

# Creating Configuration Files

<!-- --> <!-- -->

## Table of contents

1 Creating Configuration Files.....	2
1.1 Creating the query-translator.py file.....	2
1.2 Formatting metadata reports.....	9
1.3 Creating the user.properties file.....	10

## 1. Creating Configuration Files

This chapter provides information to create the various configuration files required by the ADL software. This includes:

1. Creating the query-translator.py file
2. Creating the bucket99.conf file
3. Formatting metadata reports
4. Creating the user.properties file

### 1.1. Creating the query-translator.py file

This section provides information to help you create the query-translator.py file, which is a python script that provides the connection between the ADL query and the fieldnames in your database. The query-translator.py file is located in [tomcat]\webapps\mw\_version\_number\_wc\WEB-INF\config\collections\collection\_name. ADL queries are translated into SQL queries using the bucket definitions you provide in this file.

#### Introduction to ADL Query Translator

(based on <http://piru.alexandria.ucsb.edu:8888/opadl/95>)

The ADL middleware query translator transforms a query from the ADL query language (see <http://www.alexandria.ucsb.edu/middleware/dtds/ADL-query.dtd> ADL-[query.dtd](http://www.alexandria.ucsb.edu/middleware/dtds/ADL-query.dtd) (<http://www.alexandria.ucsb.edu/middleware/dtds/ADL-query.dtd>) ) into SQL queries that are native to the database management system (DBMS) used to access a collection or group of collections.

The ADL query translator consists of scripts written in the Python language which can be modified by collection developers to support his/her collections. The advantage of using Python to implement the query translation is that this scripting language is widely used, has a simple and concise syntax relative to other programming languages and can be readily modified by the collection developer.

The script query-translator.py is used to define the bucket mappings that are supported by a collection. The following table shows the nine standard ADL query buckets:

ADL standard query buckets	
Name	Type
adl:geographic locations	spatial
adl:dates	temporal

## Creating Configuration Files

adl:types	hierarchical
adl:formats	hierarchical
adl:title	textual
adl:subject-related text	textual
adl:assigned terms	textual
adl:originators	textual
adl:identifiers	identification

**Table 1: Table1**

It is not necessary to define all nine of the standard ADL buckets, however a query will fail if an ADL query is used that contains a constraint that corresponds to a bucket that is not defined, so the collection developer needs to consider what types of queries are supported by the item-level metadata for a collection and what types of queries should be supported for a collection.

The collection developer is not limited to the nine ADL standard buckets, as additional buckets can be defined for any collection.

### ADL query translator paradigms

Each bucket uses paradigms to define the mechanics of constraint-to-SQL translation. For example, for textual buckets, the following paradigms are provided: Textual\_Constant, Textual\_InformixVerity, Textual\_LikeDelimitedSubstring, Textual\_LikeSubstring. These paradigms correspond to Python scripts, located in the ./modules/paradigms directory, that implement the mechanics of the translation. For complete information on query translator paradigms, see the

<http://www.alexandria.ucsb.edu/research/docs/happyDocs/queryTranslator/modules/index.html>

[ADL Query Translator Guide](#)

(<http://www.alexandria.ucsb.edu/research/docs/happyDocs/queryTranslator/modules/index.html>)

.

Each bucket definition may support field-level constraints (see

<http://www.alexandria.ucsb.edu/research/docs/happyDocs/queryTranslator/modules/UniversalTranslator.htm>

[UniversalTranslator.py](#)

(<http://www.alexandria.ucsb.edu/research/docs/happyDocs/queryTranslator/modules/UniversalTranslator.htm>

) using adapter paradigms. For example, the Adapter\_Concatenation.py paradigm can be used to define a bucket that searches over multiple metadata fields where each field can use a different paradigm to define the mechanics of how each field in the bucket is searched. An individual field can be searched by specifying a field in the ADL query.

The script [UniversalTranslator.py](#)

(<http://www.alexandria.ucsb.edu/research/docs/happyDocs/queryTranslator/modules/UniversalTranslator.htm>  
<http://www.alexandria.ucsb.edu/research/docs/happyDocs/queryTranslator/modules/UniversalTranslator.htm>

uses the bucket definitions in `query-translator.py` and calls the paradigms specified in the bucket definitions to construct SQL statements. The SQL statements obtained from each bucket are joined together to obtain an SQL statement that is the complete translation of the ADL query that was passed to the middleware that can be sent to a collection DBMS. The following summarizes the paradigms that are currently available.

