Oracle® Big Data Discovery

Extensions Guide

Version 1.0.0 • Revision A • March 2015

ORACLE

Copyright and disclaimer

Table of Contents

Preface

Oracle Big Data Discovery is a set of end-to-end visual analytic capabilities that leverage the power of 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 available extension and customization options for Oracle Big Data Discovery. It describes how to install and use the Studio's Component SDK that lets you create custom Security Managers, develop custom Studio components, and work with QueryFunction Java classes. Additionally, this guide has a section on how to use the custom transform functions (also known as the Transform API for Big Data Discovery).

Who should use this guide?

This guide is intended for developers who want to use the Studio Component SDK (for creating custom components, or custom Security Managers). This guide also is for business analysts and developers who want to learn how to use the custom transform functions (Transform API).

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.
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*.

Part I

Using the Component SDK

Chapter 1

Installing and Configuring the Component SDK

The Component SDK supports custom development for components and data security.

About the Component SDK

The Component SDK allows developers to extend Studio by creating and deploying custom Security Managers and components.

A Security Manager is used to restrict access to specific data.

A custom component is used to visualize data in Studio. Once you deploy a custom component, it can be added to a project page.

As part of developing custom components, you can also create custom `QueryFunctions`, used to retrieve and display data on a component.

To see the full generated documentation for the Component SDK, see the *Component SDK API Reference (Javadoc)*.

Requirements for using the Component SDK

Before using the Component SDK, make sure that you meet the system and skill set requirements.

Required knowledge and skills

In order to work with the Component SDK, you should be familiar with Java development and JavaScript.

Components are extensions of a custom version of the Java Portlet class, so to develop a custom component, you should also have some understanding of Java portlets and the Portlet specification.

The Component SDK generates Eclipse projects, so it also helps to be familiar with Eclipse.

Supported platforms

While Big Data Discovery is always deployed on a Linux system, you can use the Component SDK from either a Windows or Linux system.

There are .bat and .sh versions of each of the Component SDK scripts.

Software requirements

All Component SDK work requires the following:

- Eclipse. You must use a version that supports JDK 1.7.
- JDK 1.7 or above
- Apache Ant 1.8.4 or higher, to build your custom items

For custom components, you may also need:

Software or License	Description
Ext JS	While Ext JS is not required, and the sample component provided with the Component SDK does not use it, most Big Data Discovery components were developed using *Ext JS 3.4*. Big Data Discovery does not include a license for Ext JS. If you want to use Ext JS for custom component development, you must obtain your own copy of it.
YUI Compressor 2.4.8	By default, when you compile a custom component, JavaScript minification is not used. While components do build successfully without JavaScript minification, for performance purposes you may want to enable it. If you enable minification, then files in the `docroot/js` directory of your custom components are minified. In order to be able to use minification to build components, you must obtain the .jar file for version 2.4.8 of YUI Compressor. The file is available at *https://github.com/yui/yuicompressor/releases/download/v2.4.8/yuicompressor-2.4.8.jar*.
JUnit	If you are planning to create unit tests for your custom components, you will need to first obtain `junit.jar`. The Component SDK can use JUnit for unit tests, but does not come with the `junit.jar` file.

Installing the Component SDK

The Component SDK is contained in a .zip file in the Big Data Discovery Media Pack.

To install the Component SDK:

1. From the Big Data Discovery Media Pack, download the Component SDK .zip file (`component-sdk-<versionNumber>.zip`).
2. Extract the Component SDK .zip file to a separate directory.

   The directory path to the Component SDK cannot contain spaces.

Once you have installed the Component SDK, you can continue with your custom development.

For information on developing custom Security Managers, see *Developing a Custom Security Manager on page 11*.

For information on developing custom components, see *Developing Custom Components on page 14*.

Preparing your system for Component SDK development

After installing the Component SDK, before you can start development, you must complete some initial preparation on your system.

This includes:

- Extracting the Studio .ear file and portal .war file
- Configuring build files to point to the directories for these extracted files
- Optionally, enabling JavaScript minification for custom components.

  If minification is enabled, then files in the `docroot/js` directory of custom components are minifiied.

To prepare your system for custom component development:

1. Extract the Studio .ear file and portal .war file:
   (a) From the Big Data Discovery Media Pack, download the .ear file.
   (b) Extract the .ear file to a directory on your machine.
   (c) From that directory, extract the file `endeca-portal.war` to a directory within the extracted .ear file directory.

   For example, if the .ear file is extracted to `/bdd_ear`, the contents of the extracted .war file might be in `/bdd_ear/portal/`.
2. Next, in the Component SDK, create and configure the build properties files:
   (a) Go to the `components` directory of the Component SDK.
   (b) In the `components` directory, create a file called `build.<user>.properties`, where `<user>` is the user name that you use to log in to the current machine.

   For example, if your user name is jsmith, then you would create a file called `build.jsmith.properties`.
   (c) Add the following properties to `build.<user>.properties`:

```
portal.base.dir=<extracted .ear file directory>
```

```
app.server.lib.global.dir=<extracted .ear file directory>/APP-INF/lib
app.server.portal.dir=<extracted portal .war file directory>
war.output.dir=<directory for generated components>
```

The `war.output.dir` setting indicates where the build process should place the .war file that it generates when you compile a custom component. This can be any directory on your system.

So for example, if:

- You extracted the .ear file to a directory called `/bdd_ear`
- You extracted the portal .war file to a `portal` directory in `/bdd_ear`
- You want the generated .war files for custom components to be placed in `/generated_components`

the settings would be:

```
portal.base.dir=/bdd_ear
app.server.lib.global.dir=/bdd_ear/APP-INF/lib
app.server.portal.dir=/bdd_ear/portal
war.output.dir=/generated_components
```

(d) In the `components` directory, create a file called `build.shared.properties`.

(e) In `build.shared.properties`, add the following property:

```
portal.base.dir=<extracted .ear file directory>
```

3. To enable JavaScript minification when building custom components:

(a) If you haven't already, obtain the required YUI Compressor .jar file. See *Requirements for using the Component SDK on page 8*.

(b) In the `components` directory of the Component SDK, update `build.<user>.properties` to add the following property:

```
yui.compressor.jar=<path to YUI Compressor .jar file>
```

4. In Eclipse, create the following Eclipse classpath variables:

Name	Path
`DF_GLOBAL_LIB`	Path to the application server global library, which is: `<extracted .ear file directory>/APP-INF/lib`
`DF_PORTAL_LIB`	Path to the Web application library, which is: `<extracted portal .war file directory>`

Chapter 2

Developing a Custom Security Manager

Using the Component SDK, you can create a custom Security Manager to customize how Big Data Discovery filters data that is displayed to users.

Creating and implementing a new Security Manager

The Component SDK includes a batch script for creating a new Security Manager.

To create a new Security Manager project:

1. From a command prompt, change to the `components/endeca-extensions` directory in the Component SDK.
2. Run the appropriate version of the `create-bddsecuritymanager` command.

   For Linux:

   ```
   ./create-bddsecuritymanager.sh <securityManagerName>
   ```

   For Windows:

   ```
   create-bddsecuritymanager.bat <securityManagerName>
   ```

   Where *<securityManagerName>* is the name you want to use for the security manager. For example:

   ```
   ./create-bddsecuritymanager.sh restrict-region-data
   ```

   The name cannot have spaces.

   This command creates a *<securityManagerName>* directory in `bddsecuritymanager`.

   This directory is an Eclipse project that you can import directly into Eclipse.

   It also contains a sample implementation that can help you understand how the Security Manager is used.

   Note: The sample implementation illustrates one way to use the API. The sample is not intended to provide a recommended design pattern for a production application.

3. Your Security Manager must implement the `applySecurity` method.

```
public void applySecurity(PortletRequest request, MDEXState mdexState, Query query) throws
BddSecurityException;
```

The Query class in this signature is `com.endeca.portal.data.Query`. This class provides a simple wrapper around a Conversation Service request.

Security Manager interface

The `com.endeca.portal.data.security.BddSecurityManager` interface represents a Security Manager capable of applying record-level security filters for BDD.

For additional details about `BddSecurityManager`, see the *Component SDK API Reference*.

Class Summary Item	Item Value or Description
Abstract base class	`com.endeca.portal.data.security.AbstractBddSecurityManager`
Concrete implementation class	`com.endeca.portal.data.security.AttributeAclSecurityManager`
Implementation behavior	The `AttributeAclSecurityManager` implementation filters records in a data set (collection) according to Access Control List (ACL) multi-assign attributes which have been added to each record during a data ingest. The class assumes that these attributes are named: • `__allow_user` for user-permissions • `__allow_group` for group-permissions • `__allow_role` for role-permissions This implementation requires a collection/data-set to have all three of these attributes if it is to be secured, even if one or more of them is not used. It is also required that each of these attributes must be multi-assign string attributes (i.e., type=mdex:string and isSingleAssign=false). Each record is filtered according to the name of the user and those of the groups/roles held by that user, the names of which need to be assigned to the above attributes.

The SDK package contains a `SampleBddSecurityManager.java` that is based on `AttributeAclSecurityManager`. The file is included in the `bddsecuritymanager.zip`, which is in the `components/endeca-extensions` directory in the Component SDK.

Building and deploying a new Security Manager

Before you can use your custom Security Manager, you must deploy it to Studio. To do this, you generate a .jar file for it, then add the .jar file to the Studio .ear file.

To build and deploy a custom Security Manager:

1. From the `<securityManagerName>-mdexsecuritymanager` directory you created for your new Security Manager, run the Ant build script.

   This generates a .jar file named `<your-security-manager-name>-bddsecuritymanager.jar`, and places it in the Security Manager project directory.
2. Add the .jar file to the `app-inf/lib` directory within the deployed .ear file for Studio.
3. Redeploy the .ear file.

Configuring Studio to use a different Security Manager

In order to for Studio to use your Security Manager, you must configure Studio to pick up and use the new class.

To configure Studio to use a different Security Manager:

1. On the **Control Panel** menu, click **Studio Settings**.
2. Change the value of `df.mdexSecurityManager` to the full name of your class, similar to following example:

   ```
   df.bddSecurityManager = com.endeca.portal.extensions.YourSecurityManagerClass
   ```
3. Click **Update Settings**.
4. To have the change take effect, restart Studio. You may also need to clear any cached user sessions.

Chapter 3

Developing Custom Components

The most common use of the Component SDK is to create and deploy custom components.

Generating the Eclipse project for the component

The Component SDK includes a script to generate an Eclipse project for a new component.

New components are extensions of the `EndecaPortlet` class, which is in turn an extension of the basic Java Portlet class.

To create a new component:

1. At a command prompt, change to the `components/portlets` directory in the Component SDK.
2. Run the appropriate `.sh` or `.bat` version of the `create` command:

   For example:

   ```
   create.sh <componentName> "<componentDisplayName>"
   ```

   Where:

Parameter	Description
<componentName>	The name of the component. The component name: • Must be all lower case. • Cannot have spaces. • Cannot include the string `-ext`, because it causes confusion with the ext plug-in extension. For example, `my-component-extension` would not be a valid name.
<componentDisplayName>	The display name for the component. The display name can have spaces, but if it does, it must be enclosed in quotation marks.

   For example:

```
create.sh my-test "My New Test Component"
```

The script creates in the `portlets` directory a new directory for the new component.

The directory is the component name, with `endeca-` pre-pended and `-portlet` appended automatically. For example, if you set the name to `my-test`, the directory is named `endeca-my-test-portlet`.

This directory is an Eclipse project that you can import directly into Eclipse.

3. Import the project into Eclipse.

   If your components depend on shared library projects located within the `/shared` directory, import those as well.

   Note that it takes some time for projects to build after they are imported.

After you generate and import the component project, you can begin the actual component development.

Obtaining query results for components

When developing a component, use the `QueryState` and `QueryResults` classes to request and receive data from data sets.

To specify the types of results the component needs, you must add the relevant `QueryConfigs` to the `QueryState`. For example:

```
QueryState query = getDataSource(request).getQueryState();
CollectionBaseView defaultBaseView = EndecaPortletUtil.getDefaultCollection(request);
query.addFunction(new NavConfig(), defaultBaseView, request.getLocale());
QueryResults results = getDataSource(request).execute(query);
```

You can then get the underlying Conversation Service API results in order to obtain the data required by your component.

```
Results discoveryResults = results.getDiscoveryServiceResults();
```

Before executing the query, you can also make other local modifications to your query state by adding filters or configurations to your query. For example:

```
String viewKey = request.getParameter(VIEW_KEY_PARAM);
DataSource ds = getDataSource(request);
QueryState query = ds.getQueryState();
SemanticView sView = ds.getCollectionOrSemanticView(viewKey, request.getLocale());
query.addFunction(new ResultsConfig(), sView, request.getLocale());
ExpressionBase expression = getDataSource(request).parseLQLExpression("Region = 'Midwest'");
query.addFunction(new SelectionFilter(expression), sView, request.getLocale());
QueryResults results = getDataSource(request).execute(query);
```

To persist `QueryState` changes to the user's session, which also updates the associated components, use `setQueryState`. For example:

```
String viewKey = request.getParameter(VIEW_KEY_PARAM);
DataSource ds = getDataSource(request);
QueryState query = ds.getQueryState();
SemanticView sView = ds.getCollectionOrSemanticView(viewKey, request.getLocale());
query.addFunction(new ResultsConfig(), sView, request.getLocale());
ExpressionBase expression = getDataSource(request).parseLQLExpression("Region = 'Midwest'");
query.addFunction(new SelectionFilter(expression), sView, request.getLocale());
ds.setQueryState(query);
```

For details on the `QueryConfig` and `QueryFunction` classes, see *Working with QueryFunction Classes on page 18*, and the *Component SDK API Reference*.

Building a component

After completing the component development, you set the build properties, then build the component in Eclipse.

To build a component:

1. Before building the component, you need to make sure the build properties are set correctly. Open the `build.xml` in the root directory of the component.

   By default, the build properties are:

   ```
   <property name="shared.libs" value="endeca-common-resources,endeca-discovery-taglib" />
   <property name="endeca-common-resources.includes" value="**/*" />
   <property name="endeca-common-resources.excludes" value="" />
   ```

   These properties are used as follows:

Property	Description
`shared.libs`	Controls which projects in the `shared/` directory to include in the component. These shared projects are compiled and included as `.jar` files where appropriate.
`endeca-common-resources.includes`	Controls which files in the `shared/endeca-common-resources` project are copied into the component. The default value is `"**/*"`, indicating that all of the files are included. These files provide AJAX enhancements (`preRender.jspf` and `postRender.jspf`).
`endeca-common-resources.excludes`	Controls which files from the `shared/endeca-common-resources` project are excluded from the component. By default, the value is `""`, indicating that no files are excluded. If your component needs to override any of these files, you must use this build property to exclude them. If you do not exclude them, your code will be overwritten.

