SANDIA REPORT
SAND2023-11430
Printed October 2023

IDB Data Loader

Steven R. Schwartz

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico
87185 and Livermore,
California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@osti.gov
Online ordering:	http://www.osti.gov/scitech

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Rd
Alexandria, VA 22312

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.gov
Online order:	https://classic.ntis.gov/help/order-methods/

National Nuclear Security Administration

ABSTRACT

The International Database of Reference Gamma-Ray Spectra of Various Nuclear Matter is designed to hold curated gamma spectral data and will be hosted by the International Atomic Energy Agency on its public facing web site. Currently, the database to be hosted is given to the International Atomic Energy Agency by Sandia. This document describes the application used by Sandia to load spectral data into a database.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

This page left blank

ACRONYMS AND TERMS

Acronym/Term	Definition
AM	Americium
DDL	Data Definition Language
IDB	International Database of Gamma
JDBC	Java Data Base Connectivity
JRE	Java Runtime Environment
LANL	Los Alamos National Laboratory
LLNL	Lawrence Livermore National Laboratory
MOX	mixed oxide
ORM	Object Relational Mapping
PU	plutonium
RDBMS	Relational Database Management System
SNL	Sandia National Laboratory
U	uranium

1. INTRODUCTION

The International Database of Reference Gamma-Ray Spectra of Various Nuclear Matter (IDB) is designed to hold curated gamma spectral data that is formatted as described in *Preparation of the IDB Spectra, v.3.pdf*, The tables and relationships that make up the IDB are described in the document *IDB_Database Tables_2021.08.31.pdf*. Both documents are included in the documents folder of the IDB Data Loader distribution.

The IDB Data Loader program uploads one set of CSV files formatted as described in *Preparation of the IDB Spectra* into an existing instance of an IDB. The program is distributed both as source and in pre-built binary jar files. This distribution also includes:

- The Data Definition Language (DDL) to create the tables in an empty IDB.
- The DDL to create the constraints on newly created IDB tables.
- Five sets of CSV data in the format described in *Preparation of the IDB Spectra, v.3* that can be used to test if the program is running correctly.
- The document, *Preparation of the IDB Spectra, v.3.pdf*, that describes the format of CSV files the Data Loader program can upload.
- The document, *IDB_Database Tables_2021.08.31.pdf.*, that describes the tables and constraints that make up the IDB.

The rest of this document describes:

- How to install and run the prebuilt jar files. This includes describing the application's prerequisites and how to change the application's configuration files.
- How to use the included DDL to create an empty IDB in a MariaDB RDBMS.
- The logic used by the program to read and upload the data.

2. INSTALL AND RUN PREBUILT JAR FILES

2.1. Application Prerequisites

The IDB Data Loader is a Java desktop application that uses Java Database Connectivity (JDBC) to connect to an IDB running on a MariaDB Relational Data Base Management System (RDBMS.). The computer that is going to run the application must have:

1. A Java Runtime Environment (JRE) installed. The load application was tested using Java 17 and Java 8 but should work with any Java version greater than 8.
2. A connection path to a MariaDB running as a service and structured such that the service accepts JDBC connections coming from the machine running the application. The load application was tested only on Maria 10.5.

2.2. Installing the Load Application

The load application is distributed in a zip archive that contains everything the application needs to run. To install, unzip the archive into a folder. A successful install will have the files and folders shown in Figure 1.

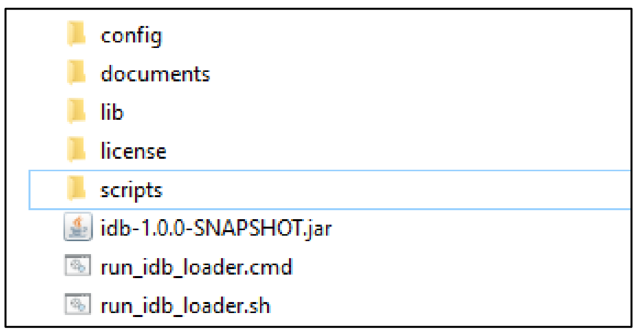


Figure 1. Files and Folders in Top Level of Distribution

2.3. Running the Load Application

On a Windows machine, the load application runs by double-clicking on the *run_idb_loader.cmd* file. The application will start and show its progress in a command window as shown in Figure 2. In Figure 2, the application lists the startup configuration data including the name and location of the individual CSV files that contain the data to be uploaded. The application has stopped and is waiting for the user to enter the password used to connect to the MariaDB database.

