XMLLoader

===============

Feeding the Pig with XML

Its always tough to parse XML, especially when it comes to PIG. Here I am explaining two approaches to parse an XML file in PIG.

1. Using Regular Expression

2. Using XPath

For simplicity I am taking a sample XML as shown below. This file should be in HDFS for processing.

<CATALOG>

<BOOK>

<TITLE>Hadoop Defnitive Guide</TITLE>

<AUTHOR>Tom White</AUTHOR>

<COUNTRY>US</COUNTRY>

<COMPANY>CLOUDERA</COMPANY>

<PRICE>24.90</PRICE>

<YEAR>2012</YEAR>

</BOOK>

<BOOK>

<TITLE>Programming Pig</TITLE>

<AUTHOR>Alan Gates</AUTHOR>

<COUNTRY>USA</COUNTRY>

<COMPANY>Horton Works</COMPANY>

<PRICE>30.90</PRICE>

<YEAR>2013</YEAR>

</BOOK>

</CATALOG>

Using Regular Expressions

===========================

REGISTER piggybank.jar

A = LOAD 'xmls/hadoop\_books.xml' using org.apache.pig.piggybank.storage.XMLLoader('BOOK') as (x:chararray);

B = foreach A GENERATE FLATTEN(REGEX\_EXTRACT\_ALL(x,'<BOOK>\\s\*<TITLE>(.\*)</TITLE>\\s\*<AUTHOR>(.\*)</AUTHOR>\\s\*<COUNTRY>(.\*)</COUNTRY>\\s\*<COMPANY>(.\*)</COMPANY>\\s\*<PRICE>(.\*)</PRICE>\\s\*<YEAR>(.\*)</YEAR>\\s\*</BOOK>'));

dump B;

Output:

==============

(Hadoop Defnitive Guide,Tom White,US,CLOUDERA,24.90,2012)

(Programming Pig,Alan Gates,USA,Horton Works,30.90,2013)

Using XPath

================

REGISTER piggybank.jar

DEFINE XPath org.apache.pig.piggybank.evaluation.xml.XPath();

A = LOAD 'xmls/hadoop\_books.xml' using org.apache.pig.piggybank.storage.XMLLoader('BOOK') as (x:chararray);

B = FOREACH A GENERATE XPath(x, 'BOOK/AUTHOR'), XPath(x, 'BOOK/PRICE');

dump B;

Output:

================

(Tom White,24.90)

(Alan Gates,30.90)

Using Apache Log Data

------------------------

You can process Apache logs (both in Common Log Format and Combined Log Format)

Loading Common Log Format

=====================

You can process Apache logs (both in [Common Log Format](http://httpd.apache.org/docs/current/logs.html#common) and[Combined Log Format](http://httpd.apache.org/docs/current/logs.html#combined)) with Mortar by using the [Piggybank](http://help.mortardata.com/technologies/pig/pig_help_and_resources) functions CommonLogLoader and CombinedLogLoader.

Loading Common Log Format

You can use CommonLogLoader as such:

data = **LOAD** 's3n://path/to/input'

**USING** org.apache.pig.piggybank.storage.apachelog.CommonLogLoader()

**AS** (addr: **chararray**, logname: **chararray**, user: **chararray**, time: **chararray**,

method: **chararray**, uri: **chararray**, proto: **chararray**,

status: **int**, bytes: **int**);

For example, the log line

81.19.151.110 - - [017/Jan/2013:13:08:02 +0000] "GET /api/function HTTP/1.1" 200 156

Will be loaded as:

(81.19.151.119,-,-,17/Jan/2013:13:08:02 +0000,GET,/api/function,HTTP/1.1,200,156)

Loading Combined Log Format

You can use CombinedLogLoader as such:

data = **LOAD** 's3n://path/to/input'

**USING** org.apache.pig.piggybank.storage.apachelog.CombinedLogLoader()

**AS** (addr: **chararray**, logname: **chararray**, user: **chararray**, time: **chararray**,

method: **chararray**, uri: **chararray**, proto: **chararray**,

status: **int**, bytes: **int**, referer: **chararray**, userAgent: **chararray**);

For example, the log line

81.19.151.110 - - [17/Jan/2013:13:08:02 +0000] "GET /api/function HTTP/1.1" 200 156 "-" "Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10\_5\_4; en-us) AppleWebKit/525.18 (KHTML, like Gecko) Version/3.1.2 Safari/525.20.1"

Will be loaded as:

(81.19.151.119,-,-,17/Jan/2013:13:08:02 +0000,GET,/api/function,HTTP/1.1,200,156,-,Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10\_5\_4; en-us) AppleWebKit/525.18 (KHTML, like Gecko) Version/3.1.2 Safari/525.20.1)

Loading Custom Log Formats

You can load any regular-expression parseable format using the Piggybank function MyRegExLoader. Pass it a pattern, and each regex group (sections of the pattern enclosed in parentheses) matched will be returned to you as a chararray.