   You can specify the `includes` and `excludes` properties for any shared library. For example:

   ```
   <property name="endeca-discovery-taglib.includes" value="**/*" />
   <property name="endeca-discovery-taglib.excludes" value="" />
   ```

2. Once the build properties are set, then in your Eclipse project, open the `build.xml` file.
3. If the project is not configured to build automatically, then in the outline view, right-click the deploy task and select **Run as...>Ant Build**.

The build process generates the component .war file, and places it in the output directory you specified. The .war file has the same name as the component.

Deploying and removing custom components

Once you have built the component .war file, you can add the component to a Big Data Discovery instance. You can also remove a component.

To deploy and remove components:

1. To deploy a custom component:
   (a) Open the Studio .ear file.
   (b) Add the component .war file to the root of the .ear file, with the other component .war files.
   (c) In the `meta-inf` directory of the .ear file, open `application.xml`
   (d) Add an entry for the new component, then save the file.

   For example:

```
<module>
    <web>
        <web-uri>my-new-component-portlet.war</web-uri>
        <context-root>/eid/my-new-component-portlet/</context-root>
    </web>
</module>
```

   (e) Redeploy the .ear file.
   (f) Restart Big Data Discovery.

   During the startup process, you can check the Big Data Discovery logs to confirm that the component loaded successfully.
2. After redeploying the .ear file, to test that the component was added successfully:
   (a) Log in to Big Data Discovery.
   (b) From within a Big Data Discovery project, click the add component option.

   Your component should be included in the list of available components.
   (c) Drag and drop the new component onto the page.
3. To remove a component:
   (a) Open the Big Data Discovery .ear file.
   (b) Remove the component .war file.
   (c) In `meta-inf/application.xml`, remove the entry for the component.
   (d) Redeploy the .ear file.

Chapter 4

Working with QueryFunction Classes

When developing custom components, you can use the provided `QueryFunction` classes to filter and query data. You can also create and implement your own `QueryFunction` classes.

Provided QueryFunction filter classes

Big Data Discovery provides the following QueryFunction filter classes. Filters are used to change the current query state.

The available filter classes are:

- `DataSourceFilter`
- `RefinementFilter`
- `NegativeRefinementFilter`
- `RangeFilter`, including the following date/time-specific range filters that extend `RangeFilter`:
  - `DateRangeFilter`
  - `TimeRangeFilter`
  - `DurationRangeFilter`
- `DateFilter`
- `LastNDateFilter`
- `GeoFilter`
- `SearchFilter`

In addition to the information here, for more details on the `QueryFunction` filter classes, see the *Component SDK API Reference*.

DataSourceFilter

Uses an EQL snippet to provide the filtering. `DataSourceFilter` refinements are not added to the **Selected Refinements** panel.

The available properties are:

Property	Description
`filterString`	The EQL snippet containing the filter information. For a `DataSourceFilter`, this would be the content of a `WHERE` clause for an EQL statement. For details on the EQL syntax, see the *EQL Reference*.

For example, to filter data to only show records from the Napa Valley region with a price lower than 40 dollars:

```
ExpressionBase expression = dataSource.parseLQLExpression("Region='Napa Valley' and P_Price<40");
DataSourceFilter dataSourceFilter = new DataSourceFilter(expression);
```

RefinementFilter

Used to filter data to include only those records that have the provided attribute values. `RefinementFilter` refinements are added to the **Selected Refinements** panel.

The properties for a `RefinementFilter` are:

Property	Description
`attributeValue`	String The attribute value to use for the refinement.
`attributeKey`	String The attribute key. Identifies the attribute to use for the refinement.
`sourceCollectionKey`	String The key of the data set. This is typically a long encoded value that starts with `default_edp`.

Property	Description
`multiSelect`	`AND |OR | NONE` For multi-select attributes, how to do the refinement if the filters include multiple values for the same attribute: • If set to `AND`, then matching records must contain all of the provided values. • If set to `OR`, then matching records must contain at least one of the provided values. • If set to `NONE`, then multi-select is not supported. Only the first value is used for the refinement. This setting must match the refinement behavior configured for the attribute in the data set. For information on using the **Views** page to view and configure the refinement behavior for an attribute, see the *Data Exploration and Analysis Guide*.

In the following example, the data is refined to only include records that have a value of 1999 for the Year attribute.

```
RefinementFilter refinementFilter = new RefinementFilter("1999", "Year", "default_edp_cc7ea");
```

NegativeRefinementFilter

Used to filter data to exclude records that have the provided attribute value. `NegativeRefinementFilter` refinements are added to the **Selected Refinements** panel.

The properties for a `NegativeRefinementFilter` are:

Property	Description
`attributeValue`	String The attribute value to use for the refinement.
`attributeKey`	String The attribute key. Identifies the attribute to use for the refinement.
`attributeType`	`BOOLEAN | STRING | DOUBLE | LONG | GEOCODE | DATETIME | TIME | DURATION` The type of value to use for the refinement. The default is `STRING`. If the attribute is a type other than string, then you must provide the type.

Property	Description
`attributeValueName`	String Optional. The value to display on the **Selected Refinements** panel for the refinement. If you do not provide a value for `attributeValueName`, then the **Selected Refinements** panel displays the value of `attributeValue`.
`ancestors`	Not supported.
`isAttributeSingleAssign`	Boolean. If set to `true`, then the attribute can only have one value. If set to `false`, then the attribute is multi-value. For information on using the **Views** page to see whether an attribute is multi-value, see the *Data Exploration and Analysis Guide*.
`sourceCollectionKey`	String The key of the data set. This is typically a long encoded value that starts with `default_edp`.

In the following example, the data is refined to only include records that do NOT have a value of Washington for the Region attribute. Because Region is a string attribute, no other configuration is needed.

```
NegativeRefinementFilter negativeRefinementFilter
= new NegativeRefinementFilter("Region", "Washington");
```

In the following example, the data is refined to only include records that do NOT have a value of 1997 for the P_Year attribute, which is a single-assign attribute. Because P_Year is not a string attribute, the attribute type LONG is specified.

```
NegativeRefinementFilter negativeRefinementFilter
= new NegativeRefinementFilter("P_Year", "1997", PropertyType.LONG,
   true, "default_edp_cc7ea760");
```

RangeFilter

Used to filter data to include only those records that have attribute values within the specified range. `RangeFilter` refinements are added to the **Selected Refinements** panel.

The properties for a `RangeFilter` are:

Property	Description
`attributeKey`	String The attribute key. Identifies the attribute to use for the filter.

Property	Description
`rangeOperator`	`LT | LTEQ |GT |GTEQ| BTWN |GCLT |GCGT | GCBTWN` The type of comparison to use. • `LT` - Less than • `LTEQ` - Less than or equal to • `GT` - Greater than • `GTEQ` - Greater than or equal to • `BTWN` - Between. Inclusive of the specified range values. • `GCLT` - Geocode less than • `GCGT` - Geocode greater than • `GCBTWN` - Geocode between
`rangeType`	`DECIMAL | INTEGER | DATE | GEOCODE | TIME | DURATION` The type of value that is being compared.
`value1`	Numeric The value to use for the comparison. For `BTWN`, this is the low value for the range. For the geocode range operators, the origin point for the comparison.
`value2`	Numeric For a `BTWN`, this is the high value for the range. For `GCLT` and `GCGT`, this is the value to use for the comparison. For `GCBTWN`, this is the low value for the range.
`value3`	Numeric Only used for the `GCBTWN` operator. The high value for the range.

In the following example, the data is refined to only include records where the value of P_Score is a number between 80 and 100:

```
RangeFilter rangeFilter
= new RangeFilter("P_Score", RangeType.INTEGER, RangeOperator.BTWN, "80", "100");
```

There are also date/time-specific range filters that extend `RangeFilter`:

- `DateRangeFilter`
- `TimeRangeFilter`
- `DurationRangeFilter`

DateFilter

Used to filter date values. Using a `DateFilter`, you can filter by subsets of the date/time value. For example, you can filter a date attribute to include all records with a specific year or specific month.

The properties for a `DateFilter` are:

Property	Description
`dateFilters`	A list of `DateFilterDimension` objects that represent the date filters to apply. Each `DateFilterDimension` object consists of: • `DatePart` constants identify each date part • Integer values to represent the values for each date part The filter only filters down to the most specific date part provided.

In the following example, the data is refined to only include records where SalesDate is June 15, 2006. The filter only provides the year, month, and day. Even if records have different hour-minute-second values for SalesDate, as long as they are within June 15, 2006, they still match this filter:

```
DateFilterDimension dfd = new DateFilterDimension();
dfd.addDatePartFilter(DatePart.YEAR, 2006);
dfd.addDatePartFilter(DatePart.MONTH, 6);
dfd.addDatePartFilter(DatePart.DAY_OF_MONTH, 15);
DateFilter dateFilter = new DateFilter("SalesDate", dfd);
```

LastNDateFilter

Used to filter the date to include records with a date attribute with a value in the last n years, months, or days.

The properties for a `LastNDateFilter` are:

Property	Description
`attributeKey`	The key name of the attribute.
`ticksBack`	The number of years, months, or days within which to include records in the results.
`datePart`	The date part to use for the filtering. The possible values are: • `YEAR` • `MONTH` • `DAY_OF_MONTH` • `HOUR` • `MINUTE` • `SECOND`

Property	Description
`sourceCollectionKey`	String The key of the data set. This is typically a long encoded value that starts with `default_edp`....

In the following example, the data is refined to only include records with SalesDate values from the last 3 years:

```
LastNDateFilter lastNDateFilter = new LastNDateFilter("SalesDate", 3, DatePart.YEAR);
```

GeoFilter

Used filter data to include records with a geocode value within a specific distance of a specific location.

The properties for a `GeoFilter` are:

Property	Description
`attributeKey`	The key name for the geocode attribute.
`rangeOperator`	The comparison operator.
`value1`	A geocode value to use as the starting point.
`radius`	The number of miles or kilometers within which to search.
`locationName`	The name of a location to use as the starting point.
`unit`	The unit of distance (mi or km) for the comparison.

SearchFilter

Used to filter the data to include records that have the provided search terms. `SearchFilter` refinements are added to the **Selected Refinements** panel.

The properties for a `SearchFilter` are:

Property	Description
`searchInterface`	String Either the name of the search interface to use, or the name of an attribute that is enabled for text search.
`terms`	String The search terms.

Property	Description
`matchMode`	`ALL` | `PARTIAL` | `ANY` | `ALLANY` | `ALLPARTIAL` | `PARTIALMAX` | `BOOLEAN` The match mode to use for the search.
`enableSnippeting`	Boolean Whether to enable snippeting. Optional. If not provided, the default is `false`.
`snippetLength`	Integer The number of characters to include in the snippet. Required if `enableSnippeting` is `true`. To enable snippeting, set `enableSnippeting` to `true`, and provide a value for `snippetLength`.

In the following example, the filter uses the "default" search interface to search for the terms "California" and "red". The matching records must include all of the search terms. Snippeting is supported, with a 100-character snippet being displayed.

```
SearchFilter.Builder builder = new SearchFilter.Builder("default", "California red");
builder.matchMode(MatchMode.ALL);
builder.enableSnippeting(true);
builder.snippetLength(100);
SearchFilter searchFilter = builder.build();
```

Provided QueryConfig functions

Studio provides the following `QueryConfig` functions, used to manage the results returned by a query. These are more advanced functions for component development.

Each `QueryConfig` function generally has a corresponding function in `DiscoveryServiceUtils` to get the results.

`QueryConfig` functions are most often used to obtain results that are specific to a component. Because of this, `QueryConfig` functions should never be persisted to the application data domain using `setQueryState()`, as this would affect all of the components that are bound to the same data. Instead, `QueryConfig` functions should only be added to a component's local copy of the `QueryState` object.

The available `QueryConfig` functions are:

- `AttributeTextValueSearchConfig`
- `AttributeValueSearchConfig`
- `BreadcrumbsConfig`
- `LQLQueryConfig`
- `RecordDetailsConfig`
- `ResultsConfig`

- `ResultsSummaryConfig`
- `SearchAdjustmentsConfig`
- `SortConfig`

In addition to the information here, for more details on the `QueryConfig` functions, see the *Component SDK API Reference*.

AttributeTextValueSearchConfig

Used for text searches, such as in the **Available Refinements** panel and the **Search Box** functions.

`AttributeTextValueSearchConfig` has the following properties:

Property	Description
`searchTerm`	String The term to search for in the attribute values.
`attribute`	String (optional) The attribute key for the attribute in which to search. Use the `attribute` property to search against a single attribute. To search against multiple attributes, use `searchWithin`.
`searchWithin`	List<String> (optional) A list of attributes in which to search for matching values.
`languageId`	String (optional) The country code for a supported language (such as "en" for English).

The following example searches for the term "merlot":

```
AttributeTextValueSearchConfig attributeTextValueSearchConfig
= new AttributeTextValueSearchConfig("merlot");
```

AttributeValueSearchConfig

Used for type-ahead in a search field. For example, used for **Available Refinements** to narrow down the list of available values for an attribute.

`AttributeValueSearchConfig` has the following properties:

Property	Description
`searchTerm`	String The term to search for in the attribute values.

Property	Description
`maxValuesToReturn`	int (optional) The maximum number of matching values to return. If you do not provide a value, then the default is 10.
`attribute`	String (optional) The attribute key for the attribute in which to search. Use the `attribute` property to search against a single attribute. To search against multiple attributes, use `searchWithin`.
`searchWithin`	List<String> (optional) A list of attributes in which to search for matching values.
`matchMode`	`ALL`|`PARTIAL`|`ANY`|`ALLANY`|`ALLPARTIAL`|`PARTIALMAX`|`BOOLEAN` (optional) The match mode to use for the search.
`relevanceRankingStrategy`	String (optional) The name of the relevance ranking strategy to use during the search.
`languageId`	String (optional) The country code for a supported language (such as "en" for English).

The following example searches for the term "red" in the WineType attribute values:

```
AttributeValueSearchConfig attributeValueSearchConfig
= new AttributeValueSearchConfig("red", "WineType");
```

BreadcrumbsConfig

Used to return the refinements associated with the query.

`BreadcrumbsConfig` has the following property:

Property	Description
`id`	String (optional) The ID of the breadcrumbs to be instantiated.

This example returns the refinements:

```
BreadcrumbsConfig breadcrumbsConfig = new BreadcrumbsConfig();
```

LQLQueryConfig

Executes an EQL query on top of the current filter state.