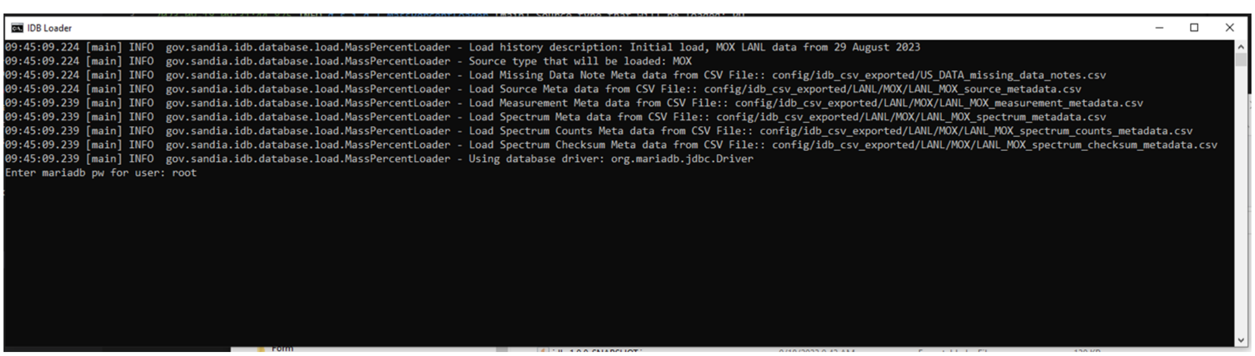
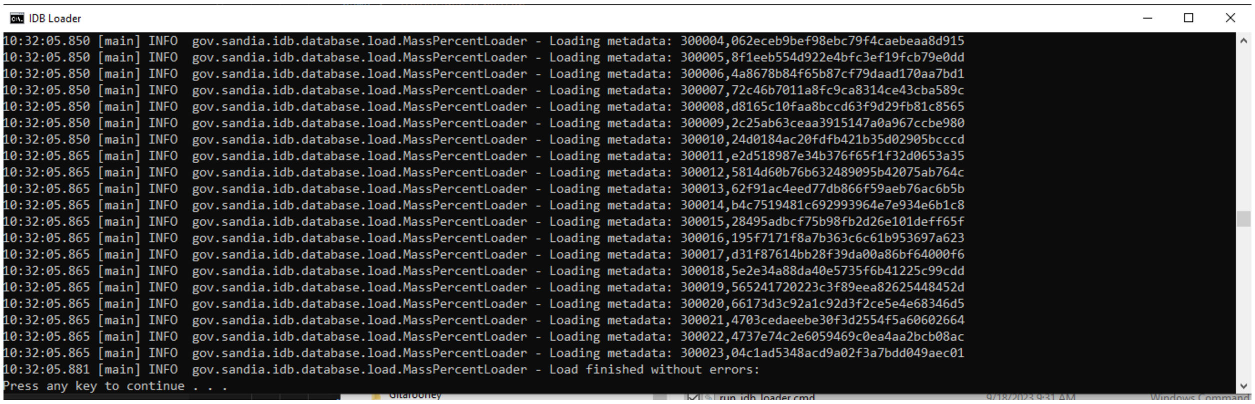


Figure 2. Console Log Output on Startup

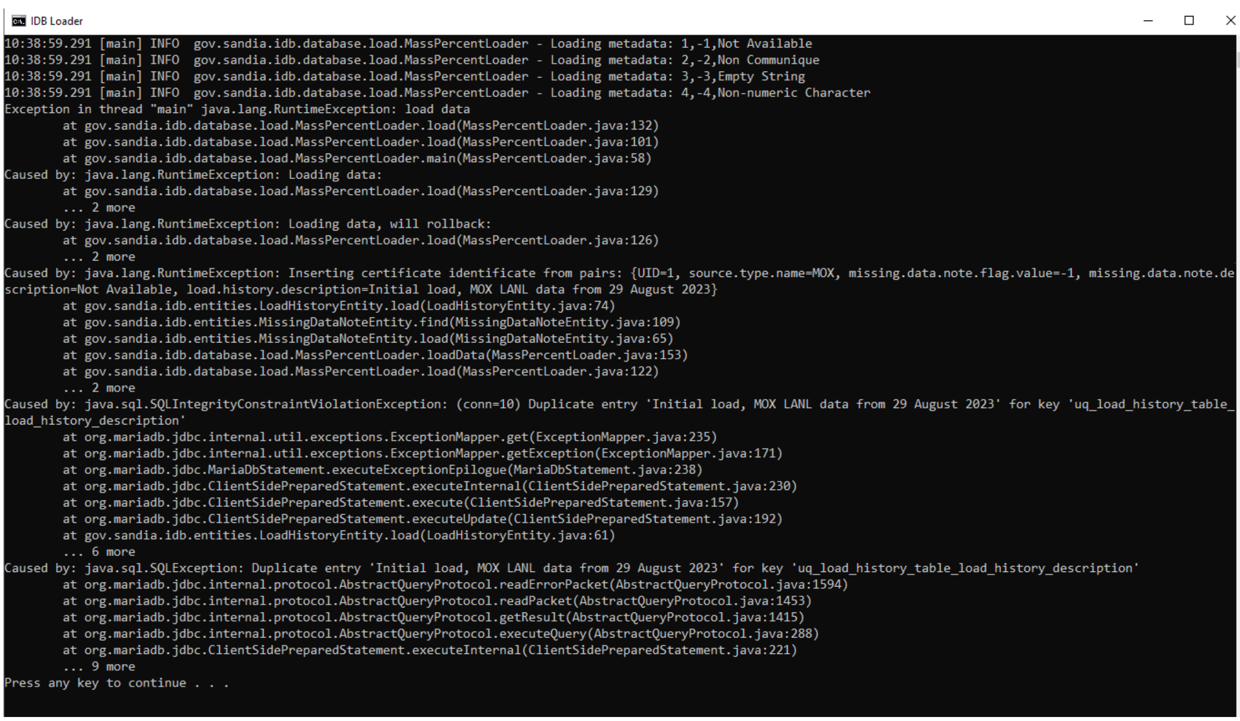
After the user enters the correct password, the application continues running until the load finishes successfully or an exception occurs. If the load succeeds, then the command window will show that the load finished without errors as shown in Figure 3. To close the console, press any key.



```
IDB Loader
10:32:05.850 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300004,062eceb9bef98ebc79f4caebeaa8d915
10:32:05.850 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300005,8f1eeb554d922e4bfc3ef19fcb79e0dd
10:32:05.850 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300006,4a8678b84f65b87cf79daad170aa7bd1
10:32:05.850 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300007,72c46b7011a8fc9ca8314ce43cba589c
10:32:05.850 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300008,d8165c10faa8bccd63f9d29fb81c8565
10:32:05.850 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300009,2c25ab63ceaa3915147a0a967ccbe980
10:32:05.850 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300010,24d0184ac20fdfb421b35d02905bcccd
10:32:05.865 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300011,e2d518987e34b376f65f1f32d0653a35
10:32:05.865 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300012,5814d60b76b632489095b42075ab764c
10:32:05.865 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300013,62f91ac4eed77db866f59aeb76ac6b5b
10:32:05.865 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300014,b4c7519481c692993964e7e934e6b1c8
10:32:05.865 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300015,28495adbcf75b98fb2d26e101deff65f
10:32:05.865 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300016,195f7171f8a7b363c6c61b953697a623
10:32:05.865 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300017,d31f87614bb28f39da00a86bf64000f6
10:32:05.865 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300018,5e2e34a88da40e5735f6b41225c99cdd
10:32:05.865 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300019,565241720223c3f89eea82625448452d
10:32:05.865 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300020,66173d3c92a1c92d3f2ce5e4e68346d5
10:32:05.865 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300021,4703cedaeebe30f3d2554f5a60602664
10:32:05.865 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300022,4737e74c2e6059469c0ea4aa2bcb08ac
10:32:05.865 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 300023,04c1ad5348acd9a02f3a7bdd049aec01
10:32:05.881 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Load finished without errors:
Press any key to continue . . .
```