For example, you could load data in the format 01234: string - string

data = **LOAD** 's3n://path/to/input'

**USING** org.apache.pig.piggybank.storage.MyRegExLoader(

'(\\d{5}):\\s\*([a-zA-Z]\*)\\s-\\s([a-zA-Z]\*)'

) **AS** (id: **chararray**, string1: **chararray**, string2: **chararray**);

Note that special characters must be double-escaped (two backslashes instead of one).

**Using CSV Data**

You can load CSV data into Mortar, or tell Mortar to output data in Excel-flavoured CSV format, by using the [Piggybank](http://help.mortardata.com/technologies/pig/pig_help_and_resources) function CSVExcelStorage.

Loading Data from CSV files

You can use CSVExcelStorage to load your data as such:

data = **LOAD** 's3n://my-s3-bucket/path/to/csv/file'

**USING** org.apache.pig.piggybank.storage.CSVExcelStorage()

**AS** (field1: **int**, field2: **chararray**);

Additionally, Mortar has extended CSVExcelStorage to take several parameters that the basic Piggybank version doesn't have. These parameters help tailor the loader to your type of CSV.

The first parameter specifies which character to use as a field delimiter. The default is comma (',').

The second parameter specifies how to treat quoted fields with newlines in them. These fields are valid CSV, but loading them properly is complicated and decreases performance. Therefore, the default behavior is 'NO\_MULTILINE', not allowing multiline fields. If you wish to allow multiline fields, set this parameter to 'YES\_MULTILINE'.

The third parameter specifies how to handle line endings for storing, and is not used for loading. Set it to 'NOCHANGE' if you want to use the fourth parameter. The parameters are in this unintuitive order to maintain backwards compatibility with an older version of CSVExcelStorage in the Piggybank not developed by Mortar.

The fourth parameter specifies what to do with header rows (the first row of each file). For loading, set it to 'SKIP\_INPUT\_HEADER' to skip header rows. If this parameter is not specified, header rows will be read.

Example of using parameters to load a TSV file (CSV with tab delimiters), skipping header rows:

data = **LOAD** 's3n://my-s3-bucket/path/to/csv/output'

**USING** org.apache.pig.piggybank.storage.CSVExcelStorage(

'\t', 'YES\_MULTILINE', 'NOCHANGE', 'SKIP\_INPUT\_HEADER'

);

Storing Pig Output in CSV Format

You can store the output of a Pig script in CSV format using CSVExcelStorage as well.

Example:

**STORE** result **INTO** 's3n://my-s3-bucket/path/to/output' **USING** org.apache.pig.piggybank.storage.CSVExcelStorage();

Again, Mortar's version of CSVExcelStorage can optionally be passed parameters.

The first parameter specifies which character to use as a field delimiter. The default is comma (',').

The second parameter controls how to handle newlines in quoted fields in the same manner as it did for loading. The default is 'NO\_MULTILINE'.

The third parameter controls how to write line endings. It may be set to 'UNIX' (line endings are LF), 'WINDOWS' (line endings are CRLF), or 'UNCHANGED' (line endings are the system default). The default is 'UNCHANGED'.

The fourth parameter specifies whether to write a header row containing the title of each field in the schema. Set it to 'WRITE\_OUTPUT\_HEADER' to write such a header to each file. If this parameter is not specified, no header row will be written.

Example of using parameters to store into a TSV format (CSV with tab delimiters), with Windows line-endings and with a header row:

data = **STORE** result **INTO** 's3n://my-s3-bucket/path/to/output'

**USING** org.apache.pig.piggybank.storage.CSVExcelStorage(

'\t', 'YES\_MULTILINE', 'WINDOWS', 'WRITE\_OUTPUT\_HEADER'

);

You can read Pig output in Excel on Windows if you use 'WINDOWS' line endings. You will have to give the output files a ".csv" extension to do so however.

You may also wish to concatenate the part files into a single file. Alternatively, you can add the clause "PARALLEL 1" to your store statement. This tells Pig to use only a single reducer, so the output will be sent to only one file.

