# XTL: Extensible Extract-Transform-Load Framework

Jivan Gubbi jcgubbi@ucla.edu
Shikha Mody smody@ucla.edu
Bradley Mont bradleymont@gmail.com
Brendon Ng brendonn8@ucla.edu

CS 214: Big Data Systems
*UCLA, Fall 2021*

## 1. Introduction

XTL is an extensible Extract-Transform-Load (ETL) framework to streamline transfers between different big data systems. XTL was designed with the goal of improving the ease of use of the various distributed data storage systems available. The tool provides users with an extremely simple JSON configuration-based interface that abstracts the coding and engineering work required in dealing with the different APIs for each separate data system.

XTL provides a modular and extensible framework for common data pipeline tasks. Many applications and use cases require the movement of data from one location to another, often involving different data storage systems and/or a transformation on the data in between. XTL consists of plug-and-play importer and exporter building blocks connecting to a user-configured transformation map-reduce script. Importers and exporters are code-once-use-again modules that are standardized to pass the inbound or outbound data to and the transformation module respectively. Once a connector is coded, it can be reused and used interchangeably with other connectors. XTL users simply change a configuration JSON file to specify which connector to be used each for importing and exporting, the connector-specific parameters, and the map-reduce transformation scripts to run in the middle.

With XTL, the work of coding data transfers and pipelines is reduced to a few lines in a configuration JSON file, cutting down on development time and effort.

## 2. Motivation

XTL is meant to minimize the client-facing programming needed to transfer and process data between big data systems. It seeks to make the user experience as configuration-based as possible rather than implementation-based. This way, XTL provides an extensible and modular system to easily fit users' needs.

There is no "one-size-fits-all" solution that allows a single unified interface to successfully and effectively address existing (along with future) big data problems. As a result, many systems have different interfaces, APIs, and supported functionality. Interacting with these different systems leads to interface-specific code which is difficult to reuse and adapt for different situations. XTL seeks to abstract out those details to make the experience of working with different big data systems a more streamlined and modular experience. By maintaining an "extract-transform-load" structure, XTL addresses a broad category of big data problems that require data that comes from a certain source to be processed (in a customized way), and loaded for the next step.

In order to load data from one source to another, the traditional approach could be: 1) manually transfer, or 2) utilize another coding

language's APIs to transform data into a format compatible with the outgoing system. Manually extracting, then processing, then loading data takes a lot of time, and is not easily scalable for repetitive tasks. Utilizing common APIs to, say, transfer a file from Hive to HDFS is more scalable but not easily reusable for modifications to the pipeline. As a result, XTL addresses these concerns with a configuration-driven approach to ETL tasks.

## 3. Technical Design

At a high level, XTL is separated into three overarching components. There are inbound connectors, the middle layer, and outbound connectors. Our design choices were made with modularity and extensibility in mind. Execution happens in three non overlapping stages, one for each layer. This separation allows us to interchange connectors and change transformations without affecting other layers. It also makes writing new connectors simple since there are no interactions between layers beyond data passed through our shared HDFS working directory.

From a user's perspective, we opted for a low code platform where users specify which connectors and transformations they are using through a simple JSON configuration file.

The inbound connector is responsible for loading data from a source into our working directory of HDFS. The implementation and specifics of the connector depends on the data source it is pulling from. Connectors read from the JSON file to get information specific to their data source, such as a table name or cluster location.

The transformation layer runs as a MapReduce job on the data in our working directory. Users are able to specify a map function and reduce function by passing any arbitrary executables through the configuration file. While there are clear limitations to using the MapReduce framework, given the scope and timeline of this project we chose to take this simple approach to show the viability of our platform. In general, this layer can be expanded to include a more complex execution engine and multiple transformations.

After the transformation has run, the outbound connector must move the output of the transformation to the chosen data sink. This is very similar to the inbound connector but also has some added complexity for structured data sinks. In this case, we must not only specify the location of the data sink but also some schema for table creation for our data within the configuration file.