`LQLQuery` has the following property:

Property	Description
`lqlQuery`	AST The EQL query to add. To retrieve the AST from the query string, call `DataSource.parseLQLQuery`.

The following example retrieves the average of the P_Price attribute grouped by Region:

```
Query query
= dataSource.parseLQLQuery("return mystatement as select avg(P_Price) as avgPrice group by Region",
true);
LQLQueryConfig lqlQueryConfig = new LQLQueryConfig(query);
```

RecordDetailsConfig

Sends an attribute key-value pair to assemble the details for a selected record. The complete set of attribute-value pairs must uniquely identify the record.

`RecordDetailsConfig` has the following property:

Property	Description
`recordSpecs`	List<RecordSpec> Each new `RecordDetailsConfig` is appended to the previous `RecordDetailsConfig`.

The following example sends the value of the P_WineID attribute:

```
List<RecordSpec> recordSpecs = new ArrayList<RecordSpec>();
recordSpecs.add(new RecordSpec("P_WineID", "37509"));
RecordDetailsConfig recordDetailsConfig = new RecordDetailsConfig(recordSpecs);
```

ResultsConfig

Used to manage the returned records. Allows for paging of the records.

`ResultsConfig` has the following properties:

Property	Description
`recordsPerPage`	Long The number of records to return at a time.

Property	Description
`offset`	Long (optional) The position in the list at which to start. The very first record is at position 0. For example, if `recordsPerPage` is 10, then to get the second page of results, the offset would be 10.
`columns`	String[] (optional) The columns to include in the results. If not specified, then the results include all of the columns.
`numBulkRecords`	Integer (optional) The number of records to return. Overrides the value of `recordsPerPage`.

The following example returns a selected set of columns for the third page of records, where each page contains 50 records:

```
ResultsConfig resultsConfig = new ResultsConfig();
resultsConfig.setOffset(100);
resultsConfig.setRecordsPerPage(50);
String[] columns = {"WineID", "Name", "Description", "WineType", "Winery", "Vintage"};
resultsConfig.setColumns(columns);
```

ResultsSummaryConfig

Gets the number of records returned from a query.

```
ResultsSummaryConfig resultsSummaryConfig = new ResultsSummaryConfig();
```

SearchAdjustmentsConfig

Returns DYM (Did You Mean) and auto-correction items for a search.

```
SearchAdjustmentsConfig searchAdjustmentsConfig = new SearchAdjustmentsConfig();
```

SortConfig

Used to sort the results of a query. Used in conjunction with `ResultsConfig`.

`SortConfig` has the following properties:

Property	Description
`ownerId`	String (optional) The ID of the `ResultsConfig` that this `SortConfig` applies to. If not provided, uses the default `ResultsConfig` ID. If you configure a different ID, then you must provide a value for `ownerId`.
`property`	String The attribute to use for the sort.
`ascending`	Boolean Whether to sort in ascending order. If set to `false`, then the results are sorted in descending order.

For example, with the following `SortConfig`, the results are sorted by the P_Score attribute in descending order:

```
SortConfig sortConfig = new SortConfig("P_Score", false);
```

Creating and deploying a custom QueryFunction class

The Component SDK allows you to create custom `QueryFunction` classes.

Generating the Eclipse project for the QueryFunction class

The Component SDK includes a script to generate the Eclipse project for the `QueryFunction` class.

To generate the Eclipse project for a new `QueryFunction` class:

1. From the command line, change to the `components/endeca-extensions` subdirectory of the Component SDK.
2. To create a `QueryFilter` class, run the appropriate `.sh` or `.bat` version of the `create-queryfilter` command.

   For example on Linux:

   ```
   ./create-queryfilter.sh <queryFilterName>
   ```

Where <*queryFilterName*> is the name you want to use for the `QueryConfig` class. The name cannot have spaces.

The command creates a new directory called <*queryFilterName*>`-QueryFilter` in the `endeca-extensions` directory.

This directory is an Eclipse project that you can import directly into Eclipse.

It contains an empty sample implementation of a `QueryFilter`.

3. To create a `QueryConfig` class, run the appropriate `.sh` or `.bat` version of the `create-queryconfig` command.

   For example on Linux:

   ```
   ./create-queryconfig.sh <queryConfigName>
   ```

   Where <*queryConfigName*> is the name you want to use for the `QueryConfig` class. The name cannot have spaces.

   The command creates a new directory called <*queryConfigName*>`-QueryConfig` in the `endeca-extensions` directory.

   This directory is an Eclipse project that you can import directly into Eclipse.

   It contains an empty sample implementation of a `QueryConfig`.

For both `QueryFilter` and `QueryConfig` classes, the skeleton implementation:

- Extends either `QueryFilter` or `QueryConfig`.
- Creates stubs for the `applyToDiscoveryServiceQuery`, `toString`, and `beforeQueryStateAdd` methods.

  `applyToDiscoveryServiceQuery` and `toString` are required methods that you must implement.

  `beforeQueryStateAdd` is an optional method to verify the query state before the function is added. This method is used to prevent invalid query states such as duplicate refinements.
- Creates a no-argument, protected, empty constructor. The protected access modifier is optional, but recommended.
- Creates a private member variable for logging.

Implementing a custom QueryFunction class

After you create your new `QueryFunction` class, you then implement it.

To implement your new `QueryFunction`, you must:

- Add private filter or configuration properties.
- Create getters and setters for any filter properties you add.
- Define a no-argument constructor (protected access modifier optional, but recommended).
- Implement the `applyToDiscoveryServiceQuery` method.

  This method is called with the following arguments:

  - The Conversation Service query
  - A `stateName` string

Your custom function should use the Conversation Service API to apply itself to the conversation service query argument.

The `stateName` argument provides the value to use for state name references in Conversation Service filters or content element configs that your custom function adds to the query.

- Implement the `toString` method, which is used to compare `QueryFunction` instances for equality.

  `toString` should be consistent and deterministic in order to accurately determine if two instances of your custom `QueryFunction` are identical or distinct.

- Optionally, implement the `beforeQueryStateAdd(QueryState state)` method to check the current query state before the function is added.

Building and deploying a custom QueryFunction class

When you have finished development on your custom `QueryFunction` class, you build it, then add the resulting .jar file to the .ear file.

To build and deploy a `QueryFunction`:

1. In your Eclipse project for the `QueryFunction`, open the `build.xml` file.
2. If the project is not configured to build automatically, then in the outline view, right-click the deploy task and select **Run as...>Ant Build**.

   The Component SDK builds the `QueryFunction`, and places the resulting .jar file in the output directory you specified.
3. To make the `QueryFunction` available to all of your custom components, place the .jar file in the `app-inf/lib` directory of the extracted .ear file.
4. To add the `QueryFunction` to the Big Data Discovery instance:

   (a) Add the .jar file to the `app-inf/lib` directory of the .ear file.

   (b) Re-deploy the .ear file.

Adding a custom QueryFunction to a custom component project

If you just want to use a custom `QueryFunction` in a specific custom component, you add its `.jar` file to the component's Eclipse build path.

To add the `QueryFunction` to a custom component project:

1. In Eclipse, right-click the component project, then select **Build Path>Configure Build Path**.
2. Click the **Libraries** tab.
3. Click **Add Variable**.
4. Select **DF_GLOBAL_LIB**.

   You should have added this variable when you set up the Component SDK. See *Preparing your system for Component SDK development on page 10*.
5. Click **Extend**.
6. Open the `ext/` directory.

7. Select the `.jar` file for your custom `QueryFunction`.
8. Click **OK**.

After adding the `.jar` file to the build path, you can import the class, and use your custom `QueryFilter` or `QueryConfig` to modify your `QueryState`.

Part II

Using the Transform API

Chapter 5

Overview

This section describes transformations and the custom transform functions available in Big Data Discovery.

This section should be used together with the generated documentation for custom Groovy functions (Groovydoc), packaged together with Big Data Discovery, and known as the *Transform API Reference*.

About transformations and transformation scripts

Transformations are changes you can make to your project data set, after the source data has been processed and loaded into Studio. Transformations can be thought of as a substitute for an ETL process of cleaning your data. Transformations can overwrite an existing attribute, modify attributes, or create new attributes.

For example, you can do any of the following transformations:

- Change an attribute's data type
- Change capitalization of values
- Remove attributes or records
- Split columns into new ones (by creating new attributes)
- Add or remove attributes, or overwrite existing attributes
- Group or bin values
- Extract information from values.

Most transformations are available directly as specific options in the **Transform** page of Studio.

You can you use the Groovy scripting language and a list of custom, predefined Groovy-based ***transform functions*** available in Big Data Discovery, to create a ***transformation script***. Transformation scripts are collections of various transformations; they can contain any of the ***transform functions***.

You can also write your own transformations from scratch using Groovy, within the same **Transform** page of Studio, using the **Transformation Editor**.

When you commit a transformation script to a project, the script runs against the data sample but does not affect the data set in the **Catalog**. You can either apply the transform script to your current project, or create a new data set using the transformation script:

- When you commit the transformation script to the project, no new entry is created in the **Catalog**, but the current project does show the effects of the transform script.

- When you create a new data set using the transformation script, a new data set entry is added to the **Catalog** for use by other projects. That new data set is a new sample of the original source Hive table after the transformation script is applied. Creating a new data set in this way does not apply the transformation script to the current project.

About Groovy

Groovy is a dynamically-typed scripting language. Code written in the Java language is valid in Groovy, so users not familiar with Groovy may resort to the Java syntax. All custom transform functions available for you in Big Data Discovery are written in Groovy.

Groovy was chosen as the basis for the Transform API because it is flexible and easy to use. Additionally, although it is a dynamic language, it can use static compilation and static type checking to make it less error-prone at runtime.

Big Data Discovery lets you use many features of the Groovy language when writing your own custom transformations; it does, however, impose a few restrictions for security reasons. For more information, see *Unsupported Groovy language features on page 53*.

You can find more information on Groovy, including tutorials, in the *Groovy documentation*.

About transform functions

Transform functions are customized Groovy functions available in Big Data Discovery that you can include in your transformation scripts. Each transform function performs a specific operation on your data, from simple ones, such as converting an attribute to a different data type, to more complex ones, such as determining the overall sentiment of a document or a string of text.

Big Data Discovery provides these types of custom transform functions:

- **Conversion functions** convert values to different data types.
- **Date functions** perform actions on Date objects, such as adding a specific amount of time to a Date.
- **Enrichment functions** are based on Data Enrichment modules in Big Data Discovery. You can use them to extract complex information from your data.
- **Geocode functions** perform actions on Geocode objects, such as calculating the distance between two Geocode objects.
- **Math functions** perform mathematical operations on numerical values.
- **Set functions** perform different actions on sets of values on multi-value attributes in Big Data Discovery, such as obtaining the size of the value set, checking whether a set is empty, or converting a multi-value attribute to a single-value attribute. Set functions only work on multi-value (also known as multi-assign) attributes.
- **String functions** perform different actions on String values, such as concatenating two String values, or splitting a single String into multiple values.

Chapter 6

Working with Transformation Scripts

These topics describe the process of creating and applying transformation scripts, using the custom transform functions.

Transformation script workflow

At a high level, writing a transformation script and applying it to your data involves the following steps:

1. Write a custom transformation using custom transform functions within Big Data Discovery, or native Groovy language.
2. Use preview mode to debug your transformation and view its effects on your data.
3. Save the transformation to your transformation script.
4. Edit your transformation script by rearranging, modifying, and deleting individual transformations.
5. Apply your script to the sample data set your project was created from. This updates your copy of the project data set (it is a sample of the source Hive table), and makes it available in **Discover** area of Studio, where you can use guided navigation and search on it, as on any other data set in your project.
6. Apply your script to the source Hive table your project is based on. This creates a new Hive table and adds a new data set to the **Catalog** in Studio.

Writing transformations

You can write transformations in the **Transform** area of Studio, using the **Transformation Editor**. Transformations can contain attributes and records from your project data set as variables, and can create new attributes to hold the transformed values.

Formats for variables

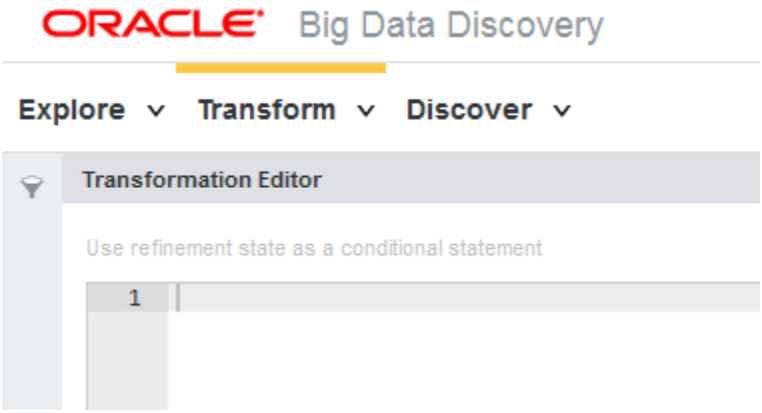
Setting transformation outputs

Functional and dot notation and function chaining

The Transformation Editor

You create transformations in the **Transformation Editor**, this is the built-in Groovy editor within the **Transform** area in Studio.

The **Transformation Editor** becomes available when you select **Custom Transform**, or **Add Attribute** from the attribute menu, or when you click the toolbox icon in the top right corner of **Transform**, in Studio:



In the editor:

- **Syntax highlighting** enables color-coding of different elements in your transformation to indicate their type.
- **Auto complete** lets you view a list of autocomplete suggestions for the word you're typing, by pressing **Ctrl+space**. Use the arrow keys to navigate this list and press **Enter** to select the highlighted item.
- **Error checking** includes a built-in static parser that performs error checking when you preview or save your transformation. For more information, see *Exception handling on page 45*.

You can enter code into the editor in two different ways, depending on your programming experience level:

- If you are comfortable with Groovy, you can type directly into the **Transformation Editor**. Your code can contain any of the supported Groovy language features, and custom transform functions available in Big Data Discovery.
- If you have limited experience with Groovy, you can create transformations using predefined lists of custom transform functions and available attributes:
  - To view the list of transform functions, click **Functions** above the **Transformation Editor**. In the **Functions** list, you can learn about each function by hovering the mouse over its name.