**Storing Data to DynamoDB**

Storing Data Into DynamoDB

As a note, your DynamoDB database must currently be located in US-EAST-1.

To store data into your DynamoDB database, customize the following template STORE statement:

*-- Percentage of the table's write throughput to use*

*-- Default: 0.5, valid Range 0.1 - 1.5*

**SET** dynamodb.throughput.write.percent **1.0**;

*-- Must disable Pig's MultiQuery optimization*

*-- when using DynamoDBStorage*

**SET** opt.multiquery false;

*-- Dynamo table where writes should go*

**%default** DYNAMODB\_TABLE 'my\_dynamo\_db\_table';

*-- AWS access key to use for writing to DynamoDB table*

**%default** DYNAMODB\_AWS\_ACCESS\_KEY\_ID 'SETME';

*-- AWS secret key to use for writing to DynamoDB table*

**%default** DYNAMODB\_AWS\_SECRET\_ACCESS\_KEY 'SETME';

*-- Load up some input data*

input\_data = **LOAD** '$INPUT\_PATH'

**USING** PigStorage()

**AS** (...);

*-- Select exactly the fields you want to store to dynamodb.*

*-- MUST include your DynamoDB table's primary key.*

exact\_fields\_to\_store = **FOREACH** input\_data

**GENERATE** my\_field **AS** name\_in\_dynamodb\_1,

my\_field\_2 **AS** name\_in\_dynamodb\_2,

my\_field\_3 **AS** name\_in\_dynamodb\_3;

*-- Store the data to DynamoDB*

**STORE** exact\_fields\_to\_store

**INTO** 's3://somewhere-in-s3-i-can-write/but-will-not-get-written'

**USING** com.mortardata.pig.storage.DynamoDBStorage('$DYNAMODB\_TABLE', '$DYNAMODB\_AWS\_ACCESS\_KEY\_ID', '$DYNAMODB\_AWS\_SECRET\_ACCESS\_KEY');

You'll need to edit the DYNAMODB\_TABLE parameter to point to your database, and edit the DYNAMODB\_AWS\_ACCESS\_KEY\_ID andDYNAMODB\_AWS\_SECRET\_ACCESS\_KEY parameters with AWS keys that can write to your DynamoDB instance.

Pig requires a location for the INTO expression, but no data will be written there. Pig will, however, check that this location is accessible and does not have data. Best practice is to pass an S3 URL that is accessible but has no data.

Schema

Any fields passed to the DynamoDBStorage STORE statement will be stored into DynamoDB. Be sure to put a FOREACH statement in front of your STORE to prune down to just the fields you want and set their field names accordingly.

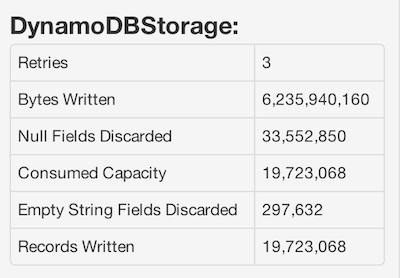
DynamoDB's [data model](http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DataModel.html) does not allow item fields to have null or empty string values. If DynamoDBStorage sees an empty or null value, it will omit that field for the given record. The rest of the record's data will be stored as usual.

Controlling Write Throughput

You can control what percentage of your DynamoDB table's [provisioned write throughput](http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ProvisionedThroughputIntro.html) will be consumed via the dynamodb.throughput.write.percentparameter. DynamoDBStorage will spread this throughput evenly across all running tasks in Hadoop, and will requery for the current throughput as each new task starts (in case you increase or reduce write throughput on the table).

Metrics

As it runs, DyanmoDBStorage will write out metrics via Hadoop counters. These metrics are visible in the Hadoop Cluster JobTracker or via the Mortar Job Details "Visualize" tab:



Collected metrics include:

* **Retries**: Number of times a batch of items must be retried, usually due to hitting a capacity threshold. If you see retries frequently, lower yourdynamodb.throughput.write.percent or provision additional write capacity on your table.
* **Bytes Written**: Bytes of data written to DynamoDB. (Note that this is less than capacity consumed, which includes metadata sent with the data).
* **Null Fields Discarded**: Number of fields with a null value that were omitted from an item
* **Empty String Fields Discarded**: Number of fields with a empty string value that were omitted from an item
* **Consumed Capacity**: Total number of DynamoDB write capacity units consumed
* **Records Written**: Total number of records written to DynamoDB

Loading Data from Fixed-Width Files

You can use FixedWidthLoader to load your data as such:

data = **LOAD** 's3n://my-s3-bucket/path/to/input'

**USING** org.apache.pig.piggybank.storage.FixedWidthLoader(

'-5, 7-10, 11-15, 16, 17-', 'SKIP\_HEADER',

'f1: int, f2: int, f3: float, f4: int, f5: chararray'

);

The first parameter is mandatory and specifies the positions of the columns. They are 1-indexed and inclusive on both ends. "-5" means columns 1 through 5, and "17-" means 17 to the end of the line. Single-character columns at position *n* can be specified as either *n*-*n* or simply *n*.