## 4. Implementation

XTL was mostly implemented in Java. This way, the inheritance and object-oriented style could be leveraged for a modular design. Java also supports libraries for platform-specific APIs that were utilized for each connector. To assist with dependency management, we utilized Maven.

Each connector (both inbound and outbound) were extensions of a general `Connector` class. This class defined the interfaces for all specific connectors to implement: a constructor, `parseJSON` (function to parse configuration JSON for that connector), and an `execute` method to be called if that connector is being used. This streamlines the process for extending supported platforms, because developers can create a new class for the new connector simply by extending the existing `Connector` class. Each specific connector that we implemented – Hive, Spark, HDFS, and local file – has its own implementations of the `Connector` class with these functions. Furthermore, the separation of concerns between connectors allowed the platform-specific APIs to be contained within that connector's file. This emphasizes reusability and a "plugin" style for XTL.

The `parseJSON` method common to all connectors is the step where the configuration

file's information gets aligned with the actual data system in use. This is a platform-specific component that extracts key information from the configuration file in order to perform the required action. This method cannot necessarily be unified across all connectors because different platforms require different information. Although platform-specific functionality is inevitable for any extract or load mechanism, our design gives flexibility to the user to define how information gets provided and parsed by XTL.

For the `execute` methods, the inbound and outbound connectors had slightly different purposes. Inbound connectors would extract data from the source, and save it in files in XTL's HDFS within a directory designated as `inbound`. Thus, the transformation step had a singular place to start its transformation. The transformation step also extends the `Connector` class (connects from inbound to outbound directories), and was implemented as a MapReduce task to leverage the distributed nature of the platform. The mapper and reducer functions are user-provided, although not required. Therefore, XTL can also serve as an intermediary to simply extract and load data from one system to another. After the MapReduce transformation is completed, it saves the results as files in an `outbound` directory of XTL's HDFS. Once again, this makes the interfaces consistent across any of the connectors. Outbound connectors' `execute` methods load the transformed data from the outbound directory to the outgoing data sink, utilizing platform-specific APIs once again. By defining the interfaces and formats of the intermediate steps, we were able to streamline how each connector can "plug in" to the entire system.

In terms of developer-friendly functionality, connecting Maven as a dependency management platform allowed for unified dependency imports. Although there were challenges with the compatibility of different dependencies with each other, Maven allowed us to have a consolidated place for dependencies and framework for XTL as a whole. It also was conducive to a Makefile which streamlined the compilation and build processes, emphasizing efficient development and testing.

## 5. Evaluation

First, we will evaluate XTL in terms of its ease-of-use, as it is a user-facing system designed to abstract away a lot of data processing implementation details from the user.

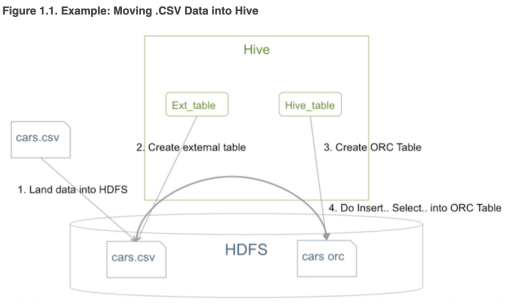


Figure 1: Traditional HDFS -> Hive Workload

We will examine the programming workload necessary to compute the following data processing task without and then with XTL: transferring CSV data from HDFS into Hive. Without XTL, the programmer must load the CSV data into HDFS, utilize SQL to create an external table for the data, utilize SQL again to create a Hive-managed table, and then execute the transfer of data between these tables. Figure 1 gives a pictorial representation of this workload.

```
{
    "importer": {
        "platform": "HDFS",
        "address": "127.0.0.1",
        "port": "8020",
        "inputFilepath": "/input/SparkSQLOutputData.csv"
    },
    "exporter": {
        "platform": "HIVE",
        "address": "localhost",
        "port": "10000",
        "tableName": "HIVE_OUTPUT",
        "tableColumns": "(COLUMN1 STRING, COLUMN2 STRING)",
        "dockerContainerID": "035f7e57561b"
    }
}
```