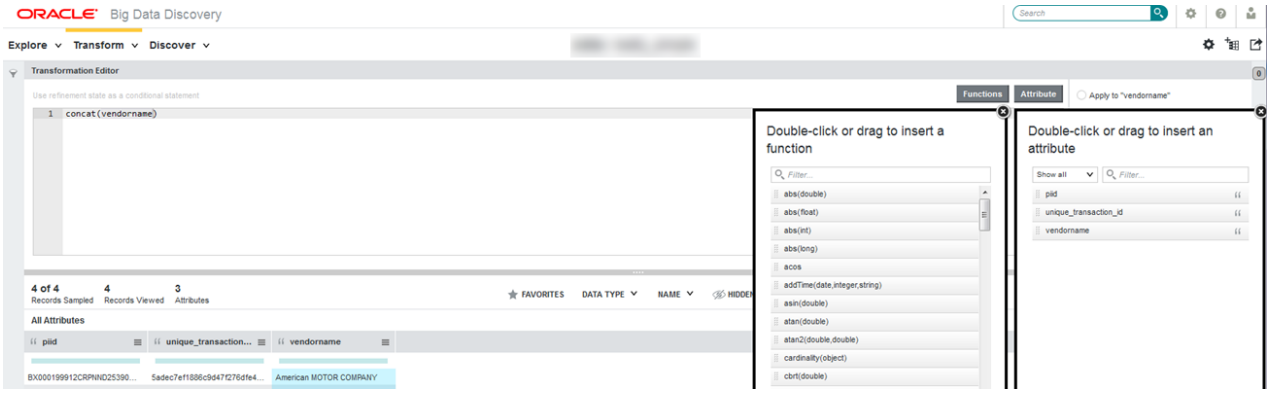
    Here is a list of custom functions you can add to your transformation script:

- To view the list of your data set's attributes, click **Attributes**. The **Attributes** list displays an icon next to each attribute's name indicating its data type.

  You can filter the **Attributes** list by data type:

To add items from either list to your transformation, click and drag its name into the **Transformation Editor**.

To add an item from the **Attributes** list as a parameter to a function, drag the attribute's name directly on top of the function's placeholder text:



Formats for variables

You can use attributes from your project data set as variables in your transformation scripts. This allows you to pass attributes to transform functions as parameters and perform other operations on them.

To include an attribute in a transformation, you can reference it using the formats described below. The specific formats you can use for a given attribute depend on whether its name meets the following requirements:

- Names should consist of a letter or underscore (_) followed by zero or more alphanumeric characters (a-zA-Z, 0-9) and underscores.
- Names can't contain any of the reserved keywords from either the **Transform** area in Studio, or from Groovy. For more information, see *Unsupported Groovy language features and Reserved Keywords on page 53*.

Note: Unlike other variables, attributes don't need to be declared.

This table describes the formats you can use to include attributes as variables in transformations.

Format syntax	Description
`<attribute>`	Attribute names that consist of a letter or underscore (_) followed by zero or more alphanumeric characters (a-zA-Z, 0-9) and underscores. Names can't contain any of the reserved keywords from either the **Transform** area in Studio, or from Groovy.
`row["_<attribute>"]`	The map format. This can be used for all attributes, including those whose names don't meet the naming requirements described above. You can use single or double quotes.

Format syntax	Description
`row."_<attribute>"`	The long dot format. This can be used for all attributes, including those whose names don't meet the naming requirements described above. You can use single or double quotes.
`row._<attribute>`	The short dot format. This can only be used for attributes whose names consist of alphanumeric characters, dollar signs ($), and underscores.

Note: The format you use for an attribute affects how it is handled by the static parser. For more information, see *Exception handling and troubleshooting your scripts on page 45*.

Setting transformation outputs

You can set your transformation to output to either the selected attribute or a new attribute (this is useful if a transformation is creating a new column).

Applying a transformation to the selected attribute overwrites the attribute with the transformed data. Setting the transformation to output to a new column adds a new attribute to your project data set.

To set the output for a transformation:

1. Select one of the radio buttons next to the **Transformation Editor**:
   - **Apply transformation to *[attribute name]***
   - **Create a New Attribute**
2. If you selected **Create a New Attribute**, enter a unique name for the new attribute in the **New Attribute Name** text box.

   The new name can only contain alphanumeric characters and underscores (_). If the name you enter contains unsupported characters, the outline of the text box turns red and you receive an error message if you try to preview or save the transformation.
3. Optionally, select the new attribute's data type from the **Data Type** dropdown menu.

   Transform automatically selects an appropriate data type, but you can override its choice.
4. If the new attribute should be multi-assign, deselect the **Single Assign** checkbox.

Functional and dot notation and function chaining

You must use proper syntax when adding transform functions to your script, or your script won't run properly. You can reference all transform functions using functional notation, as described in this topic.

```
<function>(<argument1>[,<argumentN>])
```

For example, the following code applies the `geotagAddress` function to an attribute called `address`:

```
geotagAddress(address)
```

You can use dot notation to include original Groovy functions that aren't specific to Big Data Discovery:

```
<attribute>.<function>()
```

For example, the following code uses the `toString` function to convert an attribute called `quantity` to a String:

```
quantity.toString()
```

Note: You can only use dot notation for original Groovy functions. For the BDD-specific transform functions, you must use functional notation.

Function chaining

Function chaining allows you to apply multiple functions to an attribute in a single statement. You chain functions by passing an attribute to one function, then passing that function to another function. The innermost function (the one receiving the attribute as a parameter) is evaluated first, and the outermost function is evaluated last.

For example, the following code takes an IP address, determines the city it originated from, then converts the name of the city to uppercase:

```
// Performs two transformations on a single attribute using one line of code:

toUpperCase(geotagIPAddressGetCity(IP_address))
```

The following code produces the same result as the code above, but is more verbose:

```
// The same two transformations as above, without chaining.
// 'city_name' is a temporary variable that stores the output of geotagIPAddressGetCity()

def city_name = geotagIPAddressGetCity(IP_address)
toUpperCase(city_name)
```

As you can see in the examples, function chaining makes your code cleaner and easier to read. Additionally, not having to include placeholder variables, such as `city_name` in the second example, helps make your code less error prone.

Exception handling and debugging

These topics describe exception handling in **Transform** and show how to debug individual transformations.

Script evaluation

Transformation scripts are evaluated top-down on each input row. This means that each transformation in the script is applied in order to the first input row, then again to the second row, and so on. This is illustrated by the following pseudo code:

```
for each input row R
        for each transform T
                R <- apply T to R
```

Additionally, each transformation can see the results of the transformations that ran before it. This is important to understand, as transformations within a script can be dependent on others. You should be aware of these dependencies when editing transformations or rearranging their order within your script.

Dynamic typing vs. static typing

This topic is provided for reverence only as it explains the differences between dynamic and static typing. Understanding the differences between dynamic and static typing is key to understanding the way in which transformation script errors are handled, and how it is different from the way Groovy handles errors. This will also help you interpret errors created by your transformation script.

Note: It is important to know that the Groovy implementation within Big Data Discovery enforces static typing. For information on exception handling in **Transform**, which uses a static parser overriding Groovy's dynamic typing behavior, see *Exception handling and troubleshooting your scripts on page 45*.

There are two main differences between dynamic typing and static typing that you should be aware of when writing transformation scripts.

First, dynamically-typed languages perform type checking at runtime, while statically typed languages perform type checking at compile time. This means that scripts written in dynamically-typed languages (like Groovy) can compile even if they contain errors that will prevent the script from running properly (if at all). If a script written in a statically-typed language (such as Java) contains errors, it will fail to compile until the errors have been fixed.

Second, statically-typed languages require you to declare the data types of your variables before you use them, while dynamically-typed languages do not. Consider the two following code examples:

```
// Java example
int num;
num = 5;
```

```
// Groovy example
num = 5
```

Both examples do the same thing: create a variable called `num` and assign it the value `5`. The difference lies in the first line of the Java example, `int num;`, which defines `num`'s data type as `int`. Java is statically-typed, so it expects its variables to be declared before they can be assigned values. Groovy is dynamically-typed and determines its variables' data types based on their values, so this line is not required.

Dynamically-typed languages are more flexible and can save you time and space when writing scripts. However, this can lead to issues at runtime. For example:

```
// Groovy example
number = 5
numbr = (number + 15) / 2  // note the typo
```

The code above should create the variable `number` with a value of `5`, then change its value to `10` by adding `15` to it and dividing it by `2`. However, `number` is misspelled at the beginning of the second line. Because Groovy does not require you to declare your variables, it creates a new variable called `numbr` and assigns it the value `number` should have. This code will compile just fine, but may produce an error later on when the script tries to do something with `number` assuming its value is `10`.

Exception handling and troubleshooting your scripts

Transform uses a static parser to override some of Groovy's dynamic typing behavior and detect parsing errors, such as undefined variables, when you preview or save your transformations.

Important: Because the static parser forces Groovy to behave like a statically-typed language, you cannot use Groovy's dynamic typing features in your transformations. For example, while undeclared variables are normally allowed in Groovy, they produce parsing errors in **Transform**.

The static parser also verifies that the attributes referenced directly in your script match those defined in your data set's schema. Any attributes that don't match (for example, ones that are misspelled) produce an error.

Important: The static parser does not verify that parameters included in your transformation match their syntax as referenced in the row map of some custom functions, such as enrichment functions. If you incorrectly reference a parameter from a function, your transformation script will not validate, but the parser will not specify an error. Therefore, check the Transform API Reference (either in this document or in the Groovydoc), to verify that you correctly reference function parameters in the row map.

If you include attributes as variables in your transformation scripts, the format you use for an attribute affects how it is handled by the static parser. For information about attribute formats, see *Formats for variables on page 41*.

If your transformation contains any parsing errors, **Transform** displays the resulting messages in the **Transformation Error** dialog box when you preview or save the transformation. Additionally, the **Transformation Editor** displays a red **X** icon next to each line that contains an error. You can hover over these icons to view more information about the error.

You should close the dialog box, fix the errors, then preview your transformation again to verify that all errors have been fixed. You cannot save your transformation to your script until it is free of errors.

Troubleshooting exceptions for set functions

You can run the following set functions from the Transform API only on multi-assign attributes (these attributes are known as multi-value attributes in Studio):

- cardinality()
- isSet()
- isEmpty()
- isMemberOf()
- toSet()
- toSingle()

These functions belong to the in-line transformations you can do in **Transform**. These set functions are applicable to sets of values on attributes that are multi-assign.

If you run any of these functions from **Transform** in Studio, and the attribute on which you attempt to run them is a single-assign (single-value) attribute, the Transform API may throw NULL or an exception, depending on the Dgraph type of the attribute.

Note: You can check if an attribute is multi-value by looking at a data set in **Explore**, and selecting a table view. A column that will have more than a single value in a cell indicates that this column represents a multi-value attribute. You can also check the value of the Multi-Value column for your data set in **Project settings>Data Views**.

To summarize, if you receive an exception when attempting to run a transformation, check if the attribute on which you run the transformation is a single-value. In this case, set functions do not apply.

Security exceptions

If your transformation script contains any of the Groovy language features that are not supported, the parser throws a security exception, which is displayed in the **Transformation Error** dialog box. Remove the code that caused the error.

For more information on the Groovy language features that can cause security exceptions, see *Unsupported Groovy language features and Reserved Keywords on page 53*.

Troubleshooting runtime exceptions

The static parser can't detect all errors, particularly runtime exceptions caused by anomalies in your data. **Transform** typically handles these errors by returning null values for data it can't process.

If you want to know more about why your transformation script is producing null values, you can wrap your code in a `try` block and set its output to a new temporary attribute of type String (it will show up in your project's data set table as a new column for an attribute of type String):

```
try {
    <transformation script>  // replace this with your transformation script code
    'OK'
  } catch (Exception ex) {
    ex.getMessage()
  }
```

When you preview the transformation script, any error messages it produces will be output to the temporary column of type String. Once you have debugged it, you can delete the `try` block and remove the temporary attribute.

Transform logging

If your transformation script fails to commit, you can learn more about the cause of the failure by looking through the Data Processing logs.

Data Processing writes its logs to a user-specified directly on each Data Processing node in the cluster. The precise location is defined in the `logging.properties` file, which is located in the `$OBDD_HOME/DataProcessing/config/` directory.

Each transformation is identified within the logs by the name of the data set it was applied to and the name of the project it originated from. You can use this information locate the messages related to your script and determine which function(s) caused the failure.

Preview mode

You can preview a transformation at any time by clicking **Preview**, to see the effect it will have on your project data set. Preview mode is also a useful debugging tool, as it detects any runtime errors or corner case exceptions your transformation contains.

When you click **Preview**, the **Transformation Editor** updates the transformation script.

When you preview a transformation, **Transform** finds and displays runtime errors that weren't detected by the static parser. It is therefore recommended that you preview your transformations and fix any errors they contain before saving them to your transformation script. You can revert the changes made in the preview by clicking **Cancel**.

Preview only updates your project data set in Studio's internal files backing the **Transform**; it does not affect any data sets in the Dgraph index, and the results are not visible to other users of your project in Studio. Additionally, your project data set is still associated with its source Hive data, so any changes made to the source are still reflected within your project.

Editing, deleting and rearranging your transformations

You can edit individual transformations after you have added them to your transformation script. You can also edit the transformation script itself by rearranging and deleting transformations.

Editing individual transformations

To edit a custom transformation, click the pencil icon next to its name in the transformation script to reopen it in the **Transformation Editor**. Make the required changes, then click **Save**.

Note: You can't make changes to transformations added from the **Quick Transformations** menu, such as **Convert to Boolean**.

Deleting transformations from your script

To remove a transformation from the transformation script, click **X** next to its name.

Transform alerts you if you delete a transformation that other transformations are dependent on.

Rearranging transformations within your script

You can rearrange the transformations in the transformation script by clicking and dragging their names up or down.

Transform alerts you if you move a transformation that other transformations are dependent on.

Applying transformation scripts to project data sets

You can apply your transformation script at any point to make changes in your project data set. When the script finishes running, users working with your project can view, search, use guided navigation and interact with the transformed data in **Transform**, **Explore**, and **Discover** areas of Studio.

The transformed data set is only available within your project. The **Commit** operation does not add a new data set to the **Catalog**, nor does it modify the source data in Hive.

Note: Due to the way BDD converts Hive source table data types to its own data types, applying your script to the project's data set may result in some omitted data types. For example, some complex Hive data types that do not match the Dgraph data types are omitted. For more information, see *Data type conversion on page 52*.

To commit your script:

1. In the **Transformation Editor**, click **Commit** at the bottom of the transformation script.

   Transform becomes locked and a message appears stating that the operation may take several minutes to complete. Don't leave or refresh the page until the script finishes running.

When the script finishes running, **Transform** displays a message indicating whether it succeeded or failed. If it succeeds, you can refresh **Transform** to view the transformed data set.