### Adaptor\_Concatenation.py

An adaptor that adds support for bucket-level textual searching to a set of paradigms (the “underlying” paradigms), each of which supports a specific field-level textual search, by treating a bucket-level search as a virtual search over the logical concatenation of the field-level textual content. Used by textual bucket type.

### Adaptor\_Constant.py

An adaptor that adds support for field-level searching to a paradigm (the “underlying” paradigm) in a trivial way by equating bucket-level and field-level searches. Used by any bucket type.

### Adaptor\_DirectQualification.py

An adaptor that adds support for field-level searching to a paradigm (the “underlying” paradigm) by adding a condition to each SELECT statement returned by the paradigm. Used by any bucket type.

### Adaptor\_IndirectQualification.py

An adaptor that adds support for field-level searching to a paradigm (the “underlying” paradigm) by adding a condition to each SELECT statement returned by the paradigm. Used by any bucket type.

### Adaptor\_IndivisibleConcatenation

An adaptor that adds support for bucket-level textual searching to a set of constituent paradigms (the “underlying” paradigms), each of which supports some kind of textual search, by treating a bucket-level search as a virtual search over the logical concatenation of the constituent textual content.

This paradigm is identical to `Adaptor_Concatenation` except that in the latter paradigm field-level search is supported, and the underlying paradigms are explicitly associated with fields; this paradigm supports bucket-level search only. As a consequence, the underlying paradigms are specified as a simple list. Used by textual bucket type.

### Adaptor\_Relationship.py

Adds an inner equijoin relationship to queries returned by an underlying paradigm. This

## *Creating Configuration Files*

adaptor is useful in situations in which a bucket is implemented by placing a constraint against a database entity that has an indirect relationship to collection items. Used by any bucket type.

### **Adaptor\_Switch.py**

An adaptor that supports both bucket-level and field-level searching by directing bucket-level constraints to one underlying paradigm and field-level constraints to another. Thus this paradigm is useful when different query translation strategies are desired for different levels of searches. Used by any bucket type.

### **Adaptor\_TemporallsContainedInRewriter.py**

Adds support for the “is-contained-in” operator to a temporal paradigm (the “underlying” paradigm) that doesn't otherwise support it. Used by any bucket type.

### **Adapter\_TermMapping.py**

No description in happyDocs.

### **Adaptor\_Union.py**

An adaptor that adds support for bucket-level searching to a set of paradigms (the “underlying” paradigms), each of which supports a specific field-level search, by treating a bucket-level search as the union of the field-level searches. Used by any bucket type.

### **Hierarchical\_Constant.py**

Translates a hierarchical constraint to a constant TRUE or FALSE depending on whether the constraint term matches one of a constant set of terms. Used by hierarchical bucket type.

### **Hierarchical\_IntegerSet.py**

Translates a hierarchical constraint to an SQL set inclusion test. Used by hierarchical bucket type.

### **Hierarchical\_StringEquality.py**

No description in happyDocs

### **Identification\_Integer.py**

Translates an identification constraint to an SQL string equality test. Used by identification bucket type.

### **Identification\_String.py**

Translates an identification constraint to an SQL string equality test. Used by identification bucket type.

### **Numeric\_Straight.py**

Translates a numeric constraint to an SQL numeric comparison. Used by numeric bucket type.

### **Spatial\_BoxCoordinatesNoCrossing.py**

Translates a box spatial constraint to a boolean combination of one or more SQL numeric comparisons assuming that object footprints are boxes described by four bounding coordinates. Used by spatial bucket type.

#### `Spatial_BoxCoordinatesNoCrossing_0_360.py`

Translates a box spatial constraint to a boolean combination of one or more SQL numeric comparisons assuming that object footprints are boxes described by four bounding coordinates. NOTE: this paradigm is identical to 'Spatial\_BoxCoordinatesNoCrossing', except that here we expect 'eastColumn' and 'westColumn' to be in the range [0,360]. Used by spatial bucket type.