Figure 2: XTL config.json for HDFS -> Hive

Using XTL, the workload of the programmer is significantly less. As seen in Figure 2, the user simply needs to create a configuration file written in JSON that specifies the details of HDFS as its input and Hive as its output, and it accomplishes the exact same task seen in Figure 1. This example workload demonstrates the power of XTL is easing the workload of the developer when performing big data operations.

Due to time constraints, we were not able to perform any tests on the scalability of XTL. However, with XTL's utilization of files in its middle layer, we expect that this work can be easily distributed among multiple machines, which will lead to excellent scalability.

Additionally, we performed several end-to-end tests on workloads such as Hive to HDFS, HDFS to Spark, Spark to local file, etc. All these workloads were successfully completed, which attests to the accuracy and robustness of XTL.

## 6. Extension

We have several goals for the future of XTL. First off, we intend on supporting as many data platforms as possible. Ideally we would like to achieve broad coverage of the most popular frameworks. With the extensible design of XTL, the process of adding new frameworks is quite simple: we need to create inbound and outbound connectors that extend our Java `Connector` class, and then specify any necessary information in a JSON configuration file.

Additionally, we hope to support distributed execution of XTL in order to maximize performance and scalability. Our use of files in the middle layer should work naturally for distributed execution.

In terms of features, we intend on adding data visualization and debugging tools as well, in order to make the system as well-rounded as possible. Lastly, as most data processing jobs involve several stages, we hope to modify XTL to support a chain of layers, opposed to our current implementation with a single transformation layer.

## 7. Conclusion

Overall, XTL provides a simple interface that is easily extensible for users to add and/or modify existing connectors for easy transfers between big data storage and processing systems. XTL was successful in reducing the level of effort in programming data transfers and pipelines. Connectors were developed for four types of data storage in Hive, Spark, HDFS, and local files. The usage of these developed connectors has proven to be flexible and easy to interchange. The framework is structured in an extensible manner and allows for easy creation of additional connectors for more data sources.

XTL shows the potential to be developed into a useful product for teams that deal with data in large or small amounts. It provides users the option to easily perform simple transfers without having to hire developers or requiring team members to learn different APIs. XTL will be beneficial for teams looking to spend less time on data engineering and more time on their product development or research. As the amount of data storage systems will continue to grow, XTL's extensible nature will ensure the product can keep up with the growing field, continuing to cut development costs and effort into the future.

Figure 1: https://docs.cloudera.com/HDPDocuments/HDP2/HDP-2.4.3/bk_dataintegration/content/moving_data_from_hdfs_to_hive_external_table_method.html

## 8. Teammate Contributions

Shikha completed the inbound and outbound Spark connectors (SparkSQL for inbound, and Spark for outbound). She also structured the JSON parsing to support reading the configuration file, refactored the initial codebase to integrate with Maven to assist with dependency management, and tested end-to-end XTL functionality.

Jivan completed the inbound and outbound HDFS connectors as well as the inbound and outbound local file connectors. He also helped work on middle layer file storage and created general use HDFS utility functions used by other connectors.

Bradley completed all aspects of the project involving Hive, including the inbound and outbound Hive connectors. He designed the structure of the JSON configuration files for Hive, and worked on creating a Docker container with Hive installed for use in the project as well.

Brendon implemented the central layer of the XTL framework including the HDFS core that the system runs on and the transformation configuration and implementation. He also developed the integration of the connectors to the middle layer and standardized the object oriented class structure of the connectors. Brendon also dealt with much of the DevOps and debugging tasks in the project including compilation, dependency issues, and code reviews.

## 9. Sources

Repository link:
https://github.com/brendon-ng/XTL