This syntax is intended to mimic that of the Unix utility "cut".

The second parameter is optional and specifies what to do with header rows (a first row containing the titles of each column). If the parameter is set to 'SKIP\_HEADER', FixedWidthLoader will skip the header row of each input file. The default behavior is to not skip the header; if you need to explicitly state this, set the parameter to 'USE\_HEADER'.

The third parameter is optional and allows you to specify a schema. You could alternatively use an AS clause, but specifying it like this allows the FixedWidthLoader to perform type coercions that Pig does not natively support, such as "17" to 17.

Storing Pig Output into Fixed-Width Format

We're not sure why you'd want to do this, but we support it anyway!

You can use FixedWidthStorer to store data as such:

**STORE** data **INTO** 's3n://my-s3-bucket/path/to/output'

**USING** org.apache.pig.piggybank.storage.FixedWidthStorer(

'-5, 7-10, 11-15, 16, 17-30', 'WRITE\_HEADER'

);

The first parameter is mandatory and specifies the positions of the columns using the same syntax as FixedWidthLoader. Note that the last column must have a defined end though; "17-" will cause an error.

The second parameter is optional and specifies whether or not to write a header row containing the name of each field at the top of each output file. The default behavior, 'NO\_HEADER' is not to write headers. Set this parameter to 'WRITE\_HEADER' to write headers. If you choose to write headers, make sure you allocate enough space in each column to fit the name of its field.

If a field does not fit in the space allotted to it, a null will be written (all spaces). The exception is floats and doubles. If a float or double field is too large to fit into a column, and the column is wide enough to fit all of the digits in the decimal left of the decimal point, the point itself, and at least one digit to the right of it, FixedWidthStorer will round the field to fit into the column.

Loading JSON Data with JsonLoader

Precursor

To load JSON data into Mortar, first ensure that your files are formatted with exactly one JSON object per line. This helps Pig and Hadoop to properly split your files for parallel processing.

Quick Start - No Schema

The simplest way to use the JsonLoader is with no schema definition:

json\_objects = **LOAD** 's3n://my-s3-bucket/path/to/json/file'

**USING** org.apache.pig.piggybank.storage.JsonLoader();

This loads each JSON document into a field called object with a Pig data type ofmap[]. All of the top-level keys in your JSON object will appear in that map.

This will give you a quick overview of your data, but Pig won't know the data types of your fields.

Providing a Schema

Once you know which fields you need from your data, you can provide JsonLoader with a schema. You pass a schema to JsonLoader as a string parameter, like this:

json\_objects = **LOAD** 's3n://my-s3-bucket/path/to/json/file'

**USING** org.apache.pig.piggybank.storage.JsonLoader(

'a: int, b: tuple(i: int, j: int), c: bag{t: tuple(i: int, j: int)}'

);

Note that this schema is passed as a parameter, *not* put into the LOAD statement AS clause.

JsonLoader will extract these fields from your data. If a field does not parse correctly, an error will be logged and the field will be read as null. Any fields that can't be found in the data will also be read as null.

See below for additional guidelines to defining your schema.

Simple Example

Let's read in some [tweets](https://dev.twitter.com/docs/platform-objects/tweets) pulled down by the [Mortar twitter gardenhose](https://github.com/mortardata/twitter-gardenhose) as a simple example.

