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, instead of the SerDe class that is shipped in the Data Processing package.

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

This procedure assumes two pre-requisites:

- The BDD Data Processing artifacts must already be present in the CDH cluster (that is, they must be present on HDFS path `/home/username/oozieEdpLib`, which is where the `data_processing_CLI` variable `hdfsEdpLibPath` should be pointing).
- Before integrating the SerDe JAR with Data Processing, the SerDe JAR should be present on the CDH cluster's Hive node. 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, whether the query is sent via Hue or from the Hive CLI.

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

1. Copy the SerDe JAR into the `hdfsEdpLibPath` directory (where all the cluster-side DP JARS are located).

2. In the `hdfsEdpLibPath` directory on HDFS, edit the `spark_worker_files.txt` and `edp_classpath` files to include the SerDe JAR name.

   You can edit the files in the Hue file browser by clicking on the file. Next, the left pane will show an **edit file** option.

   Note that you do not need to edit the files on the client machine on which the Data Processing CLI is run.

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.

About the DP CLI

The DP CLI (Command Line Interface) shell utility is used to launch Data Processing workflows.

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 in Oozie. 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.

You can run the DP CLI either manually or from a cron job. By default, the BDD installer does not create a cron job as part of the installation procedure.

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, depending on the presence of a special table property, `skipAutoProvisioning`. The `skipAutoProvisioning` property 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. 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.

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.
  - 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.

Logging

The DP CLI logs detailed information about its workflow into the log file defined in the `$CLI_HOME/config/logging.properties` file. This file is documented in *Logging configuration on page 38*.

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 configuration properties are contained in the `data_processing_CLI` script.

To set the CLI configuration parameters, open the `data_processing_CLI` script with a text editor. Some of the default values for the parameters are populated from the `bdd.conf` configuration file used during the installation of Big Data Discovery.

In general, the settings below should match those in the **Data Processing Settings** panel on Studio's Control Panel. Parameters that must be the same are mentioned in the table. For information on Studio's **Data Processing Settings** panel, see the *Administrator's Guide*.

Data Processing Defaults

The parameters in `data_processing_CLI` that set the Data Processing defaults are:

Data Processing parameter	Description
`maxRecordsProcessed`	The maximum number of records to be processed for each Hive table (that is, the number of sampled records from the table). The default is 1000000. In effect, this sets the maximum number of records in a BDD data set. You can override this setting by the CLI `--maxRecords` flag.
`runEnrichment`	Specifies whether to run the Data Enrichment modules. The default is `true`. You can override this setting by the CLI `--runEnrichment` flag.
`defaultLanguage`	The language for all attributes in the created data set. The default language code is `en` (US English). For the supported language country codes, see *Supported languages on page 59*.
`edpDataDir`	Specifies the location of the HDFS directory where data ingest and transform operations are processed. The default location is the `/user/bdd/edp/data` directory. Must match the **bdd.edpDataDir** setting in Studio.

Settings controlling access to the Dgraph Gateway

These parameters are used in `data_processing_CLI` for the Dgraph Gateway that is managing the Dgraph nodes:

Dgraph Gateway parameter	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`

Settings controlling access to Hadoop

The parameters that define connections to CDH processes and resources are:

Hadoop parameter	Description
`oozieHost`	Name of the host on which the Oozie server is running. The default value is at the BDD installation time. Must match the **bdd.hadoopClusterHostname** setting in Studio.
`ooziePort`	Port on which the Oozie server is listening. The default value is set at the BDD installation time. Must match the **bdd.oozieServerPort** setting in Studio.
`oozieJobsDir`	Path to the working directory for Oozie Data Processing job files. The default location is the `/user/bdd/edp/oozieJobs` directory. Must match the **bdd.edpOozieJobsDir** setting in Studio.
`oozieWorkerJavaExecPath`	Path to the `java` executable file of the Java SDK on the Oozie worker that should be used to launch the Data Processing process. Must match the **bdd.javaPath** setting in Studio.
`hdfsEdpLibPath`	HDFS path to the Data Processing libraries directory. The default location is the `/user/bdd/edp/lib` directory. Must match the **bdd.hdfsEdpLibPath** setting in Studio.
`hiveServerHost`	Name of the host on which the Hive server is running. The default value is set at the BDD installation time. Must match the **bdd.hadoopClusterHostname** setting in Studio.
`hiveServerPort`	Port on which the Hive server is listening. The default value is set at the BDD installation time. Must match the **bdd.hiveMetastoreServerPort** setting in Studio.
`sparkMasterHost`	Name of the host on which the Spark Master server is running. The default value is set at the BDD installation time. Must match the **bdd.hadoopClusterHostname** setting in Studio.
`sparkMasterPort`	Port on which the Spark Master server is listening. The default value is set at the BDD installation time. Must match the **bdd.sparkServerPort** setting in Studio.