Figure 3. Console Output Showing a Successful Load

If an error occurs during the load, nothing will be uploaded to the database and the error will be logged to the command window and to the log file. Figure 4 shows the exceptions logged if data being uploaded already exists in the database.



```
IDB Loader
10:38:59.291 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 1,-1,Not Available
10:38:59.291 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 2,-2,Non Communique
10:38:59.291 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 3,-3,Empty String
10:38:59.291 [main] INFO  gov.sandia.idb.database.load.MassPercentLoader - Loading metadata: 4,-4,Non-numeric Character
Exception in thread "main" java.lang.RuntimeException: load data
        at gov.sandia.idb.database.load.MassPercentLoader.load(MassPercentLoader.java:132)
        at gov.sandia.idb.database.load.MassPercentLoader.load(MassPercentLoader.java:101)
        at gov.sandia.idb.database.load.MassPercentLoader.main(MassPercentLoader.java:58)
Caused by: java.lang.RuntimeException: Loading data:
        at gov.sandia.idb.database.load.MassPercentLoader.load(MassPercentLoader.java:129)
        ... 2 more
Caused by: java.lang.RuntimeException: Loading data, will rollback:
        at gov.sandia.idb.database.load.MassPercentLoader.load(MassPercentLoader.java:126)
        ... 2 more
Caused by: java.lang.RuntimeException: Inserting certificate identificate from pairs: {UID=1, source.type.name=MOX, missing.data.note.flag.value=-1, missing.data.note.de
scription=Not Available, load.history.description=Initial load, MOX LANL data from 29 August 2023}
        at gov.sandia.idb.entities.LoadHistoryEntity.load(LoadHistoryEntity.java:74)
        at gov.sandia.idb.entities.MissingDataNoteEntity.find(MissingDataNoteEntity.java:109)
        at gov.sandia.idb.entities.MissingDataNoteEntity.load(MissingDataNoteEntity.java:65)
        at gov.sandia.idb.database.load.MassPercentLoader.loadData(MassPercentLoader.java:153)
        at gov.sandia.idb.database.load.MassPercentLoader.load(MassPercentLoader.java:122)
        ... 2 more
Caused by: java.sql.SQLIntegrityConstraintViolationException: (conn=10) Duplicate entry 'Initial load, MOX LANL data from 29 August 2023' for key 'uq_load_history_table_
load_history_description'
        at org.mariadb.jdbc.internal.util.exceptions.ExceptionMapper.get(ExceptionMapper.java:235)
        at org.mariadb.jdbc.internal.util.exceptions.ExceptionMapper.getException(ExceptionMapper.java:171)
        at org.mariadb.jdbc.MariaDbStatement.executeExceptionEpilogue(MariaDbStatement.java:238)
        at org.mariadb.jdbc.ClientSidePreparedStatement.executeInternal(ClientSidePreparedStatement.java:230)
        at org.mariadb.jdbc.ClientSidePreparedStatement.execute(ClientSidePreparedStatement.java:157)
        at org.mariadb.jdbc.ClientSidePreparedStatement.executeUpdate(ClientSidePreparedStatement.java:192)
        at gov.sandia.idb.entities.LoadHistoryEntity.load(LoadHistoryEntity.java:61)
        ... 6 more
Caused by: java.sql.SQLException: Duplicate entry 'Initial load, MOX LANL data from 29 August 2023' for key 'uq_load_history_table_load_history_description'
        at org.mariadb.jdbc.internal.protocol.AbstractQueryProtocol.readErrorPacket(AbstractQueryProtocol.java:1594)
        at org.mariadb.jdbc.internal.protocol.AbstractQueryProtocol.readPacket(AbstractQueryProtocol.java:1453)
        at org.mariadb.jdbc.internal.protocol.AbstractQueryProtocol.getResult(AbstractQueryProtocol.java:1415)
        at org.mariadb.jdbc.internal.protocol.AbstractQueryProtocol.executeQuery(AbstractQueryProtocol.java:288)
        at org.mariadb.jdbc.ClientSidePreparedStatement.executeInternal(ClientSidePreparedStatement.java:221)
        ... 9 more
Press any key to continue . . .
```

