# Data Ingestion - Project Requirements

## Overview

With the increase in the volume of data, both structured and unstructured, being generated in recent years, data ingestion has become a task of particular importance. This project deals with the task of designing a generic data ingestion system. This system will contain a novel JSON schema that can be generated without understanding the complexities of database systems, but still maintain enough context to allow for a near one-one mapping to a DB schema. The input data can vary in size, type, and use cases, and the system is expected to be able to parse excel sheets, CSVs, and other data formats, which may be stored in a hierarchical, segmented, or single-file fashion.

## System Requirements

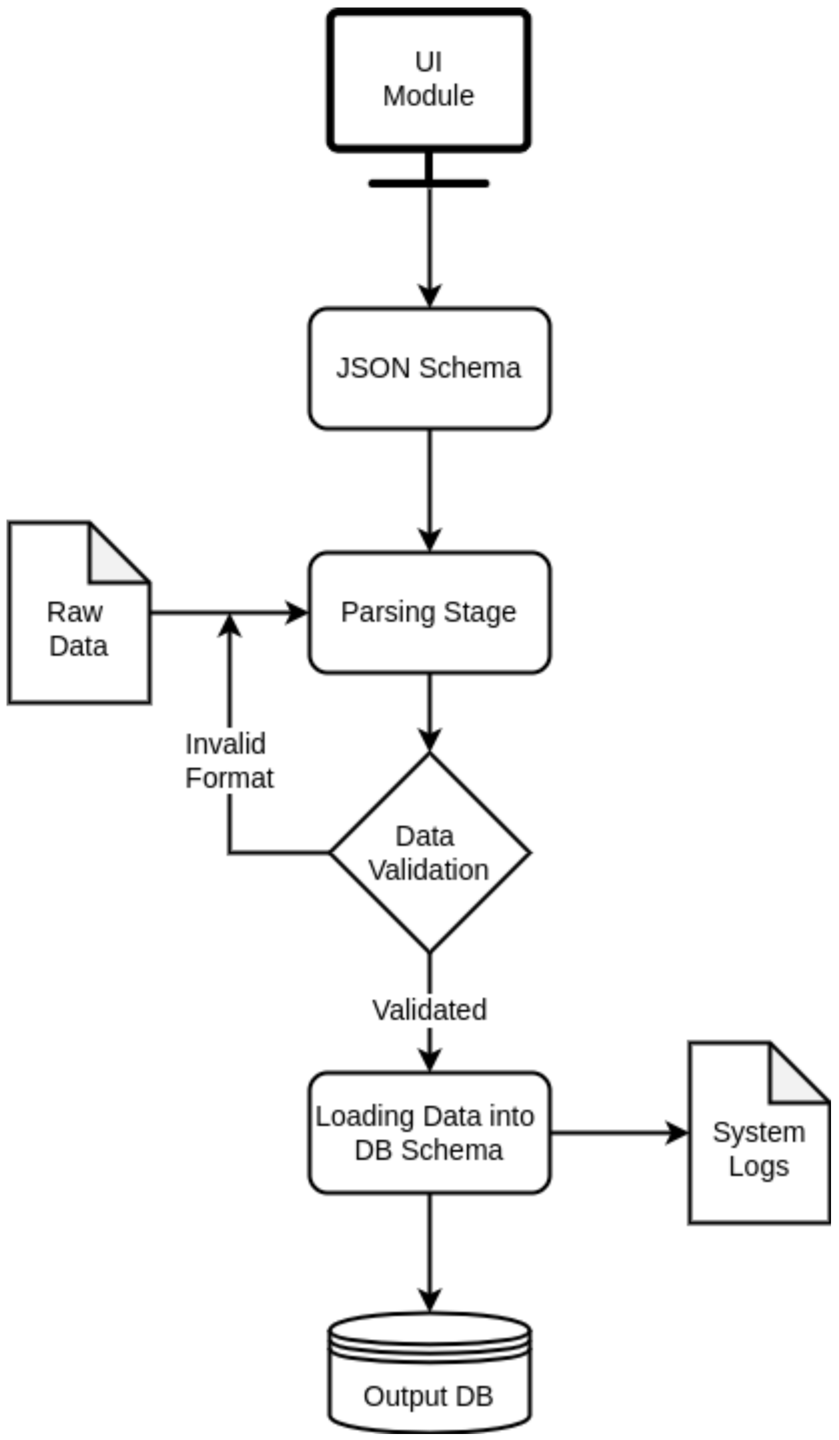
The project will mostly be based on python3. The parsing, validation, loading, and dumping of the data will be handled with dedicated python objects or modules.