To load the tweets without a schema, we'd use:

tweets = **LOAD** 's3n://twitter-gardenhose-mortar/tweets'

**USING** org.apache.pig.piggybank.storage.JsonLoader();

However, let's provide a schema to get just the fields we want with their data types:

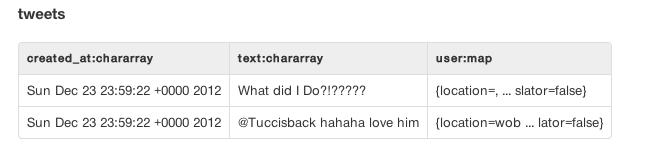
tweets = **LOAD** 's3n://twitter-gardenhose-mortar/tweets'

**USING** org.apache.pig.piggybank.storage.JsonLoader(

'created\_at:chararray, text:chararray, user:map[]'

);

If you illustrate this script, you should see something like:



We can now use the fields in the script as we please, say to find ones that include exclamation marks:

tweets = **LOAD** 's3n://twitter-gardenhose-mortar/tweets'

**USING** org.apache.pig.piggybank.storage.JsonLoader(

'created\_at:chararray, text:chararray, user:map[]'

);

exclamatory\_tweets = **FILTER** tweets **BY** text **matches** '.\*[!].\*';

user\_extracted = **FOREACH** exclamatory\_tweets

**GENERATE** created\_at,

text,

user#'name' **AS** name:**chararray**,

user#'screen\_name' **AS** screen\_name:**chararray**;

Storing JSON data with JsonStorage

You can store the result of a Pig script in JSON format using JsonStorage. It will store each tuple from a Pig relation as a JSON object on a single line. The schema of the relation to be stored must be fully specified.

Example:

*-- Pig 0.9*

**STORE** result **INTO** 's3n://my-s3-bucket/path/to/json/output'

**USING** org.apache.pig.piggybank.storage.JsonStorage();

*-- Pig 0.12*

**STORE** result **INTO** 's3n://my-s3-bucket/path/to/json/output'

**USING** org.apache.pig.builtin.JsonStorage();

Mortar Project Example

For a full example in a Mortar Project, clone down the [mortar-examples](https://github.com/mortardata/mortar-examples) repository and check out the [coffee\_tweets](https://github.com/mortardata/mortar-examples/blob/master/pigscripts/coffee_tweets.pig) pigscript.

FromJSON UDFs

If you have JSON data stored within a field of a larger schema (as in the case where a tsv log file might have JSON fields) there are two UDFs that can help extract that data.

FromJsonWithSchema

**DEFINE** FromJson org.apache.pig.piggybank.evaluation.FromJsonWithSchema('f1: int, f2: chararray');

data = **LOAD** 'my\_data' **USING** TextLoader **AS** (text: **chararray**);

json = **FOREACH** data **GENERATE** **FLATTEN**(FromJson**(**text));

The schema of the json relation will be (f1:int, f2: chararray)

FromJsonInferSchema

data = **LOAD** 'my\_data' **USING** TextLoader **AS** (text: **chararray**);

json = **FOREACH** data **GENERATE** org.apache.pig.piggybank.evaluation.FromJsonInferSchema(text);

The schema of the json relation will be (object: map[]), where f1 and f2 are keys in the map.

JsonLoader Schema Guidelines

A few additional guidelines for providing a schema to JsonLoader:

* Fields in your data may be in a different order than fields in your schema, so long as the field names match.
* JsonLoader will perform type coercion if it is possible (for example, converting "17" to 17 for an int field).
* If you specify an object field as my\_field: map[], that field will be loaded in the same way as the document would have if you had not passed JsonLoader a schema. This can be useful for variable schema elements, for example theoptions field in: {req\_param\_1, req\_param\_2, options: {}}.
* To load a field with an initial underscore (eg: "\_id"), use the original field name in the schema. When it loads, the field will have the prefix "underscore" because Pig does not allow names with prefixing underscores.

e.g. 'f1: int, \_f2: chararray' would load the fields (f1, underscore\_f2)

* To load a field with spaces in its name, use backslashes to escape the original field name in the schema. When it loads, the spaces will be replaced by underscores.

e.g. 'f1: int, f\\ 2: chararray' would load the fields (f1, f\_2)

* Pig requires every value in a bag to be part of a tuple, while JSON allows arrays to hold values directly.
  + To load the flat array "arr": [1, 2, 3, 4], you would specify the schema 'arr: {t: (i: int)}'.

It would be loaded as {(1), (2), (3), (4)}. Note that the name "t" of the inner tuple does not matter.

* + If you knew that the flat array would have only four elements, you could cast it to a tuple,

e.g. 'arr: (i: int, j: int, k: int, l: int)'' would load (1, 2, 3, 4).

* + Nested arrays can be loaded too. You could load "arr": [[1, 2], [3, 4]]

with the schema 'arr: {t: (arr: {t: (i: int)})}', producing {({(1), (2)}), ({(3), (4)})}.