Figure 4. Console Output Showing Duplicate Data Exceptions

The log information written to the command window is also written to a log file in the *logs* folder of the application (NOTE: The *logs* folder is created the first time that the application is run and is not part of the distribution. If the load application has never been run, then there will not be a *logs* folder).

3. CONFIGURING THE LOAD APPLICATION

3.1. Changing Configuration File Name And Location

The location and names of the CSV files that will be uploaded and the database connection information are passed to the load application via its main configuration file. The name and location of the configuration file is passed to the load application as part of the command that starts the application. Figure 5 shows the Windows CMD file opened in the Windows Notepad text editor. The file contains the Java command to run the jar file along with the a -D option that controls logging output and the input string that specifies the location and name of the configuration file. In the figure, the string giving the location and name of the configuration file is circled in red. The input string in the figure is *config/lanl_mox_mass_percent_loader.properties* which means that the loader application will look for a file named *lanl_mox_mass_percent_loader.properties* in the *config* folder of the loader application distribution.

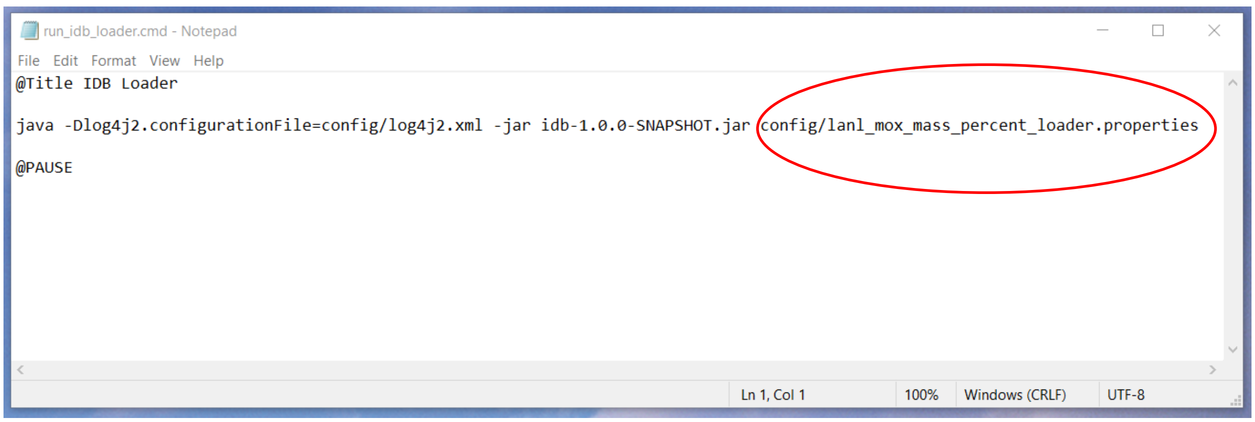


Figure 5. Windows CMD File Opened in Notepad

To change which configuration file is used, change the input parameter to give the relative path and name of the desired configuration properties file. The configuration file can be named anything and located anywhere on the file system as long as the path to the file is relative to the location of the jar file. Figure 6 shows what the input string would be if a configuration file named *my.properties* was placed in a folder named *my_folder* located two directories above the directory containing the load application jar file.