When you commit your transformation script, the data processing component in Big Data Discovery does the following:

1. Obtains the schema for the transformed data set from the Dgraph.
2. Transforms the data using the transformation script.
3. Creates a new project data set based on the schema and metadata and populates it with the transformed data.

You can continue to work on your script after you apply it to the data set (recall that a project data set in Big Data Discovery is a sample of your source Hive table). You can also reapply the transformation script to the data set as many times as you like.

Transform locking

The **Transform** area in Studio provides a locking mechanism to ensure that multiple users working with the same data set within a project can't transform the data set at the same time.

Transform locks a data set when a user previews or saves a transformation (that is, when the user clicks **Preview** or **Apply to Script** from the **Transformation Editor**). The lock remains set for a period of time, which is extended each time the user previews or saves a transformation, or while they are actively working in the **Transformation Editor**.

Transform thus locks the data set when a user clicks **Commit to Project**, which runs the transformation script against the project's data set. This lock applies to all users, including the one who ran the script.

A lock on a data set remains set until any of these conditions are met:

- The script finishes running.
- Studio times out from inactivity after a period of 30 minutes.
- The session is ended by the user signing out of Studio.

When a data set is locked, users can only perform the following actions:

- View the data set
- Copy transformation scripts
- Toggle rows and values
- Sort columns.

Creating a new Hive table with the transformation script

When you use **Create a Data Set** in the Transformation Editor, your transformation script is applied to the source Hive table your project data set was created from. This operation creates a new Hive table in the Dgraph index and adds a new data set to the **Catalog**.

Note: Due to the way BDD converts Hive source table data types to its own data types, applying your script to the source table may result in some omitted or changed data types. For example, some complex Hive data types that do not match the Dgraph data types are omitted. For more information, see *Data type conversion on page 52*.

To create a new data set:

1. Click the menu icon in the transformation script panel and select **Create a Data Set**.

   The **Create a Data Set** dialog box opens.

2. In the **New Hive Table Name** field, enter a unique name for the new Hive table.

   The name you choose can only contain alphanumeric characters and underscores.

3. In the **New Hive Table Data Directory**, enter the location in HDFS where you want your table to be stored.

4. In the **New Data Set Name** field, enter a unique name for the new data set.

   This is the name the new data set will have in **Catalog**. The name you choose can be different from the Hive table's name.

5. Optionally, enter information about your transformation script or new data set in the **Comments** field.

   This will be stored as the new table's metadata, along with the transformation script and the date the table was created.

6. Click **Save**.

   A dialog box appears indicating that the transformation is in progress and may take several hours to complete.

If the script is successful, the new Hive table will be added to the index and the new data set will appear in **Catalog**.

If you do not see the new data set in **Catalog**, then the script failed. You can learn more about why it failed by checking the Data Processing logs. For more information, see *Transform logging on page 46*.

When you apply your transformation script to the source Hive table, data processing in Big Data Discovery does the following:

1. Obtains the transformation script from Studio.
2. Retrieves the schema of the transformed project data set from the Dgraph.
3. Creates a new Hive table (let's name it HT2 in this example), using the project data set's schema.
4. Loads the data row by row from the original source Hive table (let's name it HT1) to the HT2 Hive table, and at the same time runs the transformation script on each loaded row, and saves the transformed data as HT2.
5. Samples the HT2 Hive table (this is the new Hive table with the transformed data) and adds the resulting data set to the **Catalog**.

Chapter 7

Transform Function Reference and Examples

This section lists data types, discusses data type conversions that take place when transformation scripts are applied, provides a list of reserved words and unsupported features of Groovy, and contains examples of custom transform function usage. It also includes the reference documentation for the custom transform functions in Big Data Discovery.

Use this section together with the *Transform API Reference* (this is the Groovydoc documentation, from the custom functions available in Groovy within Big Data Discovery).

Data types

In transformation scripts, the attribute's data type is represented as a Groovy data type. This topic discusses how the Dgraph data types match the Groovy data types.

When you create a project based on a data set found in **Catalog**, this data set is indexed in Big Data Discovery, and each attribute in the project's data set is assigned a Dgraph data type (also known as the `mdex:<type>`). All Dgraph data types begin with `mdex:`, and those used for multi-assign attributes end with `-set`. For more information on Dgraph data types, see the *Data Processing Guide*.

In transformation scripts, Groovy data types are used, as described in the following table.

When you commit your script, it outputs data with Groovy data types. These data types are then converted to the appropriate Dgraph data types when the data is written to a data set.

In addition to the settings shown in this table, the following two considerations apply:

- Multi-assign Dgraph attributes correspond to Set Groovy types. For example, a Dgraph type `mdex:int-set` (which is a type used in the Dgraph for multi-assign attributes of type Integer), corresponds to a Java type `set <integer>`.
- `Long` data types are converted to Integer data types if they are small enough.

Groovy data type	Corresponding mdex data types
`Boolean`	`mdex:boolean`, `mdex:boolean-set`

Groovy data type	Corresponding mdex data types
`Integer`	`mdex:int`, `mdex:int-set`
`Long`	`mdex:long`, `mdex:long-set`
`Double`	`mdex:double`, `mdex:double-set`
`Date`	`mdex:dateTime`, `mdex:dateTime-set`, `mdex:time`, `mdex:time-set`
`Geocode`	`mdex:geocode`, `mdex:geocode-set`
`String`	`mdex:string`, `mdex:string-set`

Data type conversions

When you apply your transformation script to the project data set or to the source Hive table (when you create a new data set from within **Transform**), the data processing in Big Data Discovery converts most of the Hive data types to its corresponding Dgraph data types. However, this can result in some of the original data types being changed or omitted. This topic discusses these data type conversions in detail.

For information on complex types in Hive tables, see *https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Types#LanguageManualTypes-ComplexTypes*. The types that are present in your source Hive tables depend on the Hadoop environment you use.

For information on which data types are supported by Big Data Discovery, see the *Data Processing Guide*.

The following table describes how different Hive data types are affected by transformation scripts. The table lists the data types the source Hive table can contain and shows the data types in the Dgraph (`mdex:<type>`) to which they are converted.

Source Hive table data type (before the transformation script is applied)	Dgraph data type	Target Hive table data type (after the transformation script is applied)
`BOOLEAN`	`mdex:boolean`	`BOOLEAN`
`TINYINT`	`mdex:long`	`BIGINT`; this type is converted to Long during ingest.
`SMALLINT`	`mdex:long`	`BIGINT` ; this type is converted to Long during ingest.
`INT`	`mdex:long`	`BIGINT`; this type is converted to Long during ingest.
`BIGINT`	`mdex:long`	`BIGINT`

Source Hive table data type (before the transformation script is applied)	Dgraph data type	Target Hive table data type (after the transformation script is applied)
FLOAT	mdex:double	DOUBLE
DOUBLE	mdex:double	DOUBLE
DECIMAL	mdex:double	DOUBLE ; this may result in loss of precision.
DATE	mdex:dateTime	TIMESTAMP
TIMESTAMP	mdex:dateTime	TIMESTAMP
STRING	Discovered mdex:<*type*>	STRING (or other primitive types)
CHAR	Discovered mdex:<*type*>	STRING (or other primitive types)
VARCHAR	Discoveredmdex:<*type*>	STRING (or other primitive types)
ARRAY (complex)	Multi-assign of the ARRAY type. For example, for an ARRAY of decimals, it becomes a multi-assign attribute of mdex:double.	ARRAY (complex) of the types obtained from the Dgraph type.
STRUCT (complex)	None	Multiple fields of this format: struct_(structName)_(fieldName)
BINARY	None	Unsupported; the entire field or column is omitted.
MAP (complex)	None	Unsupported; the entire field or column is omitted.
UNION (complex)	None	Unsupported; the entire field or column is omitted.

Unsupported Groovy language features and Reserved Keywords

This topic lists reserved keywords and those Groovy language features that are not supported in Big Data Discovery.

Reserved Keywords

Reserved keywords are words that have special meanings in Groovy language and therefore cannot be used as variable or function names in Groovy scripts. The following table lists Groovy's reserved keywords:

abstract	as	assert

boolean	break	byte
case	catch	char
class	const	continue
def	default	do
double	else	enum
extends	false	final
finally	float	for
goto	if	implements
import	in	instanceof
int	interface	long
native	new	null
package	private	protected
public	return	short
static	strictfp	super
switch	synchronized	this
threadsafe	throw	throws
transient	true	try
void	volatile	while

Additionally, the following keywords are reserved by the transform functions used in Big Data Discovery:

DEFAULTLANG	MILLISECONDS	SECONDS
MINUTES	HOURS	DAYS
WEEKS	MONTHS	YEARS
DATEFORMAT_DEFAULT		

Attributes with reserved keywords for names can only be referenced via the row map format; referencing them directly will produce an error. For more information on the row map format, see *Formats for variables on page 41*.

Unsupported functions

For security reasons, **Transform** does not support all of Groovy's original classes. Transformation scripts that contain methods of unsupported classes produce errors and cannot be saved to the script in Studio.

You can use any of the functions listed in the **Transformation Editor**'s **Functions** list, as well as functions from the following classes (other original Groovy functions are not supported).

Note: If a function you need to use is unsupported, contact Oracle Customer Support.

This table lists the supported Groovy classes:

`Math`	`Integer`	`Float`
`Double`	`Long`	`BigDecimal`
`Date`	`Geocode`	`Object`
`Closure`	`String`	`Set`
`Array`	`InvokerHelper`	`Exception`
`Rowbinding`		

Examples

This section contains examples of different types of transformations you can create using transform functions and Groovy.

Extracting a date

This example pulls out the year-month. Note that the date formats adhere to the **SimpleDateFormat** class in Java. For information on **SimpleDateFormat** class, see: *http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html*.

```
toString(pickup_datetime, 'yyyy-MM')
```

Time conversion

This example uses the `floor` function to convert `trip_time_in_secs` to minutes:

```
floor(trip_time_in_secs/60)
```

`trip_time_in_seconds` is first divided by 60 to determine the number of minutes in the trip. The `floor` function then rounds this number down and returns it as a double.

Date calculation

The following code uses the `diffDatesWithPrecision` function to calculate the number of days to `pickup_datetime`:

```
diffDatesWithPrecision(today(),pickup_datetime,DAYS)
```

`today()` obtains the current date, `pickup_datetime` is the pickup date, and `DAYS` specifies the time unit to return the result in.

Locating string patterns using `find` and regular expressions

This example shows how to use the `find` function. Assume you have an attribute `amz_desc`, and it has the following value:

```
The future is here people and it has arrived in the form of an LED digital bracelet watch. That's
right. You'll never
have to live the disappointing life of not owning a digital bracelet watch. This revolutionary piece
of technology is not
only stylish but it will completely change the way you read time.
```

You can use the following transformation code with `find` function and regular expressions in it. This script locates a string pattern that begins with "LED" and ends with "h.", where in between, there can be zero or more of any characters (excluding a new line):

```
find(amz_desc,'LED.*h\\.')
```

This script produces the following result:

```
LED digital bracelet watch. That's right. You'll never have to live the disappointing life
 of not owning a digital bracelet watch.
```

In the output, you can see that the script a part of the first sentence, and includes it because it starts with "LED". Next, the script looks for the last occurrence of "h.", which is a letter h followed by a period at the end of the sentence.

Note also that the script must escape the second "." , because the script wants that the second "." is treated as a regular period in the end of the sentence, and not as a regular expression for any character excluding a new line. Typically, to escape a character, "\" is used, whoever, in this case, "\" must be used twice "\\". This is because the transformation script must pass the "\" literally (as text) to the Groovy language, which then treats it as an escape character for the "." period.

String replacement using `replace`

This example replaces strings:

```
replace(cost,'\\$','')
```

Substring replacement using `trim` and `replace`

This example removes `County` suffix from `pickup_county` attribute:

```
trim(replace(pickup_county,'County',''))
```

The above code uses method chaining to perform multiple actions with a single statement. `replace` first locates the substring `County` in the attribute `pickup_county` and replaces it with a blank String (`''`), which essentially removes it. `trim` then removes all leading and trailing whitespace from the result.

Substring replacement using regular expressions

The following code masks the number in the `medallion` attribute by replacing it with `'X'`:

```
replace(medallion,'[0-9]','X')
```

The `replace` function locates all numeric characters in the `medallion` attribute using the regular expression `[0-9]`, which defines a range of characters. It then replaces any characters that match this pattern with the String `X`.

Terms extraction using TF/IDF algorithm (extraction of key phrases)

The following code demonstrates how to extract terms from a message title and body using the `extractKeyPhrases` custom Transform function (it is one of enrichment functions). This function extracts key phrases using TF/IDF algorithm, which takes the total number of times each term appears within the String and offsets that value by the number of times it appears within a larger body of work. Offsetting the value helps filter out frequently-used terms like "the" and "it". The body of work used as the control is selected internally based on the String's language; for example, the model used for English is based on a New York Times corpus.

```
extractKeyPhrases(concat(message_title,' ',message_body))
```

The `concat` method combines the values of `message_title` and `message_body` into a single String, separated by a space. `exTractKeyPhrases` then extracts and returns key terms from the new String.

Terms extraction against a specified whitelist

In this example, the first line defines a whitelist named `tagList`, and the second line uses the `extractWhiteListTags` custom Transform function (it is one of the enrichment functions), which first matches the input text against the specified whitelist, next, finds and extracts all occurrences of any terms listed in the whitelist (in English), as `WhitelistTags`, and then returns a list of matching expansions:

```
def tagList = "WallMart\tWal-Mart\r\nWalMart\tWal-Mart\r\nWalMart\tWal-Mart\r\nCVS\r
\nTarget\r\nSams\tSam'sClub\r\nSams Club\tSam's Club\r\nCostco\r
\nMacys\tMacy's\r\nMacy's\r\nUlta\r\nTesco\r\nMetro\r\nSafeway"
extractWhitelistTags(full_text,tagList,"en",true,false)
```

Note that the language specified is English (`en`), the matches are case-sensitive (`true`), and unbounded, thus match the whole words only (`false`).

reverseGeotagCity example

The following code uses two attributes to create a Geocode object, then returns the Geocode's `city` field:

```
reverseGeotagGetCity(toGeocode(pickup_lat,pickup_long))
```

geotagAddress* examples

The `geotagAddress*` functions converts a valid address String to a Geocode object. Because your data set may contain ambiguous or incomplete addresses, `geotagAddress*` functions have multiple variants, such as for country or region, that let you obtain more detailed output.

Let's consider the address "Vernon, CA". This address is ambiguous, because "CA" is the abbreviation for both Canada and the state of California. Additionally, Canada and California both have a city named "Vernon".