* + Finally, you can combine nested array loading and casting arrays to tuples. You could load a JSON array-of-arrays of latitude-longitude coordinate pairs,

"coord\_arr": [[40.664167, -73.938611], [37.783333, -122.416667]],

using the schema 'coord\_arr: {t: (coords: (lat: double, long: double))}'.

JsonLoader Schema Example

Let's say you received the following objects as responses to calling some API.

**{** **"response":** **{** **"id":** **10123,** **"thread":** "Sloths"**,** **"comments":** **[**"Sloths are adorable"**,** "So chill"**]** **},** **"response\_time":** **0.425** **}**

**{** **"response":** **{** **"id":** **13828,** **"thread":** "Bigfoot"**,** **"comments":** **[]** **}** **,** **"response\_time":** **0.517** **}**

You could load these without a schema using:

data = **LOAD** 's3n://my-s3-bucket/path/to/responses'

**USING** org.apache.pig.piggybank.storage.JsonLoader();

responses = **FOREACH** data **GENERATE** object#'response' **AS** response: **map**[];

out = **FOREACH** responses

**GENERATE** response#'id' **AS** id: **int**, response#'thread' **AS** thread: **chararray**,

response#'comments' **AS** comments: {t: (comment: **chararray**)};

**STORE** out **INTO** 's3n://path/to/output' **USING** PigStorage('|');

If you run this script, you will get:

10123|Sloths|{(Sloths are adorable),(So chill)}

13828|Bigfoot|{}

Suppose you had a third row though:

**{** **"response":** **{** **"id":** **10123,** **"thread":** "Sloths"**,** **"comments":** **[**"Sloths are adorable"**,** "So chill"**]** **},** **"response\_time":** **0.425** **}**

**{** **"response":** **{** **"id":** **13828,** **"thread":** "Bigfoot"**,** **"comments":** **[]** **}** **,** **"response\_time":** **0.517** **}**

**{** **"response":** "Could not find any results for query \"Nessy\""**,** **"response\_time":** **0.788** **}**

You could not cast the string value for "response" in the last object to a map as you did before.

To handle this properly (i.e. write a null for that field instead of crashing), you need to specify a schema.

data = **LOAD** 's3://my-s3-bucket/path/to/responses'

**USING** org.apache.pig.piggybank.storage.JsonLoader('

response: (id: **int**, thread: **chararray**, comments: {t: (comment: **chararray**)}),

response\_time: **double**

');

out = **FOREACH** data **GENERATE**

**FLATTEN**(response) **AS** (id, thread, comments),

response\_time;

**STORE** data **INTO** 's3n://my-s3-bucket/path/to/output' **USING** PigStorage('|');

If you run this script, you will get:

10123|Sloths|{(Sloths are adorable),(So chill)}|0.425

13828|Bigfoot|{}|0.517

|||0.788

Note that we did not have to specify types for the fields in out, since we already specified them in the load statement.

JsonStorage Performance Tips

JsonStorage uses a dynamic buffer as part of the storage process. If you know that each row of your data will never exceed a certain size and need a performance boost, you can tell JsonStorage to use a fixed-size buffer. You specify this size in kilobytes as a string parameter which must parse to an integer.

Example:

*-- Pig 0.9*

**STORE** result **INTO** '$OUTPUT\_PATH' **USING** org.apache.pig.piggybank.storage.JsonStorage('16');'

*-- Pig 0.12*

**STORE** result **INTO** '$OUTPUT\_PATH' **USING** org.apache.pig.builtin.JsonStorage('16');'

Database Store Statement

Microsoft SQL Server

If you are using SQL Server as your JDBC Platform, use the following template STORE statement:

**%default** DATABASE\_HOST 'dbIdentifier.XXXXXXXXXXX.your-time-zone.rds.amazonaws.com:<port number>';

**%default** DATABASE\_NAME 'MY\_DATABASE';

**%default** DATABASE\_USER 'MY\_USER';

**%default** DATABASE\_PASS 'MY\_PASS';

**%default** DATABASE\_DRIVER 'com.microsoft.sqlserver.jdbc.SQLServerDriver';

**%default** DATABASE\_TYPE 'sqlserver';

**STORE** my\_result **INTO** 'hdfs:///unused-ignore'

**USING** org.apache.pig.piggybank.storage.DBStorage('$DATABASE\_DRIVER',

'jdbc:$DATABASE\_TYPE://$DATABASE\_HOST',

'$DATABASE\_USER',

'$DATABASE\_PASS',

'USE $DATABASE\_NAME INSERT INTO my\_table(my\_col\_1,my\_col\_2,my\_col\_3) VALUES (?,?,?)');