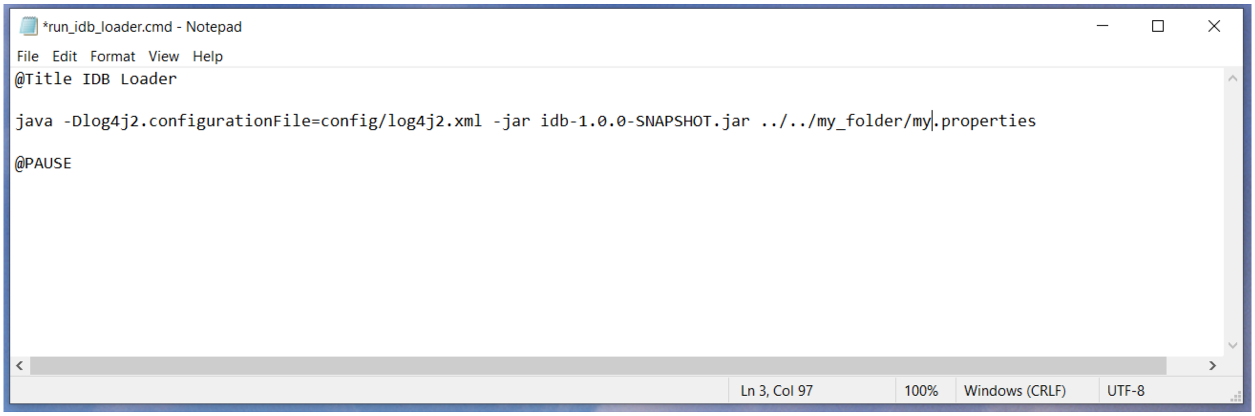


Figure 6. Windows CMD File With Different Input Properties File

The distribution includes five sample property files. Each of the five sample property files is preconfigured to load one of the five sets of CSV spectral data released by the US national laboratories. The sample property files are in the in the *config* folder of the distribution. The CSV spectral data is in the *config/csv_idb_exported* folder; the Lawrence Livermore National Laboratory (LLNL) subfolder contains spectral data released by LLNL and the Los Alamos National Laboratory (LANL) subfolder contains spectral data released by LANL. An exploded directory tree view of the included CSV data is shown in Figure 7.