The address does not contain a postal code, making it impossible to determine which is the correct Vernon. There are several ways of handling this.

This example returns the region "British Columbia":

```
geotagAddressGetRegion("Vernon,CA")
```

This example returns the country "CA" for Vernon, British Columbia, Canada:

```
geotagAddressGetCountry("Vernon,CA")
```

This example, however, returns information for Vernon, California, USA:

```
geotagAddressGetCountry("Vernon,CA",["PREFERRED_LEVEL":"REGION"])
```

This example returns information for Canada, because Vernon, CA has a higher population and thus is the closest match:

```
geotagAddressGetCountry("Vernon,CA",["STRICT_MODE":false])
```

Finally, this example returns null (no value), because the address is invalid:

```
geotagAddressGetCountry("Vernon,CA",["STRICT_MODE":true])
```

Simple conditional statement

The following code uses an if...else statement to assign a flag to a record based on the value of the `tip_amount` attribute.

```
// if the value of tip_amount is greater than 0, assign the record the 'Tip' flag:
if(tip_amount>0){
    'Tip'
    }
// if the above statement is false, assign the record the 'No Tip' flag:
else{
    'No Tip'
    }
```

Advanced conditional statement

The following code assigns each record a flag based on the `tip_amount` attribute. However, this one uses a series of if...else statements to assign different flags to each percentage range:

```
// if the tip was more than 25% of the total fare, assign the record the Large Tip flag:
if(tip_amount/fare_amount>.25){
    'Large Tip'
}

/
/ if the tip was less than or euqal to 25% and higher than 18%, assign the record the Standard Tip
flag:
else if(tip_amount/fare_amount<=.25 || tip_amount/fare_amount>.18){
    'Standard Tip'
}

// if the tip was less than or equal to 18%, assign the record the Small Tip flag:
else if(tip_amount/fare_amount<=.18 || tip_amount/fare_amount>0){
    'Small Tip'
}

// if all of the above statements failed assign the record the No Tip flag:
else{
```

```
    'No Tip'
}
```

Advanced conditional statement using a variable

This example performs the same operation as the previous example, but uses a variable to store the tip percentage rather than calculating it in each statement.

```
// calculate the tip percentage and assign it to PercentageTip
def PercentageTip = tip_amount/fare_amount

// if the value of PercentTip is greater than .25 (25%),
// assign the record the Large Tip flag
if(PercentTip>.25){
    'Large Tip'
}

// if the value of PercentTip is less than or equal to .25 and higher than .18
// assign the record the Standard Tip flag
else if(PercentTip<=.25 && PercentTip>.18){
     'Standard Tip'
}

// if the value of PercentTip is less than or equal to .18 and higher than 0
// assign the record the Small Tip flag
else if(PercentTip<=.18 && PercentTip>0){
     'Small Tip'
}

// if all of the above statements failed assign the record the No Tip flag
else{
    'No Tip'
}
```

Advanced conditional statement using date logic

The following code uses a series of else...if statements to create a multi-assign value. It uses the `diffDatesWithPrecision` function to calculate the amount of time between the current date and the attribute `dropoff_datetime` in days. It then assigns the record a list of String values that specify the different ranges of time it falls into.

```
if(diffDatesWithPrecision(dropoff_datetime,today(),'days')<=30){
    toSet('Last 30 Days','Last 90 Days','Last 180 Days')
}
else if(diffDatesWithPrecision(dropoff_datetime,today(),'days')<=90){
    toSet('Last 90 Days','Last 180 Days')
}
else if(diffDatesWithPrecision(dropoff_datetime,today(),'days')<=180){
   toSet('Last 180 Days')
}
else{
    toSet('Greater than 180 Days')
}
```

Examples with multi-assign values

The following examples demonstrate how to work with multi-assign values.

This example takes a multi-assign attribute named `ItemColor` and uses `toUpperCase` function to convert its values to upper case:

```
ItemColor.collect
{toUpperCase(it)}
```

The following code iterates through the values in a multi-assign attribute called `MedalsAwarded` and adds a specific number of points for each type of medal it finds to the variable `MedalValue`. It then returns `MedalValue`, which contains the total number of points awarded for each medal in the attribute.

```
def MedalValue=0

for (int i in 0..cardinality(MedalsAwarded)-1) {
     if(indexOf(MedalsAwarded[i],'Gold')>=0){
          MedalValue=MedalValue+3;
     }
     else if(indexOf(MedalsAwarded[i],'Silver')>=0){
          MedalValue=MedalValue+2;
     }
     else if(indexOf(MedalsAwarded[i],'Bronze')>=0){
          MedalValue=MedalValue+1;
     }
}
MedalValue
```

For example, if `MedalsAwarded['Gold','Silver','Gold']`, the final value of `MedalValue` would be `8`.

Here is another option for iterating through the values in a multi-assign attribute:

```
def MedalValue=0

for (x in MedalsAwarded) {
     if(indexOf(x,'Gold')>=0){
          MedalValue=MedalValue+3;
     }
     else if(indexOf(x,'Silver')>=0){
          MedalValue=MedalValue+2;
     }
     else if(indexOf(x,'Bronze')>=0){
          MedalValue=MedalValue+1;
     }
}
MedalValue
```

Multi-assign value operations using method chaining

The following code iterates through a multi-assign attribute called `PartIdentifier` and replaces its identification numbers with `X` to mask them.

```
PartIdentifier.{collect(trim(replace(substring(it,1,10),'[0-9]','X')))}
```

The above example uses method chaining to perform a number of operations on the values of `PartIdentifier` in a single statement. The first part of the statement, `PartIdentifier.collect` calls the `collect` method on the `PartIdentifier` attribute. `collect` runs the code in the outermost set of parentheses on each of the values in the `PartIdentifier` multi-assign attribute.

`collect` first calls the `substring` method, which returns the substring of the String `it`, defined by the character in position 1 through position 10, where `it` is the implicit Groovy variable ranging over the numbers of `PartIdentifier`.

This substring is passed to the `replace` method, which replaces all numeric characters (`'[0-9]'`) with `X`. The `trim` method then takes the masked String and removes all leading and trailing whitespace from it.

List of transform functions

This section lists and describes custom transform functions in Big Data Discovery.

Conversion functions

Conversion functions change a value from one data type to another.

This table describes the conversion functions that **Transform** supports. The same functions are described in the *Transform API Reference* (Groovydoc).

These functions rely on the Java `dateFormat`:
http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html.

User Function	Return Data Type	Description
`toBoolean(Number n)`	`Boolean`	Converts a number to a Boolean value; for example, `toBoolean(1)` evaluates to `true`. Note that only `0` evaluates to `false`. Any number other than `0` (including negative numbers) evaluates to `true`.
`toBoolean(String s)`	`Boolean`	Converts a String to a Boolean value; for example, `toBoolean("yes")` returns `false`. Note that this method only evaluates to `true` if the String is `"true"`. Capitalization is ignored, so `"true"`, `"TRUE"`, and `"tRuE"` are all `true`.

`toDouble(Boolean b)`
`toDouble(Integer i)`
`toDouble(Long l)`
`toDouble(String s)`
	`Double`	Converts a Boolean, Integer, Long, or String to a Double.

User Function	Return Data Type	Description

`toInteger(Double d)`
`toInteger(Long l)`
`toInteger(Boolean b)`
`toInteger(String s)`
	`Integer`	Converts a Double, Long, Float, Boolean, or String to an Integer. If the original value is too large, causes an exception.
		  **Note:** These functions are not supported in BDD 1.0.

`toLong(Boolean b)`
`toLong(Date d)`
`toLong(Double d)`
	`Long`	Converts a Boolean, Date, or Double to a Long.

`toLong(String s)`	`Long`	Converts a valid String to a Long. Invalid Strings (for example, one that contains letters) return `null`.  **Note:** This function does not support locale parsing. If you need to specify a locale to properly convert the String, use a different Groovy method. For example, to properly convert the German String `"1.025,7"` to a long, you could use `NumberFormat.getInstance(Locale.GERMANY).parse("1.025,7")`.
`toString(Object arg)`	`String`	Converts an Object of any data type to a String.

Date functions

Date functions perform actions on Date objects, such as obtaining the month information from a specific date or adding time to a date.

This table describes the Date functions that **Transform** supports. The same functions are described in the *Transform API Reference* (Groovydoc).

User Function	Return Data Type	Description
`addTime(Date date, Integer timeToAdd, String timeUnit)`	`Date`	Adds time to a Date object. The time unit used must be one of the following: • `MILLISECONDS` • `SECONDS` • `MINUTES` • `HOURS` • `DAYS` • `WEEKS` • `MONTHS` • `YEARS`
`diffDates(Date firstDate, Date secondDate, String timeUnit)`	`Long`	Calculates the difference between two dates as a long in a specific time unit. The time unit must be one of the following: • `MILLISECONDS` • `SECONDS` • `MINUTES` • `HOURS` • `DAYS` • `WEEKS` • `MONTHS` • `YEARS`

User Function	Return Data Type	Description
`diffDatesWithPrecision(Date firstDate, Date secondDate, String timeUnit)`	`double`	Calculates the difference between two Dates as a double in a specific time unit. The time unit used must be one of the following: • `MILLISECONDS` • `SECONDS` • `MINUTES` • `HOURS` • `DAYS` • `WEEKS` • `MONTHS` • `YEARS`
`getDay(Date date, String timeZone, String locale)`	`Integer`	Returns a Date's day value. You can specify a time zone and locale as optional parameters; the defaults are `null` and `"en"`, respectively.
`getHour(Date date, String timeZone, String locale)`	`Integer`	Returns a Date's hour value. You can specify a time zone and locale as optional parameters; the defaults are `null` and `"en"`, respectively.
`getMilliSecond(Date date, String timeZone, String locale)`	`Long`	Returns a Date's millisecond value based on a time zone and locale parameters that you specify. The defaults are `null` and `"en"`, respectively.
`getMinute(Date date, String timeZone, String lcoale)`	`Integer`	Returns a Date's minute value. You can specify a time zone and locale as optional parameters; the defaults are `null` and `"en"`, respectively.
`getMonth(Date date, String timeZone, String locale)`	`Integer`	Returns a Date's month value. You can specify a time zone and locale as optional parameters; the defaults are `null` and `"en"`, respectively.
`getSeconds(Date, String timeZone, String locale)`	`Long`	Returns a Date's seconds value. You can specify a time zone and locale as optional parameters; the defaults are `null` and `"en"`, respectively.
`getYear(Date date, String timeZone, String locale)`	`Integer`	Returns a Date's year value. You can specify a time zone and locale as optional parameters; the defaults are `null` and `"en"`, respectively.
`isDate(String originalString, String dateFormat)`	`Boolean`	Determines whether a String is a valid Date value with a specific format.

User Function	Return Data Type	Description
`toDate(long l)`	`Date`	Converts a Long to a Date object.
`toDate(String date, String defaultFormat)`	`Date`	Converts a String to a Date object using a specific date format. Invalid date Strings return `null`.
`today(Date, String timeZone, String locale)`	`Date`	Returns the current date. You can specify a time zone and locale as optional parameters; the defaults are `null` and `"en"`, respectively.
`toString(Date date, String dateFormat, String locale)`	`String`	Converts a Date to a String. You must specify the Date's format and a locale, which default to `DATEFORMAT_DEFAULT` and `"en"`, respectively.
`truncateDate(Date date, String timeUnit)`	`Date`	Truncates a Date based on a given time unit. The time unit used must be one of the following: • `MILLISECONDS` • `SECONDS` • `MINUTES` • `HOURS` • `DAYS` • `WEEKS` • `MONTHS` • `YEARS` For example, `truncateDate((toDate("2015/03/31 21:34:56")),MONTHS)` returns `2015-03-01 00:00:00 UTC`.

Date constants

Date constants define the default Date format and the time units that can be passed to Date functions.

This table describes the Date constants that **Transform** supports.

Constant Name	Data Type	Description
`DATEFORMAT_DEFAULT`	`Object`	Defines the default Date format: `"yyyy/MM/dd HH:mm:ss"`
`DAYS`	`Object`	Defines the constant for days: `"days"`
`HOURS`	`Object`	Defines the constant for hours: `"hours"`
`MILLISECONDS`	`Object`	Defines the constant for milliseconds: `"milliseconds"`

Constant Name	Data Type	Description
`MINUTES`	`Object`	Defines the constant for minutes: `"minutes"`
`MONTHS`	`Object`	Defines the constant for months: `"months"`
`SECONDS`	`Object`	Defines the constant for seconds: `"seconds"`
`WEEKS`	`Object`	Defines the constant for weeks: `"weeks"`
`YEARS`	`Object`	Defines the constant for years: `"years"`

Enrichment functions

Enrichment functions are based on Data Enrichment modules used as part of data processing in Big Data Discovery. You can use these functions to extract meaningful information from your data and modify attributes to make them more useful for analysis.

The same functions are described in the *Transform API Reference* (Groovydoc).

More information on the Data Enrichment modules is available in the *Data Processing Guide*.

Transform supports the following enrichment functions:

detectLanguage

Finds the language of a given document and returns an Oracle language code (for example, `es` for Spanish). For accurate results, the text should contain at least ten words.

`detectLanguage`accepts the following parameter:

- `text`. This is the data in type String to perform language detection on.

extractKeyPhrases

Extracts key phrases from a String and returns a list of phrases. The function calculates key phrases using TF/IDF algorithm, which takes the total number of times each term appears within the String and offsets that value by the number of times it appears within a larger body of work. Offsetting the value helps filter out frequently-used terms like "the" and "it". The body of work used as the control is selected internally based on the String's language; for example, the model used for English is based on a New York Times corpus. The `extractKeyPhrases` function is a wrapper function for the TF/IDF Term extractor enrichment module.

The number of key phrases returned by `extractKeyPhrases` is a function of the TF/IDF curve. By default, it stops returning terms when the score of a given term falls below ~68%.

`extractKeyPhrases` accepts the following parameters:

- `text`. The text in type String that is to be processed. It is recommended that you convert the text to lowercase first, especially if it is in all caps.
- `language`. An optional parameter that specifies the language name or code (for example "en", "English", "German") to improve accuracy. Supported languages are English (UK/US), Portuguese (Brazilian), Spanish, French, German, and Italian. When specified it forces the function to use a model specific to that language. When not specified, or when passed as `null` (this is the default), the language is automatically detected.

Note: When you create a new attribute as a result of using this function, make sure the attribute is of type multi-assign.

extractNounGroups

Returns a String containing noun groups. A noun group is any noun, such as "movie" or "building". This is a wrapper function for the Noun Group Extractor enrichment module. This module finds and returns noun groups from a string attribute in each of the supported languages. It is used in tag cloud visualization, for finding commonly occurring themes in the data.