General JDBC

To store data into any other JDBC complient database, use the following template STORE statement:

**%default** DATABASE\_HOST 'ec2-XX-XXX-XXX.compute-1.amazonaws.com';

**%default** DATABASE\_NAME 'MY\_DATABASE';

**%default** DATABASE\_USER 'MY\_USER';

**%default** DATABASE\_PASS 'MY\_PASS';

**%default** DATABASE\_DRIVER 'org.postgresql.Driver';

**%default** DATABASE\_TYPE 'postgresql';

**STORE** my\_result **INTO** 'hdfs:///unused-ignore'

**USING** org.apache.pig.piggybank.storage.DBStorage('$DATABASE\_DRIVER',

'jdbc:$DATABASE\_TYPE://$DATABASE\_HOST/$DATABASE\_NAME',

'$DATABASE\_USER',

'$DATABASE\_PASS',

'INSERT INTO my\_table(my\_col\_1,my\_col\_2,my\_col\_3) VALUES (?,?,?)');

You'll need to edit the %default parameters to connect to your specific database. Additionally, DATABASE\_DRIVER and DATABASE\_TYPE need to specify the JDBC driver name and database vendor. The example above illustrates usage of a Postgres database driver. See the table below for possible values or consult your vendor's JDBC driver documentation. Finally, change the INSERT statement to reflect your target database table and columns. Make sure that the number of fields in the alias you store corresponds to the number of fields in your insert statement.

Also, you'll need to ensure that the table exists before running the insert.

To connect directly to your database server from Mortar there are a few additional details you should know.

* Your database server must be running in EC2
* If you have trouble connecting to your server, you may need to email[support@mortardata.com](mailto:support@mortardata.com) so that we can unfirewall that port on your account's Mortar clusters.

JDBC Dependencies

Connecting to a JDBC database requires a driver specific to each RDBMS vender or project. This typically is found in a Java jar file. Due to licensing restrictions, some JDBC drivers can not be distributed by Mortar. However, in many cases these drivers can be freely downloaded from the internet and supplied with your Mortar Project by placing the necessary jar in your project's lib/ directory. The table below describing several common database engines notes whether or not you must supply a JDBC driver with your project.

Common JDBC Platforms

| **Platform** | **Driver Name** | **Example JDBC Conn String** | **Required Driver** |
| --- | --- | --- | --- |
| PostgreSQL | org.postgresql.Driver | jdbc:postgresql://$dbhost/$dbname | No |
| MySQL | com.mysql.jdbc.Driver | jdbc:mysql://$dbhost/$dbname | [Yes](http://repo1.maven.org/maven2/mysql/mysql-connector-java/5.1.29/mysql-connector-java-5.1.29.jar) |
| MSSQL Server | com.microsoft.sqlserver.jdbc.SQLServerDriver | jdbc:mysql://$dbhost:$port | [Yes](http://www.microsoft.com/en-us/download/details.aspx?displaylang=en&id=11774) |
| Greenplum | org.postgresql.Driver | jdbc:postgresql://$dbhost/$dbname | No |

Note about downloading the Microsoft SQL Server driver: You will need to download one of the .tar files. Once expanded, in the enu folder you will find the jar files. Select the jar file which suits your SQL query tool and copy it to your project's lib/directory.

Transactions

The underlying store function, [DBStorage](https://github.com/apache/pig/blob/trunk/contrib/piggybank/java/src/main/java/org/apache/pig/piggybank/storage/DBStorage.java), does insertions within a transaction. It will send batches of 100 rows at a time to the database, but will only commit the transaction when the Hadoop task finishes successfully. This guarantees that you will not see duplicate rows inserted.

Note that there is one transaction per Map or Reduce task, not per MapReduce job. So if the MapReduce Job fails, there may be partial data stored in the target database table from tasks that succeeded.

Controlling Database Load

One thing to be careful about is load on your PostgreSQL database. Each Map or Reduce task will open a connection to the database. With large amounts of data (and hence number of tasks), this can cause significant load on your database.