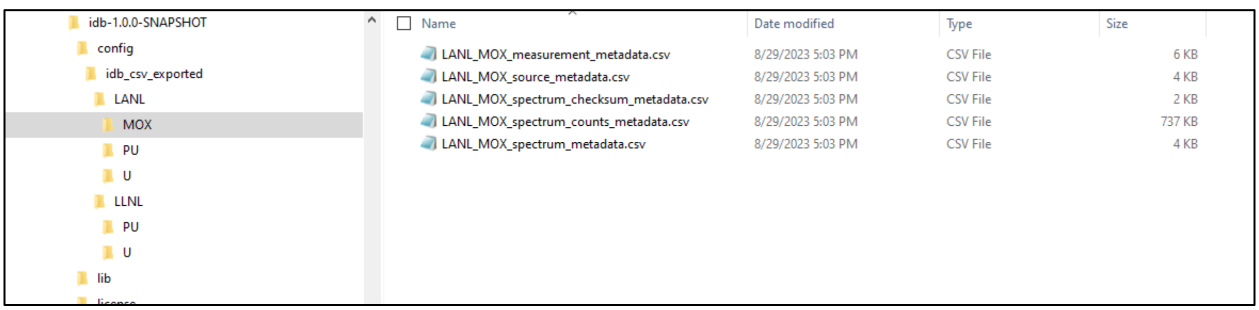


Figure 7. Directory Structure of Included Sample CSV Files

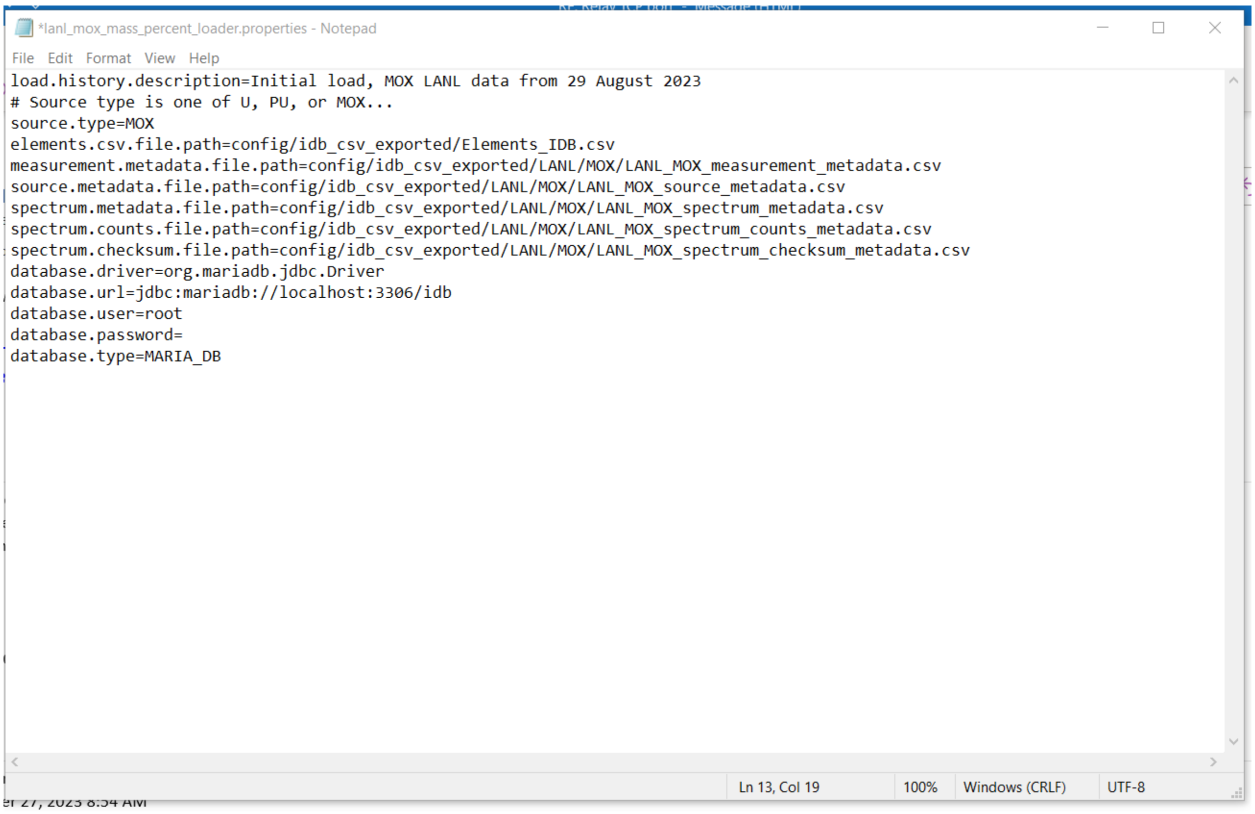
3.2. Main Properties Configuration File

The load application's configuration file is a standard Java property file, which means:

- The file is a text file and can be edited by any text editing application
- Each property entry has a name and a value separated by an equal sign
- Each property is on a separate line
- Lines starting with the character # are treated as comments and ignored

Figure 8 shows the contents of the configuration file *lanl_mox_mass_percent_loader.properties* file. The load application uses the property name to look up what value it should use for a particular configuration setting. For example, the load application will use the property name *database.user* to get the username that should be used to connect to the database: in this case, the value is *root*.

A configuration setting can be changed by changing a property value (Table 1 lists the property names and what configuration setting the property controls). For example, the value *root* should be changed to be a different username because, by default, *root* has admin privileges in MariaDB databases. The username used to connect to the database can be changed to *idb_user* by replacing *root* with *idb_user* as the property value for the property named database.user in the property file (for the load application to function correctly, the user must be a valid MariaDB user with privileges to read, insert, update, and delete records in all *idb* tables.)



```
load.history.description=Initial load, MOX LANL data from 29 August 2023
# Source type is one of U, PU, or MOX...
source.type=MOX
elements.csv.file.path=config/idb_csv_exported/Elements_IDB.csv
measurement.metadata.file.path=config/idb_csv_exported/LANL/MOX/LANL_MOX_measurement_metadata.csv
source.metadata.file.path=config/idb_csv_exported/LANL/MOX/LANL_MOX_source_metadata.csv
spectrum.metadata.file.path=config/idb_csv_exported/LANL/MOX/LANL_MOX_spectrum_metadata.csv
spectrum.counts.file.path=config/idb_csv_exported/LANL/MOX/LANL_MOX_spectrum_counts_metadata.csv
spectrum.checksum.file.path=config/idb_csv_exported/LANL/MOX/LANL_MOX_spectrum_checksum_metadata.csv
database.driver=org.mariadb.jdbc.Driver
database.url=jdbc:mariadb://localhost:3306/idb
database.user=root
database.password=
database.type=MARIA_DB
```


Figure 8. Contents of lanl_mox_mass_percent_loader.properties

Table 1 lists the property names and describes what configuration setting the property can change. The CSV spectral data is a set of five files and five of the properties give the name and location of each of the five files in one set of spectral data.

Table 1. Load Application Configuration Properties

NAME	DESCRIPTION
load.history.description	A string describing the CSV data. The string is inserted into the load_history_description column of the load_history_table and is used to identify data loaded from this upload. The string must not already exist in the database.
source.type	The source material type. Currently must be one of MOX, PU, or U.
elements.csv.file.path	The five CSV files that make up a set of spectral data reference isotope mass fractions, but do not break out the elements separately. The load application uses an additional CSV file to add element names and symbols to the database's element_table. The

NAME	DESCRIPTION
	property value gives the name and location of the elements CSV file. The default value is: *config/idb_csv_exported/Elements_IDB.properties* and only needs to be changed if isotope mass fractions are added for elements aside from: Americium, Plutonium, or Uranium.
measurement.metadata.file.path	The name and location of the CSV measurement metadata file.
source.metadata.file.path	The name and location of the CSV source metadata file.
spectrum.metadata.file.path	The name and location of the CSV spectrum metadata file.
spectrum.counts.file.path	The name and location of the CSV spectrum counts file.
spectrum.checksum.file.path	The name and location of the CSV spectrum checksum file.
database.driver	The name of the JDBC driver used to connect to the database. The value should not be changed unless a different or newer driver must be used.
database.url	The connection string used to connect to the database. The default value assumes MariaDB can be reached on *localhost* using the default *3306* port and that the name of the database is *idb*.
database.user	The name used to connect to the database. The default value *root* should not be used unless MariaDB is only running locally and only hosting the *idb* database and the database is not in production.
database.password	The password of the user whose name is given in the *database.user* property. If the password value is empty, then the load application will ask for the password when it starts running.

4. DATABASE CREATION

The load application can only load data into an existing database; the database can have data from previous uploads, but the database and its tables must exist. The distribution includes files that can create the tables and the table constraints that make up the *idb*. Use the MariaDB command line or the HeidiSQL application to:

1. Create a new database schema named *idb*
2. Execute the create table statements in the *idb_tables.sql* file located in the distribution's *scripts/database/MariaDB/ddl* folder
3. Execute the create constraints statements in the *idb_fk_indes.sql* file that is also located in the distribution's *scripts/database/MariaDB/ddl* folder

The load application will likely not be able to load data if an error occurs during table or constraint creation.