#### `Spatial_InformixGeodetic.py`

Translates a spatial constraint to an Informix Geodetic DataBlade function call. Used by spatial bucket type.

#### `Spatial_InformixMapInfo.py`

Translates a spatial constraint to an Informix MapInfo function call. Used by spatial bucket type.

#### `Spatial_PostgreSQLBoxNoCrossing.html`

No description in happyDocs

#### `Temporal_BeginEnd.py`

Translates a temporal constraint to a pair of SQL date comparisons. This paradigm assumes that a collection item's temporal footprint is described by a begin date and an end date. Used by temporal bucket type.

#### `Temporal_IntegerYear.py`

Translates a temporal constraint to a pair of SQL numeric comparisons. This paradigm assumes that a collection item's temporal footprint is described by a single integer year. Used by temporal bucket type.

#### `Temporal_SingleDate.py`

Translates a temporal constraint to a pair of SQL date comparisons. This paradigm assumes that a collection item's temporal footprint is described by a single date. Used by temporal bucket type.

#### `TextUtils.py`

Handy character mappings and character delete lists. Used by temporal bucket type.

#### `Textual_Constant.py`

Translates a textual constraint to a constant TRUE or FALSE depending on whether the constraint text matches a body of constant text. Used by textual bucket type.

#### `Textual_InformixVerity.py`

## *Creating Configuration Files*

Translates a textual constraint to an Informix Verity DataBlade function call. Used by textual bucket type.

### `Textual_LikeDelimitedSubstring.py`

Translates a textual constraint to a boolean combination of one or more substring matches using SQL LIKE operators. Used by textual bucket type.

### `Textual_LikeSubstring.py`

Approximates a textual constraint with substring matching using SQL LIKE operators. Used by textual bucket type.

### `Textual_MySQLFulltext.py`

Translates a textual constraint to a MySQL full text index search. Used by textual bucket type.

### `Unsupported.py`

Indicates a bucket is unsupported by translating any constraint against the bucket to a constant FALSE. Used by any bucket type.

## Examples

(based on

[http://piru.alexandria.ucsb.edu:8888/opadl/uploads/124/query-translator.py\\_new\\_style.txt](http://piru.alexandria.ucsb.edu:8888/opadl/uploads/124/query-translator.py_new_style.txt))

(Is this a good example? Others?)

## Query Translator Configuration Guide

Link to happyDocs

(<http://www.alexandria.ucsb.edu/research/docs/happyDocs/queryTranslator/modules/index.html>)

## Examples of using different paradigms

(based on <http://piru.alexandria.ucsb.edu:8888/opadl/129>)

The use of the following paradigms are discussed:

1. bucket adaptors
2. db support adaptors
3. Standard Schema Support adaptors
4. data type support
  - spatial
  - temporal
  - text
  - hierarchical
  - numeric
  - identifier

Bucket Adaptors Buckets searching single values

- Bucket searches a single data type (field/column), and fielded searching is not needed. Use just the paradigm:

```
UT.Bucket("table","id",P.paradigm(...))
```

- Bucket searches a single data type (field/column), and fielded searching is desired. Use the Adaptor [Constant paradigm](#)

(<http://www.alexandria.ucsb.edu/research/docs/happyDocs/queryTranslator/modules/paradigms/Adaptor>)  
:

```
UT.Bucket("table","id", Adaptor_Constant ("field name", P.paradigm(...)) )
```

This is just two parameters, so the separator between the field name and the paradigm is a comma.

#### Buckets searching multiple values

- Bucket searches a multiple data type (field/column), fielded searching is desired, and any success is success for every field (is this correct greg?). Use the Adaptor [Union paradigm](#)

(<http://www.alexandria.ucsb.edu/research/docs/happyDocs/queryTranslator/modules/paradigms/Adaptor>)  
:

```
UT.Bucket("table","id",Adaptor_Union({DICTIONARY of Paradigms}) )
```

The dictionary looks like {"field 1": Paradigm1, "field 2": Paradigm2}

- Text Bucket searches a multiple data type (field/column), and fielded searching is desired. Use the Adaptor [Concatenation paradigm](#)