`extractNounGroups` accepts the following parameters:

- `text`. The String to be processed.
- `language`. An optional parameter that specifies the language name or code (for example "en", "English", "German") to improve accuracy. Supported languages are English (UK/US), Portuguese (Brazilian), Spanish, French, German, and Italian. When specified it forces the function to use a model specific to that language. When not specified, or when passed as `null` (this is the default), the language is automatically detected.

extractWhiteListTags

Uses a dictionary-matching algorithm that locates elements of a finite set of strings (the whitelist) within input text. The function finds all occurrences of any whitelist terms and returns a list of matching expansions. The input text is matched against a whitelist. A whitelist is newline-delimited. This is a wrapper function for the Whitelist Tagger enrichment module.

Each line may be either a comment (indicated with a # as the first character), or a matching directive comprised of either one or two values (separated by TAB). The second value is used to rewrite the match output.

Here is a simple example whitelist:

- helium
- neon
- argon
- krypton
- xenon
- radon

It could be rewritten as follows:

- heliumHe

- neonNe
- argonAr
- kryptonKr
- xenonXe
- radonRn

When this whitelist is run on the text "The only noble gas is radon", it would produce an output list of ['Rn']

`extractWhiteListTags` accepts the following parameters:

- `text`. The String to process.
- `whitelist`. A document containing whitelisted terms. This should be a plain text file containing a newline-delimited list of literals and configuration terms.
- `language`. An optional parameter that specifies the String's language to improve accuracy. Set to English by default. Supported languages are English (US/UK), Danish, German, Spanish, French, Italian, Japanese, Korean, Simplified Chinese, Traditional Chinese, and Portuguese (Brazilian).
- `caseSensitive`. Indicates whether input is case-sensitive (the default is `false`).
- `unbounded`. Indicates whether to match whole words only (when set to `false` which is the default), or parts of words (when set to `true`). Ensures that `"red"` does not match `"reduce"`.

geotagAddress*

A set of the following functions:

- **geotagAddressGetCity**
- **geotagAddressGetCountry**
- **geotagAddressGetGeocode**
- **geotagAddressGetPostcode**
- **geotagAddressGetRegion**
- **geotagAddressGetSubRegion**
- **geotagAddressGetRegionID**
- **geotagAddressGetSubRegionID**

Converts a valid address String to a Geocode object, such as city, country, geocode, postcode, region, subregion or region and subregion IDs. This is a wrapper function for the Address Geotagger data enrichment module. It adds a multi-assign attribute (column) to your data set that contains the following fields:

- `city`
- `country`
- `geocode` (the address's latitude and longitude coordinates)
- `latitude`
- `longitude`
- `population`
- `postal_code`

- `region`
- `sub_region`
- Geoname ID for the `region` or `sub_region`

`geoTagAddress*` accepts the following parameters:

- `arg1` `address`. The address String to process. This must be less than or equal to 350 characters.
- Map. This is a map of advanced options:
  - `PREFERRED_LEVEL`. An optional parameter in type String that specifies an administrative division to improve accuracy. This can be set to only one of the following values (case-insensitive):
    - `CITY`. Target for a city match.
    - `COUNTRY`. Target for a country match.
    - `REGION`. Target for a region match, such as "state" in the United States.
    - `SUB_REGION`. Target for a subregion match, such as "county".
    - `NONE`. If this value is used, the function returns the most populous location that most closely matches the address String. This is the default value.

    Note: Administrative divisions vary depending on the country, so the returned values may be different than expected. Also, if your input value is not in the acceptable list, an exception is thrown.

  - `STRICT_MODE`. An optional Boolean parameter that specifies how the function should handle ambiguous or improperly-formatted addresses, such as one that contains an incorrect postal code. This can be set to one of the following:
    - `true`. If the address is invalid, the function returns `null`.
    - `false`. If the address is invalid, the function returns the closest match. This is the default.

The following example shows how to specify these parameters for a function `geotagAddressGetSubRegion` in a map:

```
geotagAddressGetSubRegion (' 1 Main Street Cambridge', ['PREFERRED_LEVEL':'CITY',
'STRICT_MODE':true])
```

geotagIPAddressGetCity

Converts an IP address to a Geocode and returns its `city` field as an Object. This is a wrapper function for the IP Address Geotagger data enrichment module that returns a single value.

`geoTagIPAddressGetCity` accepts the following parameters:

- `IPAddress`. The IP address to process, in type String.
- `language`. An optional String parameter that specifies the output language. The default value is `null`, which sets the language to English.

geotagIPAddressGetCountry

Converts an IP address to a Geocode and returns its `country` field as an Object. This is a wrapper function for the IP Address Geotagger data enrichment module that returns a single entity type.

`geoTagIPAddressGetCountry` accepts the following parameters:

- `IPAddress`. The IP address to process, in type String.
- `language`. An optional String parameter that specifies the output language. The default value is `null`, which sets the language to English.

geotagIPAddressGetGeocode

Converts an IP address to a Geocode and returns its `geocode` field as an Object. This is a wrapper function for the IP Address Geotagger data enrichment module that returns a single entity type.

`geoTagIPAddressGetGeoCode` accepts the following parameters:

- `IPAddress`. The IP address to process, in type String.
- `language`. An optional String parameter that specifies the output language. The default value is `null`, which sets the language to English.

geotagIPAddressGetPostCode

Converts an IP address to a Postal Code and returns its `postal_code` field as an Object. This is a wrapper function for the IP Address Geotagger data enrichment module that returns a single entity type.

`geoTagIPAddressGetPostCode` accepts the following parameters:

- `IPAddress`. The IP address to process, in type String.
- `language`. An optional String parameter that specifies the output language. The default value is `null`, which sets the language to English.

geotagIPAddressGetRegion

Converts an IP address to a Geocode and returns its `region` field as an Object. This is a wrapper function for the IP Address Geotagger data enrichment module that returns a single entity type.

`geoTagIPAddressGetRegion` accepts the following parameters:

- `IPAddress`. The IP address to process, in type String.
- `language`. An optional String parameter that specifies the output language. The default value is `null`, which sets the language to English.

geotagIPAddressGetRegionID

Converts an IP address to a Geocode and returns its Geoname ID for the `region` field as an Object. This is a wrapper function for the IP Address Geotagger data enrichment module that returns a single entity type.

`geoTagIPAddressGetRegionID` accepts the following parameters:

- `IPAddress`. The IP address to process, in type String.
- `language`. An optional String parameter that specifies the output language. The default value is `null`, which sets the language to English.

geotagIPAddressGetSubRegion

Converts an IP address to a Geocode and returns its `sub_region` field as an Object. This is a wrapper function for the IP Address Geotagger data enrichment module that returns a single entity type.

`geoTagIPAddressGetSubRegion` accepts the following parameters:

- `IPAddress`. The IP address to process, in type String.
- `language`. An optional String parameter that specifies the output language. The default value is `null`, which sets the language to English.

geotagIPAddressGetSubRegionID

Converts an IP address to a Geocode and returns its Geoname ID for the `sub_region` field as an Object. This is a wrapper function for the IP Address Geotagger data enrichment module that returns a single entity type.

`geoTagIPAddressGetSubRegion` accepts the following parameters:

- `IPAddress`. The IP address to process, in type String.
- `language`. An optional String parameter that specifies the output language. The default value is `null`, which sets the language to English.

getLocationEntities

Returns all location entities within a String as an Object. Location entities are names of places, such as "Boston" or "Canada". This function creates a new multi-assign column in your data set. This is a wrapper function for the name Entity extractor data enrichment module that returns a single entity type.

`getLocationEntities` accepts the following parameter:

- `text`. The String to process.

getNegativeLocationEntitySentiment

Locates passages within a String that contain location entities and returns the negative sentiment of those passages as an Object.

`getNegativeLocationEntitySentiment` accepts the following parameters:

- `text`. The String to process.
- `language`. An optional parameter that specifies the language in type String to improve accuracy. If set to `null` (which is the default value), the language is automatically detected. Supported language is English only.

getNegativeNounGroupsSentiment

Locates passages within a String that contain noun groups and returns the negative sentiment of those passages as an Object.

`getNegativeNounGroupsSentiment` accepts the following parameters:

- `text`. The String to process.

- `language`. An optional parameter that specifies the language in type String to improve accuracy. If set to `null` (which is the default value), the language is automatically detected. Supported languages are English (UK/US), Portuguese (Brazilian), Spanish, French, German and Italian.

getNegativeOrganizationEntitySentiment

Locates passages within a String that contain organization entities and returns the negative sentiment of those passages as an Object.

`getNegativeOrganizationEntitySentiment` accepts the following parameters:

- `arg1`. The String to process.
- `language`. An optional parameter that specifies the String's language to improve accuracy. If set to `null` (which is the default value), the language is automatically detected. Supported language is English only.

getNegativePersonEntitySentiment

Locates passages within a String that contain person entities and returns the negative sentiment of those passages as an Object.

`getNegativePersonEntitySentiment` accepts the following parameters:

- `arg1`. The String to process.
- `language`. An optional parameter that specifies the String's language to improve accuracy. If set to `null` (which is the default value), the language is automatically detected. Supported language is English only.

getNegativeTFIDFSentiment

Extracts key phrases in sentences that have a negative sentiment.

`getNegativeTFIDFSentiment` accepts the following parameters:

- `arg1`. The String to process.
- `language`. An optional parameter that specifies the String's language to improve accuracy. If set to `null` (which is the default value), the language is automatically detected. Supported languages are English (UK/US), Portuguese (Brazilian), Spanish, French, German and Italian.

getOrganizationEntities

Returns an Object containing the organization entities found within a String. This is a wrapper function for the Name Entity extractor data enrichment module that returns a single entity type.

Note: This function creates a new multi-assign column in your data set.

`getOrganizationEntities` accepts the following parameter:

- `arg1`. The String to process.

getPersonEntities

Returns an Object containing the person entities found within a String. This is a wrapper function for the Name Entity extractor data enrichment module that returns a single entity type.

Note: This function creates a new multi-assign column in your data set.

`getPersonEntities` accepts the following parameter:

- `arg1`. The String to process.

getPositiveLocationEntitySentiment

Locates passages within a String that contain location entities and returns the positive sentiment of those passages as an Object.

`getPositiveLocationEntitySentiment` accepts the following parameters:

- `arg1`. The String to process.
- `language`. An optional parameter that specifies the String's language to improve accuracy. If set to `null` (which is the default value), the language is automatically detected. Supported language is English only.

getPositiveNounGroupsSentiment

Locates passages within a String that contain noun groups and returns the positive sentiment of those passages as an Object.

`getPositiveNounGroupsSentiment` accepts the following parameters:

- `arg1`. The String to process.
- `language`. An optional parameter that specifies the String's language to improve accuracy. If set to `null` (which is the default value), the language is automatically detected. Supported language is English only.

getPositivePersonEntitySentiment

Locates passages within a String that contain person entities and returns the positive sentiment of those passages as an Object.

`getPositivePersonEntitySentiment` accepts the following parameters:

- `arg1`. The String to process.
- `language`. An optional parameter that specifies the String's language to improve accuracy. If set to `null` (which is the default value), the language is automatically detected. Supported language is English only.

getPositiveOrganizationEntitySentiment

Locates passages within a String that contain organization entities and returns the positive sentiment of those passages as an Object.

`getPositiveOrganizationEntitySentiment` accepts the following parameters:

- `arg1`. The String to process.
- `language`. An optional parameter that specifies the String's language to improve accuracy. If set to `null` (which is the default value), the language is automatically detected. Supported language is English only.

getPositiveTFIDFSentiment

Extracts key phrases in sentences that have a positive sentiment.

`getNegativeTFIDFSentiment` accepts the following parameters:

- `arg1`. The String to process.
- `language`. An optional parameter that specifies the String's language to improve accuracy. If set to `null` (which is the default value), the language is automatically detected. Supported languages are English (UK/US), Portuguese (Brazilian), Spanish, French, German, and Italian.

getSentiment

Returns an Object containing the overall sentiment of a String. This is a wrapper function for the Sentiment Analysis (document level) data enrichment module. The String's sentiment can be one of the following:

- `POSITIVE`
- `NEGATIVE`

`getSentiment` accepts the following parameters:

- `arg1`. The String to process.
- `language`. An optional parameter that specifies the String's language to improve accuracy. Supported languages are English (UK/US), Portuguese (Brazilian), Spanish, French, German, and Italian. If set to `null` (which is the default value), the language is automatically detected.

reverseGeotagGetCity

Returns the `city` field from a Geocode as an Object. Searches for cities within the specified radius from the entered Geocode. This is a wrapper function for the Reverse Geotagger data enrichment module that returns a single value.

`reverseGeotagGetCity` accepts the following parameter:

- `geo`. The Geocode to process.
- `language`. An optional parameter that specifies the output language. The default value is `null`, which sets the output language to English.
- `proximityThreshold`. An optional parameter that specifies the maximum distance in miles allowed for input geocode and output geographic location. If this parameter is not specified, the default of 100 miles is used. If the distance exceeds the threshold, null is returned.

reverseGeotagGetCountry

Returns the `country` field from a Geocode as an Object. Searches for countries within the specified radius from the entered Geocode. This is a wrapper function for the Reverse Geotagger data enrichment module that returns a single value.

`reverseGeotagGetCountry` accepts the following parameter:

- `geo`. The Geocode to process.
- `language`. An optional parameter that specifies the output language. The default value is `null`, which sets the output language to English.

- `proximityThreshold`. An optional parameter that specifies the maximum distance in miles allowed for input geocode and output geographic location. If this parameter is not specified, the default of 100 miles is used. If the distance exceeds the threshold, null is returned.

reverseGeotagGetPostCode

Returns the `postal_code` field from a Geocode as an Object. Searches for post codes within the specified radius from the entered Geocode. This is a wrapper function for the Reverse Geotagger data enrichment module that returns a single value.

`reverseGeotagGetPostCode` accepts the following parameter:

- `geo`. The Geocode to process.
- `language`. An optional parameter that specifies the output language. The default value is `null`, which sets the output language to English.
- `proximityThreshold`. An optional parameter that specifies the maximum distance in miles allowed for input geocode and output geographic location. If this parameter is not specified, the default of 100 miles is used. If the distance exceeds the threshold, null is returned.

reverseGeotagGetRegion

Returns the `region` field from a Geocode as an Object. Searches for regions within the specified radius from the entered Geocode. This is a wrapper function for the Reverse Geotagger data enrichment module that returns a single value.