5. APPLICATION LOGIC

The load application's requirements are that it:

- Load one complete set of CSV files.
- Not duplicate data already in the database. For example, every entry in the measurement metadata CSV files has a value set for the *sample.material.type*; however, the entry value can only be one of *MOX*, *PU*, or *U*. When the load application reads an entry in the measurement CSV file, it must check if that entry's *sample.material.type* value already exists in the database and, if so, not reload the material type for that entry.
- Restore the database to the state it was in prior to the beginning of the load if an error occurs during the load.

5.1. Application Start

The load application starts in the *MassPercentLoader*'s main class which simply passes the input parameter containing the name and location of the main configuration file to a *load* method whose only argument is a *String*. That method reads the file into a Java properties instance and passes the properties instance to a second overloaded *load* method that starts the actual load process. The main load call sequence is shown in Figure 9. The entire load is performed inside one transaction so that changes to the database are not committed until all the data is successfully loaded.

```
try (final Connection conn = makeIDBConnection(properties)) {
    logger.info("Loaded connection...");
    conn.setAutoCommit(false);

    try {
        loadElementData(conn,
                Paths.get(properties.getProperty(MassPercentLoader.ELEMENTS_CSV_FILE_PATH)));
        loadSourceTypeData(conn);
        loadData(conn, properties);
        conn.commit();
    } catch (final Exception e) {
        conn.rollback();
        throw new RuntimeException("Loading data, will rollback: ", e);
    }
} catch (final Exception e) {
    throw new RuntimeException("Loading data: ", e);
}
```

Figure 9. Load Code Block

Before loading the CSV spectral data, the application reads and loads data from a CSV file that defines chemical elements that could appear in the source metadata's isotope mass fraction values. The distribution includes an elements.csv file that lists the elements *Americium*, *Plutonium*, and *Uranium*. Unless the spectral data being loaded includes mass fractions of other elements, the elements file does not need to be changed. The application also loads the three allowed source types (*MOX*, *PU*, U*)*. The load application always checks to see if the element and source type data needs to be loaded even though the data will typically not change and will, therefore, only be loaded into a new database. Finally, the spectral data in the CSV files is read and loaded in the *loadData* method.

5.2. Reading CSV Spectral Data

The load application uses the structure built into a set of CSV files to determine the order that the files are loaded. In the CSV files, a measurement metadata entry can reference a source entry by the source entry's source UID field , but a source metadata entry will never reference a measurement UID entry. Similarly, an entry in the spectrum metadata CSV file can reference a measurement entry UID, but a measurement metadata entry never references a spectrum UID entry. Finally, entries in both the spectrum counts and spectrum checksum metadata CSV files reference the spectrum metadata UID, but entries in the spectrum metadata CSV file never reference entries in either the spectrum counts or spectrum checksum CSV files. The load application loads data files in the order of source metadata, measurement metadata, spectrum metadata, spectrum counts, spectrum md5 so that any reference to a UID will be satisfied with data that has already been read and loaded.

Each spectral data file is read into memory as a list of name/value pairs. The name/value pairs are stored in a map whose key and value are String types. The map's key is the name of a field as is given in the CSV file, for example, the *sample.material.type* field in the measurement metadata CSV file. The map's value is the value of the field value given in one line of the CSV file. Figure 10 is a listing of the method that reads and parses a metadata file

```
private List<Map<String, String>> readMetadaData(final Path metadataCSVFilePath) {
    logger.info("Reading metadata from: " + metadataCSVFilePath);
    final List<Map<String, String>> nameValuePairsList = new ArrayList<>();
    try {
        List<String> headerList = new ArrayList<>();
        for (final String line : Files.readAllLines(metadataCSVFilePath)) {

            Map<String, String> nameValuePairs = new HashMap<>();
            final String[] values = parseLine(line);
            if (values.length == 0) {
                continue;
            } else if (values[0].toUpperCase().startsWith("UID")) {
                logger.info("Loading header data: " + line);
                headerList = Arrays.asList(values);
            } else {
                logger.info("Loading metadata: " + line);
                for (int i = 0; i < values.length; i++) {
                    nameValuePairs.put(headerList.get(i), values[i]);
                }
                nameValuePairsList.add(nameValuePairs);
            }
        }
    } catch (final Exception e) {
        throw new RuntimeException("Loading metadata: ", e);
    }
    return nameValuePairsList;
}
```

Figure 10. Read Metadata File Method

The *parseLine* method in the listing handles special cases of input such as a comment or a comma occurring inside an input field. The line containing the names of the field is found by assuming that the line starts with the characters UID.

The load application loops over the returned list and passes the map of name/value pairs to the load method of the class that represents the main table for that type of CSV data. For example, when the load application reads the spectral data in the measurement metadata file, it passes a map of name/value pairs representing one line of data to the *MeasurementEntity*'s *load* method.

5.3. Transforming Data

The load application loads one line of data from one CSV file at a time by:

1. Transforming the name/value pair map of CSV data into instances of classes that mirror the tables in the database. An instance of a class created during the transformation corresponds to one row of data in one table in the database
2. Checking to see if either a previously created class instance from this data upload or a row in the database from previous data uploads contains the same values as the newly created instance. If so, then the class instance created from the CSV data is replaced by either the previously created instance or by an instance that is created from values obtained from the database. If not, then the newly created instance is inserted into the database and its database key is set. Note that the key value is retrieved as part of the insert, but the insert is not committed.
3. Using a recursive like pattern to build up the tree structure of table relational dependencies.