(<http://www.alexandria.ucsb.edu/research/docs/happyDocs/queryTranslator/modules/paradigms/Adaptor>)  
:

```
UT.Bucket("table","id",Adaptor_Concatenation({DICTIONARY of Paradigms}) )
```

Dictionary looks like {"field 1": Paradigm1, "field 2": Paradigm2}

- Text Bucket searches a multiple data type (field/column), and fielded searching is not needed. Use the Adaptor [IndivisibleConcatenation paradigm](#)

(<http://www.alexandria.ucsb.edu/research/docs/happyDocs/queryTranslator/modules/paradigms/Adaptor>)  
:

```
UT.Bucket("table","id",P.Adaptor_IndivisibleConcatenation([LIST of Paradigms]))
```

The list looks like [Paradigm1, Paradigm2]

#### Query different columns for bucket versus field-level searching

You want to use a different datatype adaptor for the bucket versus field-level searching. An example of this is text searching. For text searching, you may want to support searching over the aggregate at the bucket level, and search individual fields

Many full-text engines support only the use of a single text index in a query, so using a concatenation or union of the paradigms will cause performance problems. For the bucket, you would search across the index generated over all the supported columns (or a single aggregated column), and for fielded searching, you search the individual indexes.

## Creating Configuration Files

- Searches a multiple data type (field/column), and fielded searching is desired, and any success is success for every field (is this correct greg?) Use the Adaptor\_ [Switch](#) (<http://www.alexandria.ucsb.edu/research/docs/happyDocs/queryTranslator/modules/paradigms/Adaptor.paradigm>):  
UT.Bucket("table", "id", Adaptor\_Switch(bucketParadigm, fieldParadigm) )  
UT.Bucket("table", "id", Adaptor\_Switch(bucketParadigm, {DICTIONARY of Paradigms}) )

Dictionary looks like {"field 1": Paradigm1, "field 2": Paradigm2)

DB Support Adaptors A join relationship is needed to get the actual item holding id

- The table to search does not contain the holding id, but a related table does. Use the Adaptor\_ [Relationship paradigm](#) ([http://www.alexandria.ucsb.edu/research/docs/happyDocs/queryTranslator/modules/paradigms/Adaptor.P.Adapter\\_Relationship](http://www.alexandria.ucsb.edu/research/docs/happyDocs/queryTranslator/modules/paradigms/Adaptor.P.Adapter_Relationship) (table, idColumn, joinColumn, cardinality, paradigm) External Table contains the actual identifier

External Table contains the actual identifier

- This is not really a paradigm, but is a translator support function that needs to be included in the query-translator.py:

As noted in the UniversalTranslator documentation, a Translator object can optionally be initialized with an ExternalIdTable object that describes a table that maps internal identifiers to external identifiers. The object has the structure

ExternalIdTable (table, internalIdColumn, externalIdColumn)

For example, a configuration

```
translator = UT.Translator(buckets, UT.ExternalIdTable("id_map", "int_id", "ext_id"))
```

Universal Schema support

adaptors Adaptor\_DirectQualification.py Adaptor\_IndirectQualification.py Datatype

Paradigms ADL Query Language

For brief semantic definitions of the nine standard ADL buckets, see

<http://www.alexandria.ucsb.edu/middleware/doc/client-interfaces.html#query-language>. For

definitions of the seven bucket types supported by ADL, see

<http://www.alexandria.ucsb.edu/middleware/dtds/ADL-query.dtd>.

Creating the bucket99.conf file This section helps you create the bucket99.conf file, which is a java properties file for configuring the database URL, collection driver and metadata driver. The bucket99.comf file is located in

[tomcat]\webapps\mw\_version\_number\_wc\WEB-INF\config\collections\collection\_name.

## 1.2. Formatting metadata reports

The report-template files are located in

[tomcat]\webapps\mw\_version\_number\_wc\WEB-INF\config\collections\collection\_name.

Creating access-report-template.xml

TBD

Creating browse-report-template.xml

TBD

Creating bucket-report-template.xml

TBD

### **1.3. Creating the user.properties file**

(is this file used in MW2?)

What about webclient.properties and collection.properties, do they have to be edited?