Hadoop parameter	Description
`sparkExecutorMemory`	Amount of memory to use per executor process, in the same format as JVM memory strings (such as, 512m, 2g, 10g, and so on). The default is `48g`. 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).
`edpJarDir`	Path to the directory where the Data Processing JAR files for Spark workers are located on the cluster. The default location is the `/opt/bdd/edp/lib` directory. Must match the **bdd.edpJarDir** setting in Studio.
`clusterOltHome`	Path to the OLT directory on the Spark worker node. The default location is the `/opt/bdd/edp/olt` directory. Must match the **bdd.clusterOlthome** setting in Studio.
`sparkMaxNumberCores`	Maximum number of CPU cores to use for a Spark job. The default is `0`. The default is used to set the same number of cores as the number of used blocks from the target data on HDFS.
`kryoMode`	Specifies whether to enable (`true`) or disable (`false`) Kryo for serialization. The default is `false` and is the recommended setting for Data Processing workflows.
`kryoBufferMemSizeMB`	Maximum object size (in MBs) to allow within Kryo. (The library needs to create a buffer at least as large as the largest single object you will serialize). The default is `1024`. Increase this setting if you get a `buffer limit exceeded` exception inside Kryo. Note that there will be one buffer per core on each worker.

JAVA_HOME setting

In addition to setting the CLI configuration properties, make sure that the `JAVA_HOME` environment variable is set to the directory containing the specific version of Java that will be called when you run the Data Processing CLI.

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.

You can use these flags if you run the CLI without any arguments. Note that each flag has a full name that begins with two dashes, such as `--maxRecords`, and an abbreviated version that uses one dash, such as `-m`.

The CLI flags are:

CLI flag	Description
`-a`, `--all`	Runs data processing on all Hive tables in all Hive databases.
`-bl`, `--blackList<`*`bl-file`*`>`	Specifies the file name for the blacklist used to filter out Hive tables. The tables in this list are ignored by Data Processing and not provisioned.
`-d`, `--database<`*`db-name`*`>`	Runs Data Processing using the specified Hive database. If a Hive table is not specified, runs on all Hive tables in the Hive database.
`-e`, `--runEnrichment`	Runs the Data Enrichment modules (except for the modules that never automatically run during the sampling phase).
`-h`, `--help`	Displays usage information.
`-kryo`, `--kryoModeFlag`	Activates `kryoMode` for an optimized serialization. This should be tested on specific data sets.
`-m`, `--maxRecords` <*num*>	Sets maximum number of records to process. Overrides the CLI script's configuration setting.
`-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.
`-nr`, `--nonRandomizedCollectionNameFlag`	Does not randomize the data set names. This flag is intended for specific testing purposes.

CLI flag	Description
`-p`, `--collectionPrefix` <*prefix*>	Specifies the name prefix for data sets. Overrides the script configuration setting.
`-perf`, `--perfDataCollection`	Used only for Oracle internal use.
`-t`, `--table` <*name*>	Runs data processing on the specified Hive table. If a Hive database is not specified, assumes the default database set in the script configuration. Note that the table is skipped in these cases: it does not exist, is empty, or has the table property `skipAutoProvisioning` set.
`-v`, `--versionNumber`	Prints the version number of the current iteration of the Data Processing component within Big Data Discovery.
`-wl`, `--whiteList` <*wl_file*>	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.

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.