The UI section of the project will be based on javascript, in particular, flowJS, which enables us to create a suitable reactJS web application allowing client-side users to easily generate their JSON mock-ups.

Some pre-processing of given raw data, and validation of final output DB will involve MySQL or NoSQL, as per data requirements.

## Project Deliverables

The project can be divided into different phases, with each phase having its own workflow and independent functionality. The different phases should be designed such that it is modular, scalable, and optimized. The stages of development for the project, in order of operations, are:



System Pipeline and Operational Flow

- **JSON Schema:**
  The first task for the project is conceptualizing an efficient JSON schema, that is able to map enough context from the provided raw data, but with around 30-50% of the description contained in a DB schema. The JSON format here will have to work

for most generic data systems, taking into account databases with multiple fields, constraints, varying relationships among fields and tables, and also logging of any occasional syntax or field errors.
The schema should contain sufficient metadata in order to provide suitable scope for the loading phase, specifying the number of tables, rows, and overall structure. In addition to this, there should be an abstracted representation of tables in JSON format. Abstractions can be made with regard to the data type of a field (this can be learnt while validating), constraints, and relationship mapping.

- **Parsing and Validation:**
  The next phase of the project involves parsing the provided raw data with respect to the given JSON format, along with basic data and system validations. This stage is necessary for monitoring any top-level faults that may occur in the generic ingestion system and prevent loading from being completed. In the parsing phase, the data file(s) are loaded, varying as per their storage mechanism, and the files are cross-checked with the JSON schema for any faults.
  In this validation phase, if there are any critical breaking errors present, such as the absence of a table, relation, or key/field mismatch, the appropriate error is presented, along with potential fixes to the raw data or JSON format. The last step of this process involves creating a final DB schema from the validated JSON format.

- **Loading Phase:**
  The loading phase of the project involves generating a data loader, that is able to sequentially load the entire raw data, and adapt it to the generated DB schema. This data is simultaneously outputted to the final DB, and logs for this process are maintained. Upon completion, a summary of this process, containing the final schema, execution logs, and other details, will be generated.
  The key factors to consider when developing the loading phase are efficiency & speed of code execution, fault tolerance, and process checkpoints. With this phase, an end-to-end prototype of the system would be completed, wherein the user is able to obtain a final DB from their raw data and JSON schema.

- **UI Module:**
  This phase of the project is designed to enhance the user's frontend experience, and make the process of generating the JSON schema for their data more intuitive. Instead of a code-based system which might have a technical overhead, the UI

provides a drag-and-drop-based application, which can allow users to better visualize their data, its fields, and any relations among them.

## JSON System Schema

Descriptions of the JSON for the file

| **Field** | **Description** | End-Used Inputted |
|---|---|---|
| database_name | The name of the database | Yes |
| table_num | Numbers of tables present in the string | No |
| enumerations | User-defined enumeration data formats | Yes |
| sets | User-defined set data formats | Yes |
| tables | A JSON-Array containing the descriptions of the table | Yes |
| relations | A JSON-Array containing the descriptions of the relations that exist between the table | Yes |

```
{
  "database_name" : <String>,
  "table_num" : <Integer>,
  "enumerations" : [
              {
                "name" : <String>,
                "fields" : [(<String>, <Int>), ... ]
              }, ...
            ],
  "sets" : [
          {
            "name" : <String>,
            "values" : [<String>, ... ],
          }, ...
        ],
  "tables" : [
            {
            "link_to_file": <String>,
            "table_name" : <String>,
            "attribute_num" : <Integer>,
            "attributes" : [
                          {
                            "attribute_name" : <String>,
                            "attribute_type" : <String>,
                            "is_unique_identifier" : <Boolean>
                          }, ...
                        ],
            "constraints" : {
                          "not_null" : [<String>, ...],
                          "unique" : [<String>, ...],
                          "primary_key" : [<String>, ...],
                          "candidate_key": [[<String>,...],...], //change need to be made
                          "foreign_key" : [[<String>, ...],...],
                          "default" : [{
                                        "attribute_name": <String>,
                                        "value": <Integer>
                                        }, ...
                                      ],
                          "check" : [{
                                    "attribute_name": [<String>, ...],
                                    "condition": <String>
                                    }, ...
                                  ],
                      }
            }, ...
          ],
"relations" : [
              {
                "table_1" : <String>,
                "table_2" : <String>,
                "type" : <Integer>, // 1 of 2 types: one-one/one-many
                "table_1_fkey" : [<String>,...],
                "table_2_fkey" : [<String>,...],
              }, ...
            ],
}
```