To limit the number of open connections, you can trigger a reduce phase before running the STORE statement and specify the number of reducers that run. Code for that would look something like:

*-- maximum number of database connections to allow*

**%default** MAX\_NUM\_DATABASE\_CONNECTIONS **20**;

*-- trigger a reduce phase with a specified number*

*-- of reducers to write to the database*

my\_input\_data = ...;

throttle\_group\_by = **GROUP** my\_input\_data

**BY** some\_field\_with\_high\_cardinality

**PARALLEL** $MAX\_NUM\_DATABASE\_CONNECTIONS;

throttle\_result = **FOREACH** throttle\_group\_by **GENERATE** **FLATTEN**(my\_input\_data);

*-- store the data out with DBStorage as above*

**STORE** throttle\_result **INTO** ...;

Loading XML Data with StreamingXMLLoader

Precursor

To load XML data into Mortar, first ensure that your files are formatted with exactly one XML object per line. This helps Pig and Hadoop to properly split your files for parallel processing.

StreamingXMLLoader takes two string parameters: the record identifier, which indicates the chunks of XML which will correspond to individual Pig records, and the (optional) tag list, which indicates which particular tags should be extracted from each record.

Quick Start - No Tag Extraction

The simplest way to use the StreamingXMLLoader is with no tag extraction, providing only a record identifier:

XML\_objects = **LOAD** 's3n://my-s3-bucket/path/to/XML/file'

**USING** org.apache.pig.piggybank.storage.StreamingXMLLoader('Entry');

This loads each XML document (delimited by 'Entry') into an individual tuple. The entire document, including the delimiting tags, will appear in the record. Each record will be a single chararray, so make sure any schema you include reflects this.

This will give you a general idea about your data, but you'll be stuck with a single giant field full of unparsed XML.

Tag Selection

Once you know which fields you need from your data, you can provide StreamingXMLLoader with a list of tags. You pass the list to StreamingXMLLoader as a second (optional) string parameter, like this:

XML\_objects = **LOAD** 's3n://my-s3-bucket/path/to/XML/file'

**USING** org.apache.pig.piggybank.storage.StreamingXMLLoader(

'Entry',

'tag1, tag2, tag3'

) **AS** (

tag1: {(attr: **map**[], content:**chararray**)},

tag2: {(attr: **map**[], content:**chararray**)},

tag3: {(attr: **map**[], content:**chararray**)},

);

It's also possible (and an extremely good idea) to add a schema in the AS clause of your LOAD statement. The records that come back are somewhat complex but structurally uniform (which is more than you can say about the average XML document), so this will benefit you in the long run.

StreamingXMLLoader will extract these fields from your data, filling the attr map with attribute name/value pairs. The raw contents of each element (not including its enclosing start and end tags) appear as a chararray.

Simple Example

Let's read in a slice of the [Protein Sequence Database](http://www.cs.washington.edu/research/xmldatasets/www/repository.html#pir). To load the protein entries without doing any element extraction:

proteins = **LOAD** 's3://mortar-example-data/protein-sequence/Protein-Sequence-DB-head.xml'

**USING** org.apache.pig.piggybank.storage.StreamingXMLLoader('ProteinEntry');

However, let's provide a tag list so we can go after specific elements:

proteins = **LOAD** 's3://mortar-example-data/protein-sequence/Protein-Sequence-DB-head.xml'

**USING** org.apache.pig.piggybank.storage.StreamingXMLLoader(

'ProteinEntry',

'protein, organism, source'

) **AS** (

protein: {(attr:**map**[], content:**chararray**)},

organism: {(attr:**map**[], content:**chararray**)},

source: {(attr:**map**[], content:**chararray**)}

);

We can now use the fields in the script as we please, say to find ones that came from a human source:

proteins = **LOAD** 's3://mortar-example-data/protein-sequence/Protein-Sequence-DB-head.xml'

**USING** org.apache.pig.piggybank.storage.StreamingXMLLoader(

'ProteinEntry',

'protein, organism, source'

) **AS** (

protein: {(attr:**map**[], content:**chararray**)},

organism: {(attr:**map**[], content:**chararray**)},

source: {(attr:**map**[], content:**chararray**)}

);

-- We know there's only one protein, organism and source element in each entry -- so we can get away with flattening like this. You might need to be more clever -- in real life, though.

proteins = **FOREACH** proteins **GENERATE**

**FLATTEN**(protein.content) **AS** protein,

**FLATTEN**(organism.content) **AS** organism,

**FLATTEN**(source.content) **AS** source;

human\_proteins = **FILTER** proteins **BY** source == 'human';

If you take a peek at that protein sequence data, you might notice that sourceelements are nested inside organism elements. By design,StreamingXMLLoader allows you to extract arbitrarily nested elements from your XML. However, a nested field that is explicitly extracted will *not* appear in the content chararray of the enclosing field.