Both lists are optional when running the DP CLI. For example, if you manually run the DP CLI with the `--table` flag to process a specific table, you do not have to specify the lists.

Default lists are provided in the DP CLI package:

- `cli_whitelist.txt` is the default whitelist name (you can use your own name for this file).
- `cli_blacklist.txt` is the default blacklist name (you can use your own name for this file).

Both default lists are essentially empty — they 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
```

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. This means that the lists are essentially empty. You must edit the whitelist file to include at least one regular expression that specifies the tables to be ingested. Similarly, to exclude any tables, edit the blacklist.

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.*
```

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 creates 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 value must be enclosed in quotes. The default value is `"0 0 * * *"`, which 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/BDD1.0/dataprocessing/edp_cli/work
/detector.lock
  -c "cd /localdisk/Oracle/Middleware/BDD1.0/dataprocessing/edp_cli && .
/data_processing_CLI -d default
  -wl /localdisk/Oracle/Middleware/BDD1.0/dataprocessing/edp_cli/config/cli_whitelist.txt
  -bl /localdisk/Oracle/Middleware/BDD1.0/dataprocessing/edp_cli/config/cli_blacklist.txt
  -mwt 1800 >> /localdisk/Oracle/Middleware/BDD1.0/dataprocessing/edp_cli/work/detector.log 2>&1"
```

You can modify these settings (such as the time schedule). In addition, be sure to monitor the size of the `detector.log` file.

Chapter 5

Data Processing Logging

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

Logging configuration

Data Processing has a default configuration file, `logging.properties`, that sets the logging properties.

By default, the `logging.properties` file is located in the `/user/bdd/edp/lib` directory. The `hdfsEdpLibPath` property in the `data_processing_CLI` file controls the location of this file in HDFS.

The file has the following properties:

Logging property	**Description**
handlers	A comma-delimited list of handler class names that are added to the root Logger. The default handlers are `java.util.logging.FileHandler` and `java.util.logging.ConsoleHandler` (with a default level of `INFO`).
`java.util.logging.FileHandler.level`	Sets the log level for all `FileHandler` instances. The default log level is `FINE`.

Logging property	Description
`java.util.logging.FileHandler.pattern`	The log file name pattern. The default is `%t/edpLog%u%g.log` which means that the file is named `edpLog%u%g.log` where: • `%u` is a unique number to resolve conflicts between simultaneous Java processes. • `%g` is the generation number to distinguish between rotating logs. `%t` specifies the system temporary directory as the location in which the log files are stored.
`java.util.logging.FileHandler.limit`	The maximum size of the file, in bytes. If this is 0, there is no limit. The default is 1000000 (which is 1 MB). Logs larger than 1MB roll over to the next log file.
`java.util.logging.FileHandler.count`	The number of log files to use in the log file rotation. The default is 10000 (which produces a maximum of 10,000 log files).
`java.util.logging.FileHandler.formatter`	The class name of the Formatter to use for the `FileHandler` instances.
`java.util.logging.ConsoleHandler.level`	Sets the default log level for all `ConsoleHandler` instances.
`java.util.logging.FileHandler.append`	Specifies whether the `FileHandler` should append onto any existing files (defaults to `false`).
`java.util.logging.ConsoleHandler.formatter`	The class name of the Formatter to use for the `ConsoleHandler` instances.
`java.util.logging.SimpleFormatter.format`	Specifies the format to use for log messages. For details on the format syntax, see: *http://docs.oracle.com/javase/7/docs/api/java/util/logging/SimpleFormatter.html*

`com.oracle.eid = FINE`
`com.oracle.endeca = FINE`
`com.oracle.endeca.pdi = INFO`
	Sets the default logging level for the Big Data Discovery loggers.

Logging property	Description
`org.eclipse.jetty = WARNING org.apache.spark.repl.` `SparkIMain$exprTyper = INFO org.apache.spark.repl.SparkILoop` `$SparkILoopInterpreter = INFO`	Sets the default logging level for the Spark and Jetty loggers.

For details on the `FileHandler` settings, see *http://docs.oracle.com/javase/7/docs/api/java/util/logging/FileHandler.html*

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.

Data Processing logging

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

Location of the log files

Each run of Data Processing produces a new log file into the OS `temp` directory of each machine that is involved in the Data Processing job. The Data Processing log files are located on each node that has been involved in a Data Processing job. These include:

- The client that started the job (which could be nodes running the DP CLI or nodes running Studio)
- An Oozie (YARN) worker node
- Spark worker nodes

The logging location on each node is defined by the `edpJarDir` property in the `data_processing-CLI` file. By default, this is the `/opt/bdd/edp/data` directory.

Log files

The Data Processing log files are named `edpLog*.log`. The naming pattern is set in the `logging.properties` configuration. The default pattern is `edpLog%u%g.log`, where `%u` is a unique number to resolve conflicts between simultaneous Java processes and `%g` is the generation number to distinguish between rotating logs. The generation number is rotated, thus the latest run of Data Processing will be generation number 0. The configuration defaults produce 10,000 log files with a maximum file size of 1MB. Logs larger than 1MB roll over to the next log file.

A sample error log message is:

```
[2015/01/15 14:14:15] INFO: Starting Data Processing on Hive Table: default.claims
[2015/01/15 14:14:15] SEVERE: Error runnning EDP
java.lang.Exception Example Error Log Message
      at com.oracle.endeca.pdi.EdpMain.main(EdpMain.java:38)
      ...
```

Finding the Data Processing logs

When a client launches a Data Processing workflow, an Oozie job is created to run the actual Data Processing job. This job is run by an arbitrary node in the CDH cluster (node is chosen by YARN). To find the Data Processing logs, you should track down this specific cluster node using the Oozie Job ID. The Oozie Job ID is printed out to the console when the DP CLI runs, or you can find it in the Studio logs.

To find the Data Processing logs:

1. Go to the Oozie Web UI and find the corresponding job using the Oozie Job ID.
2. Click on the job to bring up detailed Oozie information.
3. Under the **Actions** pane, click the **DataProcessingJavaTask** action.
4. In the **Action Info** tab of the **Action** pane, find the **External ID**. The external ID matches a YARN Job ID.
5. Go to the **YARN HistoryServer Web UI** and find the corresponding job using the Oozie External ID. To do so:
    1. Browse the Cloudera Manager and click the YARN service in the left pane.
    2. In the **Quick Links** section in the top left, click **HistoryServer Web UI**.

6. Click the job to bring up detailed MapReduce information. The Node property indicates which machine ran the Data Processing job.
7. Log into the machine and go to the Data Processing directory on the cluster. By default, this is the `/opt/bdd/edp/data` directory. All the logs for Data Processing should reside in this directory.
8. To find a specific log, you may need to use `grep` (or other similar tool) for the corresponding workflow information.

Useful CDH logs

There are some CDH log files that may contain valuable information for debugging issues with the Data Processing component of Big Data Discovery.

YARN logs

To find the Data Processing logs in YARN:

1. Go to the Oozie Web UI and find the corresponding job using the Oozie Job ID.
2. Click the job to bring up detailed Oozie information.
3. Under the **Actions** pane, click the **DataProcessingJavaTask** action.
4. In the **Action Info** tab of the **Action** pane, find the External ID. The external ID matches a YARN Job ID.
5. Go to the **YARN HistoryServer Web UI** and find the corresponding job using the Oozie External ID. To do so:
    1. Browse the Cloudera Manager and click the YARN service in the left pane.
    2. In the **Quick Links** section in the top left, click **HistoryServer Web UI**.
6. Click the job to bring up detailed MapReduce information.
7. Click the **Map** task type to go to the **Map Tasks** page for the job.
8. Click the Map task. There should be only one Map task on this page.
9. Click the **logs** link. This displays a page with some logging information and links to the `stdout` and `stderr` full logs for the Map task.
10. In either the `stderr` or `stdout` log type sections, go to the **Click here for the full log** link. This displays the full log for the selected log type.

The `stdout` log lists the Data Processing operation type that was invoked for the workflow, as shown in this abbreviated entry:

```
>>> Invoking Main class now >>>

Main class        : com.oracle.endeca.pdi.EdpOozieJobReceiver
Arguments         :
                    PROVISION_DATASET_FROM_HIVE
                    {
  "@class" : "com.oracle.endeca.pdi.client.config.EdpEnvConfig",
  "endecaServer" : {
    "@class" : "com.oracle.endeca.pdi.concepts.EndecaServer",
    "host" : "web04.us.example.com",
    "wsPort" : 7001,
    "contextRoot" : "/endeca-server",
    "ssl" : false
  },
...
```

The Arguments field lists the operation type in the Data Processing workflow:

- `APPLY_TRANSFORM_TO_DATASET` — updates a project data set by applying a transformation to it.
- `APPLY_TRANSFORM_TO_DATASOURCE` — creates a new BDD data set (and a corresponding Hive table) by applying a transformation to an existing project data set and saving the transformed data to the new Hive table. This operation is also called forking the data set.
- `CLEANUP_DATASETS` — deletes any BDD data set that does not have a corresponding source Hive table.
- `CLEANUP_ORPHANED_DATASETS` — deletes any BDD data set that was generated from a Studio project, and the project no longer exists.
- `PROVISION_DATASET_FROM_HIVE` — creates a new BDD data set from a Hive table.

Spark worker logs

Inside of the main Data Processing log, you can find several references to a specific Spark job's Application ID. They are of the form `app-TIMESTAMP-INCREMENTALCOUNTER`. This Application ID is necessary to find the corresponding Spark workers.

You can display a specific Spark worker log by using the **Spark Web UI**. To do so, select the Spark job on the Spark Web UI and find each of the Spark workers used to run the Data Processing job. Here you have access to the `stdout` and `stderr` from each worker. The logs for each Spark worker are similar but should differ slightly because they are running on separate machines.

Chapter 6

Data Enrichment Modules

This section describes the Data Enrichment modules.

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. (You cannot configure which modules do or do not run.) However, 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.

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.
- Noun Group: 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.

The Data Enrichment modules support:

- English (UK/US)
- French
- German
- Italian
- Portuguese (Brazil)
- Spanish

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.

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:

- `<colname>_entity_person`
- `<colname>_entity_loc`
- `<colname>_entity_org`

In addition, the Transform API has the following functions that are wrappers around the Name Entity extractor to return single values from the input text:

- `getPersonEntities` returns the name of each person identified in the input.
- `getOrganizationEntities` returns the name of each organization identified in the input.
- `getLocationEntities` returns the name of each location identified in the input.

Example

Assume the following input text:

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

The output might be:

```
ext__entity_loc: New York City
ext_entity_org: Acme Corporation
ext_entitY_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 sampling phase of a Data Processing workflow.

Output

The output of this plugin is an ordered list of phrases (single- or multi-word) which 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 `<colname>_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:

- American English
- French
- German

- Italian
- Portuguese (Brazil)
- Spanish

Configuration options

This module never runs automatically during a Data Processing workflow.

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

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 `<colname>_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.

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:

- `<colname>_sub_sent_neg` (for negative phrases)
- `<colname>_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.

Some valid formats are:

- City + State
- City + State + Postalcode
- City + Postalcode
- Postalcode + 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 postalcode 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
```

Sub-GeoTaggers

The Address GeoTagger module is a wrapper around these sub-GeoTaggers that can be run separately:

- City GeoTagger — returns the same information as the Address GeoTagger.
- Region GeoTagger — returns geographical information for a region, which consists of the geocode, the region name, the region ID, and the country code.

- SubRegion GeoTagger — returns geographical information for a sub-region, which consists of the geocode, the region name, the region ID, the sub-region name, the sub-region ID, and the country code.
- Country GeoTagger — returns geographical information for a country, which consists of the geocode and the country code.

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.

For Transform operations, there are two flags (`preferred_level` and `isStrict`) to control the input and output.

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:

- `<colname>_geo_geocode` — the latitude and longitude values of the address (such as "42.35843 -71.05977").
- `<colname>_geo_city` — corresponds to a city (such as "Boston").
- `<colname>_geo_country` — the country code (such as "US").
- `<colname>_geo_postcode` — corresponds to a postal code, such as a zip code in the US (such as "02117").
- `<colname>_geo_region` — corresponds to a geographical region, such as a state in the US (such as "Massachusetts").
- `<colname>_geo_regionid` — the ID of the region in the GeoNames database (such as "6254926" for Massachusetts).
- `<colname>_geo_subregion` — corresponds to a geographical sub-region, such as a county in the US (such as "Suffolk County").
- `<colname>_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:

- `<colname>_geo_geocode` — the latitude and longitude values of the address (such as "40.71427 -74.00597 ").
- `<colname>_geo_city` — corresponds to a city (such as "New York City").
- `<colname>_geo_region` — corresponds to a region, such as a state in the US (such as "New York").
- `<colname>_geo_regionid` — the ID of the region in the GeoNames database (such as "5128638 " for New York).
- `<colname>_geo_postcode` — corresponds to a postal code, such as a zip code in the US (such as "02117").
- `<colname>_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, postal code, 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 postal code.

Output

The output of this module consists of these attribute names and values:

- `<colname>_geo_city` — corresponds to a city (such as "Boston").
- `<colname>_geo_country` — the country code (such as "US").
- `<colname>_geo_postcode` — corresponds to a postal code, such as a zip code in the US (such as "02117").
- `<colname>_geo_region` — corresponds to a geographical region, such as a state in the US (such as "Massachusetts").
- `<colname>_geo_regionid` — the ID of the region in the GeoNames database (such as "6254926" for Massachusetts).
- `<colname>_geo_subregion` — corresponds to a geographical sub-region, such as a county in the US (such as "Suffolk County").
- `<colname>_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 `<colname>_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 `<colname>_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 `<colname>_lang`.

Chapter 7

Data Model in Big Data Discovery

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.

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

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.

By default, all attributes are single-assign. To make an attribute multi-assign, you must update the attribute configuration.

Primary keys

In the Dgraph data model, a primary-key attribute is used, for identifying records and collections (data sets). This topic provides a summary of each of these primary keys.

Collection primary key

For the Dgraph to identify each collection, the collection (at creation time) must have an attribute configured as its primary key. The primary-key attribute must be created with the following properties set to `true`:

- `IsSingleAssign`
- `IsUnique`
- `IsRequired`

The `IsUnique` property assures that no two records can have the same value setting for the primary-key attribute.

Record primary key

For the Dgraph to identify a record, it must have an assignment from exactly one primary-key attribute. This assignment is known as a record primary key, or record spec. This record primary key is the same attribute as the collection primary-key attribute. In other words, for a record to belong to a given collection, its primary-key attribute must be the same as the collection primary key.

Because the primary-key attribute must be single assign and unique, the attribute may be assigned only once in any record and a given attribute value may be assigned to at most one record (that is, no two records in a Dgraph index have the same value for this attribute).

Attribute data types

The attribute type identifies the type of data allowed for the Dgraph 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.
`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.

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:

- Arabic — `ar`
- Basque — `eu`
- Belarusian — `be`
- Bosnian — `bs`
- Bulgarian — `bg`
- Catalan — `ca`
- Chinese, simplified — `zh_CN`
- Chinese, traditional — `zh_TW`
- Croatian — `hr`
- Czech — `cs`
- Danish — `da`
- Dutch — `nl`
- English, American — `en`
- English, British — `en_GB`
- Estonian — `et`
- Finnish — `fi`
- French — `fr`
- French, Canadian — `fr_ca`
- Galician — `gl`
- German — `de`
- Greek — `el`
- Hebrew — `he`
- Hungarian — `hu`
- Indonesian — `id`
- Italian — `it`
- Japanese — `ja`
- Korean — `ko`
- Latvian — `lv`
- Lithuanian — `lt`
- Macedonian — `mk`

- Malay — `ms`
- Norwegian Bokmal — `nb`
- Norwegian Nynorsk — `nn`
- Persian — `fa`
- Polish — `pl`
- Portuguese — `pt`
- Portuguese, Brazilian — `pt_BR`
- Romanian — `ro`
- Russian — `ru`
- Serbian, Cyrillic — `sr_Cyrl`
- Serbian, Latin — `sr_Latn`
- Slovak — `sk`
- Slovenian — `sl`
- Spanish — `es`
- Swedish — `sv`
- Thai — `th`
- Turkish — `tr`
- Ukrainian — `uk`
- Valencian — `vc`
- Vietnamese — `vn`
- 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 `mdex-property_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 indices

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 it 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.

All dictionary files are stored in the index directory.

Chapter 8

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

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 (such as, importing) 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 loading into BDD

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.

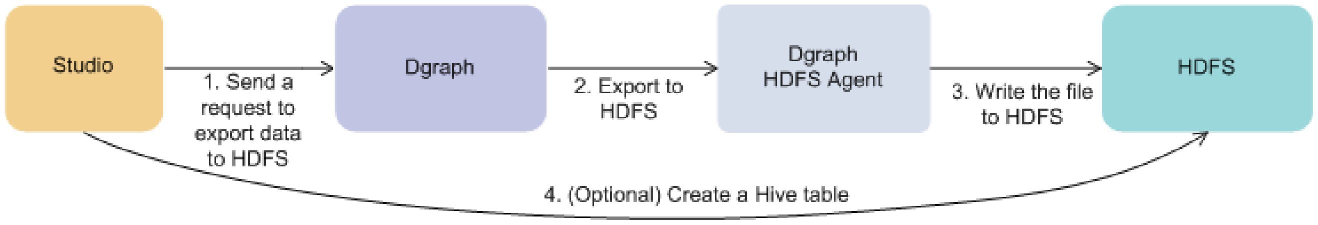
Exporting data from Studio into HDFS

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 or CSV file to either an external directory on your computer, or to HDFS. For details on the operation, see the *Data Exploration and Analysis Guide*.

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:


Studio
1. Send a request to export data to HDFS
Dgraph
2. Export to HDFS
Dgraph HDFS Agent
3. Write the file to HDFS
HDFS
4. (Optional) Create a Hive table

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 the data set named default_edp_999. (Note that a data set is called a collection in the Dgraph). The messages have been edited for readability:

```
New import request received: Collection name: default_edp_999,
   location: /user/bdd/.dataIngestSwamp/default_edp_999, user name: yarn
Finished reading 57076 records for Collection name: default_edp_999,
   location: /user/bdd/.dataIngestSwamp/default_edp_999, user name: yarn
fetchMoreRecords Collection name: default_edp_999,
   location: /user/bdd/.dataIngestSwamp/default_edp_999, user name: yarn
createBulkIngester default_edp_999
Starting ingest for: Collection name: default_edp_999,
   location: /user/bdd/.dataIngestSwamp/default_edp_999, user name: yarn
sendRecordsToIngester 57076
fetchMoreRecords Collection name: default_edp_999,
   location: /user/bdd/.dataIngestSwamp/default_edp_999, user name: yarn
closeBulkIngester
fetchMoreRecords Collection name: default_edp_999,
   location: /user/bdd/.dataIngestSwamp/default_edp_999, user name: yarn
Ingest finished with 57076 records committed and 0 records rejected.
   Status: INGEST_FINISHED. Request info: Collection name: default_edp_999,
   location: /user/bdd/.dataIngestSwamp/default_edp_999, user name: yarn
update dataSetInventory request result:
   ... <ingest:numRecordsAffected>1</ingest:numRecordsAffected> ...
```

In the example:

1. The Data Processing workflow has written a set of Avro files in the `/user/bdd/.dataIngestSwamp/default_edp_999` directory in HDFS.
2. The Dgraph HDFS Agent reads 57,076 records from the HDFS directory.
3. The `createBulkIngester` operation is used to instantiate a Bulk Load ingester instance for the `default_edp_999` collection.
4. The `sendRecordsToIngester` operation sends the 57,076 records to the Dgraph's ingester.
5. The Bulk Load instance is closed with the `closeBulkIngester` operation.
6. The `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.
7. The Dgraph HDFS Agent updates the `ingestStatus` attribute of the DataSet Inventory with the final status of the ingest operation. The `numRecordsAffected=1` response indicates that the DataSet Inventory record update was successful.

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.

Index