Descriptions of the JSON of enumerations

| **Field** | Description | End-User Inputted |
|---|---|---|
| name | Name of the enumeration | Yes |
| fields | JSON array containing enumeration list | Yes |

Descriptions of the JSON of sets

| **Field** | Description | End-User Inputted |
|---|---|---|
| name | Name of the set | Yes |
| values | JSON array containing set | Yes |

Descriptions of the JSON of tables

| **Field** | Description | End-User Inputted |
|---|---|---|
| link_to_file | Path to the original table file | Yes |
| table_name | The name of the table | Yes |
| attributes_num | The number of attributes present in the table | No |
| attributes | JSON-Array containing the descriptions of the attributes of the table | Yes |

Descriptions of the JSON of attributes

| **Field** | **Description** | End-User Inputted |
|---|---|---|
| attribute_name | Name of the attribute | Yes |
| attribute_type | The type of the attribute | Yes |
| is_unique_identifier | Whether the attribute is a unique identifier for the table or not | Yes |

Descriptions of the JSON of constraints

| **Field** | **Description** | End-User Inputted |
|---|---|---|
| not_null | JSON Array containing attribute names that can't be null | Yes |
| unique | JSON Array containing attribute names that have to contain unique values | Yes |
| primary_key | JSON Array containing attribute names that are primary keys for the table | Yes |
| foreign_key | JSON Array containing attribute names that would preserve foreign key relations with other tables | Yes |
| default | JSON Array containing JSON Object, having two parameters, attribute name and corresponding default value | Yes |
| check | JSON Array containing JSON Object, having two parameters, attribute name(s), and corresponding conditions | |

Descriptions of the JSON of relations

| **Field** | **Description** | End-User Inputted |
|---|---|---|
| table_1 | Name of the 1st table involved in the relation | Yes |
| table_2 | Name of the 2nd table involved in the relation | Yes |
| type | The type of relation, this option will be one of the 2 types of relation that can exist and hence would be an integer, with 0 being one-to-one and 1 being one-to-many | Yes |
| table_1_fkey | The foreign key attribute of table 1 | Yes |
| table_2_fkey | The foreign key attribute of table 2 | Yes |

## Supported Data Types

The nature of the data that can be recorded in a database table is represented by data types. For example, if we wish to store string data in a specific column of a table, we must declare a string data type for this column. The data types that can are supported by our framework are as follows:

| **Type** | **Abstraction** | **Description** | **Corresponding SQL Type** |
|---|---|---|---|
| String | STRING | All alphanumeric characters | CHAR, VARCHAR, TEXT, TINYTEXT, MEDIUMTEXT |
| Integers | INT | All integer types of data values | INT, INTEGER, TINYINT, SMALLINT, MEDIUMINT, BIGINT, NUMERIC |
| Floating | FLOAT | All numbers including those with the decimal | FLOAT, REAL, DOUBLE, DECIMAL |
| Boolean | BOOL | True/False Values | BOOL, BOOLEAN |
| Date | TIME | It would be stored as the number of seconds that have passed since UNIX epoch time | DATE, DATETIME, TIMESTAMP, TIME |
| Large Object Datatype | BINARY | Data type to store images, sound files etc by converting them to RAW(binary) format. | BLOB, TINYBLOB, MEDIUMBLOB, LONGBLOB |

| | | | |
|---|---|---|---|
| | | The ingestion system, ideally, does not store files in this format, but would instead divert them to cloud storage and store its URL. | |
| Pseudo-random Value | RAND | Randomized value generated in a given space. In general, this is not desirable to be used for most systems. Can be useful for arbitrary hashing | n/a |
| Unique ID | UID | Automatic ID generator, that follows the provided naming syntax provided by users. Used for identifying primary keys with better context. | n/a |
| Enumeration | ENUM | An enumeration is a user-defined string object that can only display one value from a set of possible values, and contains a mapping of a string entity to a numeric value. An ENUM list can contain up to 65535 values. If a value that is not in the list is entered, a blank value is entered. | ENUM |
| Set | SET | A set is a user-defined object containing 0 or more values from a list of given possible values. A SET list can include up to 64 values. | SET |