Every entity class has a load method and every load method:

1. Passes the name/value pair map to a find method which:
    a. Reads values from the name/value pair map and sets each value into the corresponding field of the class. If needed, the map value is converted from a String to the type of field. If entity field is another entity class, then the name/value pair map is passed to the load method of the field entity class. This recursive like behavior is shown in the first few lines of the listing in Figure 11 where the Detector entity is setting its Detector Type and Analyzer entities by passing the name value pairs map to the *load* methods of the *DetectorType* and *Analyzer* classes.
    b. Checks a local cache to see if an entity of this class type has already been loaded. If so, then that instance of the class is returned and the new instance created is thrown away
    c. Checks the database to see if the table this class instance mirrors has the same field values as the class instance. If there is an existing row, then that row's technical key is set into the entity's id field so that the *load* method will not insert the new entity. The entity is then inserted into the local cache and the entity is returned. This is shown in Figure 11. Note that the comparison to find existing cached or database entities only compares the non-id fields of the entity because the new instance created from the name/value pair map will not yet have the technical key generated when an entity is inserted into the database

```java
public static final DetectorEntity find(final Connection conn, final Map<String, String> nameValuePairs) {

    final DetectorEntity entity = new DetectorEntity();
    entity.setDetectorTypeEntity(DetectorTypeEntity.load(conn, nameValuePairs));
    entity.setAnalzyerEntity(AnalyzerEntity.load(conn, nameValuePairs));

    for (final DetectorEntity cachedEntity : DetectorEntity.cache) {
        if (DetectorEntity.haveEqualValues(entity, cachedEntity)) {
            return cachedEntity;
        }
    }

    // now check the database...
    try (final PreparedStatement statement = conn
            .prepareStatement(DetectorEntity.SELECT_DETECTOR_BY_DETECTOR_TYPE_ANALYZER)) {
        if (entity.getDetectorTypeEntity() == null) {
            statement.setNull(1, Types.INTEGER);
        } else {
            statement.setInt(1, entity.getDetectorTypeEntity().getDetectorTypeId());
        }
        if (entity.getAnalzyerEntity() == null) {
            statement.setNull(2, Types.INTEGER);
        } else {
            statement.setInt(2, entity.getAnalzyerEntity().getAnalyzerId());
        }

        try (final ResultSet rs = statement.executeQuery()) {
            if (rs.next()) {
                entity.setDetectorId(rs.getInt("detector_id"));

                DetectorEntity.cache.add(entity);
                return entity;
            }
        }
    } catch (final Exception e) {
        throw new RuntimeException("Retrieving Analyzer for name: " + entity, e);
    }

    return entity;
}
```

Figure 11. Find Method of Detector Entity Class

2. If the find method doesn't return anything, returns an error
3. If the find method returns an entity that has an id set, then return that entity because it means that the data is already in the database
4. If an entity is returned that does not have a key set, then:
    a. Insert the entity and ask that the technical key id generated during the insert is returned
    b. Set the returned key to be the id of the entity
    c. Set the entity into the local cache
    d. Return the entity

Additional processing steps, if needed, are handled by the entity best suited to perform whatever additional processing is required. For example, in the CSV measurement metadata data files, the *configuration.attenuators* can include multiple material types in the input string. In the database's *attenuator_table*, however, each material type is in its own row. The *AttenuatorEntity* knows how to split the *configuration.attenuators* string into a set of *AttenuatorEntity* instances and the entity's *find* method returns a set of entities to its *load* method. The *AttenuatorEntity load* method than tests each entity in the set to see if it needs to be inserted into the database.

When all the data from all the files is transformed, the sequence of inserts is committed. The commit will only contain new data and the relationships of the new data will be correct when inserted.

6. IMPROVEMENTS

1. Replace static methods. The static methods in the entity classes are a remnant of the development process. Abstracting duplicate code into a base class that could be subclassed by entity classes would make the code simpler.
2. Replace with an Object Relational Mapping (ORM) framework like Hibernate. The entity classes are already very ORM like. Replacing the current quasi-ORM with a real ORM would make it easier to add support for additional databases.
3. Configurable parsers: The current parser hard codes parsing rules for special case data found in the CSV files from the US national labs. A line parser that had configurable rules would let other providers of data generate parsing rules for their own special cases.

DISTRIBUTION

Email—Internal

Name	Org.	Sandia Email Address
Jesse John Bland	6832	jjbland@sandia.gov
Steven Robert Schwartz	6832	srschwa@sandia.gov
Technical Library	1911	sanddocs@sandia.gov

Email—External

Name	Company Email Address	Company Name
Dipti Dipti	D.Dipti@iaea.org	International Atomic Energy Agency
Ludmila Marian	l.marian@iaea.org	International Atomic Energy Agency
Jonathan Dreyer	dreyer4@llnl.gov	Lawrence Livermore National Laboratory
Duc Vo	ducvo@lanl.gov	Los Alamos National Laboratory

This page left blank

Sandia National Laboratories

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.