`reverseGeotagGetRegion` accepts the following parameter:

- `geo`. The Geocode to process.
- `language`. An optional parameter that specifies the output language. The default value is `null`, which sets the output language to English.
- `proximityThreshold`. An optional parameter that specifies the maximum distance in miles allowed for input geocode and output geographic location. If this parameter is not specified, the default of 100 miles is used. If the distance exceeds the threshold, null is returned.

reverseGeotagGetRegionID

Returns the Geoname region ID field from a Geocode of the `region` field as an Object. Searches for regions within the specified radius from the entered Geocode. This is a wrapper function for the Reverse Geotagger data enrichment module that returns a single value.

`reverseGeotagGetRegion` accepts the following parameter:

- `geo`. The Geocode to process.
- `language`. An optional parameter that specifies the output language. The default value is `null`, which sets the output language to English.
- `proximityThreshold`. An optional parameter that specifies the maximum distance in miles allowed for input geocode and output geographic location. If this parameter is not specified, the default of 100 miles is used. If the distance exceeds the threshold, null is returned.

reverseGeotagGetSubRegion

Returns the `sub_region` field from a Geocode as an Object. Searches for sub-regions within the specified radius from the entered Geocode. This is a wrapper function for the Reverse Geotagger data enrichment module that returns a single value.

`reverseGeotagGetSubRegion` accepts the following parameter:

- `geo`. The Geocode to process.
- `language`. An optional parameter that specifies the output language. The default value is `null`, which sets the output language to English.
- `proximityThreshold`. An optional parameter that specifies the maximum distance in miles allowed for input geocode and output geographic location. If this parameter is not specified, the default of 100 miles is used. If the distance exceeds the threshold, null is returned.

reverseGeotagGetSubRegionID

Returns the Geoname ID of the Geocode from the `sub_region` field as an Object. Searches for sub-regions within the specified radius from the entered Geocode. This is a wrapper function for the Reverse Geotagger data enrichment module that returns a single value.

`reverseGeotagGetSubRegion` accepts the following parameter:

- `geo`. The Geocode to process.
- `language`. An optional parameter that specifies the output language. The default value is `null`, which sets the output language to English.
- `proximityThreshold`. An optional parameter that specifies the maximum distance in miles allowed for input geocode and output geographic location. If this parameter is not specified, the default of 100 miles is used. If the distance exceeds the threshold, null is returned.

runExternalPlugin

Runs the external Groovy script as defined in an external file of `pluginName`, and returns the result of the script.

`runExternalPlugin` accepts the following parameters:

- `pluginName`. The name of the external plugin.
- `arg1`. An argument passed to the external plugin.

stripTagsFromHTML

Removes any HTML, XML and XHTML markup tags from the input String and returns the result as an Object. This is a wrapper function for the Tag Stripper data enrichment module.

`stripTagsFromHTML` accepts the following parameter:

- `arg1`. The HTML String to process.

toPhoneticHash

Produces a String hash of the input text (English only) that represents the phonetics of the text.

A word's phonetic hash is based on its pronunciation, rather than its spelling. One application for phonetic hashes is search engines. If a search term does not return any results, the search engine can compare the term's phonetic hash to the hashes of other terms, and return results for the term that is the best fit. For example, "purple" and "pruple" have the same phonetic hash (PRPL), so a search for the misspelled term "pruple" would still yield results for "purple".

`toPhoneticHash` accepts the following parameter:

- `arg1`. The String to process.

Geocode functions

Geocode functions perform different actions on Geocode objects, such as calculating the distance between two Geocode values or obtaining a Geocode's latitude coordinate.

This table describes the Geocode functions that **Transform** supports. The same functions are described in the *Transform API Reference* (Groovydoc).

Important: For those geocode functions where type Double is required for inputs, make sure you enter values that are within valid ranges. The range for valid latitude values is from -90.0 to 90.0; the range for valid longitude values is from -180.0 to 180.0. Also, note that geocode functions do not accept type Long.

User Function	Return Data Type	Description
`distance(Geocode geo1, Geocode geo2)`	`Double`	Calculates the distance between two Geocode values, in kilometers. Note that geocode functions do not accept type Long.
`getLatitude(Geocode geo)`	`Double`	Returns the latitude coordinate of a Geocode value. Note that geocode functions do not accept type Long.
`getLongitude(Geocode geo)`	`Double`	Returns the longitude coordinate of a Geocode value. Note that geocode functions do not accept type Long.
`isGeocode(String s)`	`Boolean`	Determines whether a String is a valid Geocode value.
`toGeocode(String s)`	`Geocode`	Converts a String to a Geocode value.
`toGeocode(Double lat, Double lon)`	`Geocode`	Converts a pair of latitude and longitude coordinates to a Geocode value. For the inputs to this function, ensure that you enter valid latitude and longitude values. The range for valid latitude values is from -90.0 to 90.0. The range for valid longitude values is from -180.0 to 180.0.

Math functions

Math functions perform mathematical operations on your data.

This table describes the math functions that **Transform** supports.

User Function	Return Data Type	Description

`abs(double d)`
`abs(float f)`
`abs(int i)`
`abs(long l)`
	`double`
	`float`
	`int`
	`long`
		Calculates the argument's absolute value.

`acos(double d)`	`double`	Calculates the arccosine of a double. The returned angle is between `0.0` and `pi`.
`asin(double d)`	`double`	Calculates the arcsine of a double. The returned angle is between `-pi/2` and `pi/2`.
`atan(double d)`	`double`	Calculates the arctangent of a double. The returned angle between `-pi/2` and `pi/2`.
`atan2(double y, double x)`	`double`	Calculates the angle theta from the conversion of rectangular coordinates `(x,y)` to polar coordinates `(r,theta)`.
`cbrt(double d)`	`double`	Calculates the cube root of a double.
`ceil(double d)`	`double`	Returns the smallest (i.e., closest to negative infinity) double value that is greater than or equal to the argument, and is equal to a mathematical integer.
`copySign(double a, double b)` `copySign(float a, float b)`	`double` `float`	Returns the first floating-point argument with the sign of the second floating-point argument.
`cos(double a)`	`double`	Calculates the trigonometric cosine of an angle.
`cosh(double d)`	`double`	Calculates the hyperbolic cosine of a double.
`exp(double d)`	`double`	Returns Euler's number *e* raised to the power of a double value.
`expm1(double x)`	`double`	Returns e^x`-1`.

User Function	Return Data Type	Description
`floor(double d)`	`double`	Returns the largest (i.e., closest to positive infinity) double value that is less than or equal to the argument, and is equal to a mathematical integer.
`getExponent(double d)`	`int`	Returns the unbiased exponent used in the representation of a double.
`hypot(double x, double y)`	`double`	Returns `sqrt(`$x^2 + y^2$`)` without intermediate overflow or underflow.
`log(double d)`	`double`	Returns the natural logarithm (base *e*) of a double.
`log10(double d)`	`double`	Returns the base 10 logarithm of a double.
`log1p(double d)`	`double`	Returns the natural logarithm of the sum of a double and 1.

`max(double a, double b)`
`max(float a, float b)`
`max(int a, int b)`
`max(long a, long b)`
	`double`
	`float`
	`int`
	`long`
		Returns the greater of the two arguments.

`min(double a, double b)`
`min(float a, float b)`
`min(int a, int b)`
`min(long a, long b)`
	`double`
	`float`
	`int`
	`long`
		Returns the lesser of the two arguments.

`nextAfter(double a, double b)` `nextAfter(float a, double b)`	`double` `float`	Returns the floating-point number adjacent to the first argument in the direction of the second.
`nextUp(double a)` `nextUp(float a)`	`double` `float`	Returns the floating-point value adjacent to the argument in the direction of positive infinity.
`pow(double a, double b)`	`double`	Returns the value of the first argument raised to the power of the second.
`rint(double a)`	`double`	Returns the double value that is closest in value to the argument and is equal to a mathematical integer.
`random()`	`double`	Returns a positive double value that is greater than or equal to 0.0 and is less than 1.0.

User Function	Return Data Type	Description
`round(double a)` `round(float a)`	`long` `int`	Returns the closest value to the argument, with ties rounding up.
`roundWithPrecision(double a, int b)`	`double`	Rounds `a` with the precision defined by `b`.
`scalb(double a, int b)` `scalb(float a, int b)`	`double` `float`	Returns $a \times 2^b$ rounded as if performed by a single, correctly-rounded floating-point multiply to a member of the float value set.
`signum(double a)` `signum(float a)`	`double` `float`	Returns the signum of the argument: 0 if the argument is 0, 1.0 if the argument is greater than 0, -1.0 if the argument is less than 0.
`sin(double a)`	`double`	Calculates the trigonometric sine of an angle.
`sinh(double a)`	`double`	Calculates the hyperbolic sine of the argument.
`sqrt(double a)`	`double`	Calculates the correctly-rounded positive square root of the argument.
`tan(double a)`	`double`	Calculates the trigonometric tangent of an angle.
`tanh(double a)`	`double`	Calculates the hyperbolic tangent of `a`.
`toRadians(double angle)`	`double`	Converts an angle measured in degrees to an approximately equivalent angle measured in radians.
`truncateNumber(double number, int precision)`	`double`	Truncates a number using the specified precision.
`ulp(double a)` `ulp(float a)`	`double` `float`	Returns the size of a ULP of the argument.

Set functions

Set functions perform various functions on values for multi-assign attributes, such as obtaining the size of the set, checking whether a set is empty, or converting an attribute from a multi-value to a single-value attribute.

This table describes the Set functions that **Transform** supports. The same functions are described in the *Transform API Reference* (Groovydoc).

User Function	Return Data Type	Description
`cardinality(Object dataSetValue)`	`Long`	Inputs a set of values on multi-assign attributes and obtains the size of that set. (A set of multi-assign attributes consists of all values assigned on multi-assign attributes in the Dgraph data set.) Works only on multi-value (multi-assign) attributes. Throws an exception if you run it on an attribute that is single-value (known also as "single-assign", in the Dgraph schema).
`isEmpty(Object dataSetValue)`	`Boolean`	Inputs a set of multi-assign attributes and checks whether this set is empty (has no assignments). Returns `true` if the set is empty. Works only on sets of multi-value (multi-assign) attributes. Throws an exception if you run it on an attribute that is single-value (known also as "single-assign", in the Dgraph schema).
`isMemberOf(Object dataSet, Object dataSetValue)`	`Boolean`	Checks whether the value belongs to the set of values in a multi-assign attribute set. Returns `true` if the value belongs to the set. Works only on multi-value (multi-assign) attributes. This function looks for an exact match: it is case-sensitive, and does not accept wildcards, or regular expressions. It throws an exception if you run it on an attribute that is single-value (known as "single-assign", in the Dgraph schema).
`isNull(Object object)`	`Boolean`	Determines whether an attribute has any assigned values, in this data set. Works on both multi-assign and single-assign attributes. Returns `true` if yes.
`isSet(Object column)`	`Boolean`	Checks if an attribute (column) is multi-value (also known as multi-assign). Returns `true` if yes. Works only on multi-value (multi-assign) attributes. Throws an exception if you run it on an attribute that is single-value (known as "single-assign", in the Dgraph schema).

User Function	Return Data Type	Description
`toSet(Object... dataSet)`	`Object[]`	Converts an Object argument list to an Object array. Works only on multi-value (multi-assign) attributes. Throws an exception if you run it on an attribute that is single-value (known as "single-assign", in the Dgraph schema).
`toSingle(Object column)`	`Object[]`	Converts a multi-value attribute into a single-value attribute. Returns a single value chosen randomly from a set of values for the attribute. Works only on multi-value (multi-assign) attributes. Throws an exception if you run it on an attribute that is single-value (known as "single-assign", in the Dgraph schema).

String functions

`String` functions perform different actions on Strings, such as converting an entire String to uppercase or removing whitespace from a String.

This table describes the String functions that **Transform** supports. The same functions are described in the *Transform API Reference* (Groovydoc).

User Function	Return Data Type	Description
`concat(String... arguments)`	`String`	Combines a list of String arguments into a single String.
`concatWithToken(String joinToken, String... arguments)`	`String`	Combines a list of String arguments into a single String using a join token. For example, `concatWithToken("|", "merlot", "cabernet", "malbec")` would return `"merlot|cabernet|malbec"`.
`contains(String originalString, String substring)`	`Boolean`	Determines whether a String contains a substring. For example, `contains("Boston", "Bos")` would return `true`.
`find(String originalString, String substring)`	`String`, `null`	Returns the first instance of a substring or regular expression within a String. Returns `null` if no match is found.
`findAll(String originalString, String substring)`	`String`, `null`	Returns a (possibly empty) list of all occurrences of a regular expression (in String format) found within a String.
`initRNG()`	`String`	Returns the initialized String.

User Function	Return Data Type	Description
`indexOf(String originalString, String substring)`	`Integer`	Returns the index of a substring within a String.
`isDouble(String s)`	`Boolean`	Determines whether a String is a `Double`.
`isInteger(String s)`	`Boolean`	Determines whether a String is an `Integer`.
`isLong(String s)`	`Boolean`	Determines whether a String is a `Long`.
`length(String s)`	`Integer`	Returns the length of a String.
`replace(String originalString, String oldExpression, String newString)`	`String`	Replaces every instance of a substring or regular expression within a String with a new text string.
`splitToSet(String originalString, String delimiter)`	`String`	Splits an original String based on a specified delimiter character.
`stripIndent(String s)`	`String`	Removes leading spaces from a String.
`substring(String s, Integer start, Integer end)`	`String`	Returns a substring from the original String, based on its start point and end point. For example, `substring("cabernet", 0, 2)` returns `"cab"`.
`substring(String s, Integer start)`	`String`	Returns a substring from the original String, based on its start point. The returned substring will be from the start point to the end of the original string. For example, `substring("cabernet", 5)` returns `"net"`.
`toLowerCase(String s, String locale)`	`String`	Converts a String to lowercase. You can optionally specify the String's locale; this defaults to `"en"`.
`toTitleCase(String s, String locale)`	`String`	Converts a String to title case. For example, `toTitleCase("sOMe STrING")` would return `"Some String"`. You can optionally specify the String's locale; this defaults to `"en"`.  **Note:** For lists of comma-separated values that don't include spaces, only the first item in the list will be converted to title case. For example, `toTitleCase("apple,cherry,plum")` returns `"Apple,cherry,plum"`.
`toUpperCase(String s, String locale)`	`String`	Converts a String to uppercase. You can optionally specify the String's locale; this defaults to `"en"`.
`trim(String s)`	`String`	Removes leading and trailing whitespace from a String.

Index

T

V