## Supported Error Cases

During the validation and loading phases, the program will contain a specific number of error and exit cases and codes. These codes can be used to validate the type of error that is present within the data, as well as the location of it. These changes may or may not be breaking and result in termination of the code in the previous save state.

| Error Code | Error Name | Error Description | Potential Breaking |
|---|---|---|---|

| | | | **Change** |
|---|---|---|---|
| 001 | Connection Error | The tool was unable to establish a connection to the data source. Likely refers to invalid file path | No ingestion possible |
| 002 | Authentication Error | The tool was unable to authenticate with the data source using the set credentials. Likely refers to SQL or file permissions | No ingestion possible |
| 003 | Out of Memory | The tool ran out of memory while ingesting data from the data source. | The tool may not be able to complete ingestion of the data source due to system error. |
| 004 | Network Timeout | The tool was unable to complete ingesting data from the data source within the specified timeout period. Acts as failsafe for large data input | The tool may not be able to complete ingestion of the data source. |
| 005 | Rate Limit Exceeded | The tool has exceeded the rate limit set by the data source. Occurs due to multi-processing parameters. | The tool may not be able to ingest data from the data source until the rate limit resets. |
| 006 | Mismatched Count Error | The number of attributes/tables are mismatched with the super count maintained in JSON | The tool may not be able to ingest data from the data source. |
| 007 | Missing Required Field | The data source is missing a required field that the tool expects to be present. Includes key constraints, required tables/attributes. | The tool may not be able to ingest data from the data source. |
| 008 | Invalid Data Format | The data source contains data that is not in a supported format by the tool. (Refer table for formats) | The tool may not be able to ingest data from the data source. |
| 009 | Invalid Field Value | The data source contains a field with an invalid value that | The tool can ignore error. |

| | | | |
|---|---|---|---|
| | | the tool is unable to process. Occurs when data loader gets value different from validated datatype. | |
| 010 | Duplicate Key | The data source contains a duplicate key that the tool is unable to process. | The tool may ignore extra key. |
| 011 | Circular Dependency Detected | The foreign key relations are detected to be cyclc, and invalid. | The tool may not be able to ingest data from the data source. |
| 012 | Data Inconsistency | The data source contains inconsistent data that the tool. Likely due to invalid CSV format. | The tool may not be able to complete ingestion of data source. |
| 013 | Constraint Inconsistency | Data present in the source does not match with the specified constraints present in JSON | The tool may ignore entry and/or constraint requirements. |
| 014 | Enumeration Error | Invalid enumeration type detected in data. | The tool may ignore and revert to string. |
| 015 | Unknown Error | An unknown error occurred while the tool was ingesting data from the data source. | The tool may not be able to complete ingestion of the data source. |

## Implementation Examples

Trains Dataset Example

Stats Dataset Example

Example 3

## UI Requirements

The UI interface that would be used by the end user to show the database schema would have the following descriptions:

1. The entire database schema would be built using a node-based graph.

    a. The nodes of the graph would represent the tables

    b. The edges of the graph would represent the relations

2. **Nodes:**

    a. Each nodes would have section to input the *name of the fields* present inside the Table, what *type of data* would the Field store (Integers, Decimals, Strings etc. which would be easily understood by a non-technical user)

    b. The table will also ask what field of the table would be the *Unique Identifier* for that table.

3. **Edges:**

    a. The edges would represent the relations that exist between the tables. In the tables we can ask what are the foreign key constraints (In a manner which is understandable by a non-technical user).

    b. The edges would also ask for what kind of relationship exists between the two tables like *one-to-one, many-to-one,* etc.

This approach of using a node-based graph allows a much more intuitive UI and UX for the user and could be built using React Library `ReactFlow.js` Link.

Aakash Jain - aakash.jain@students.iiit.ac.in

Ritvik Aryan Kalra - ritvik.kalra@research.iiit.ac.in

UI Tool Requirements Document