Using Models in Production

DATABASE SCORING

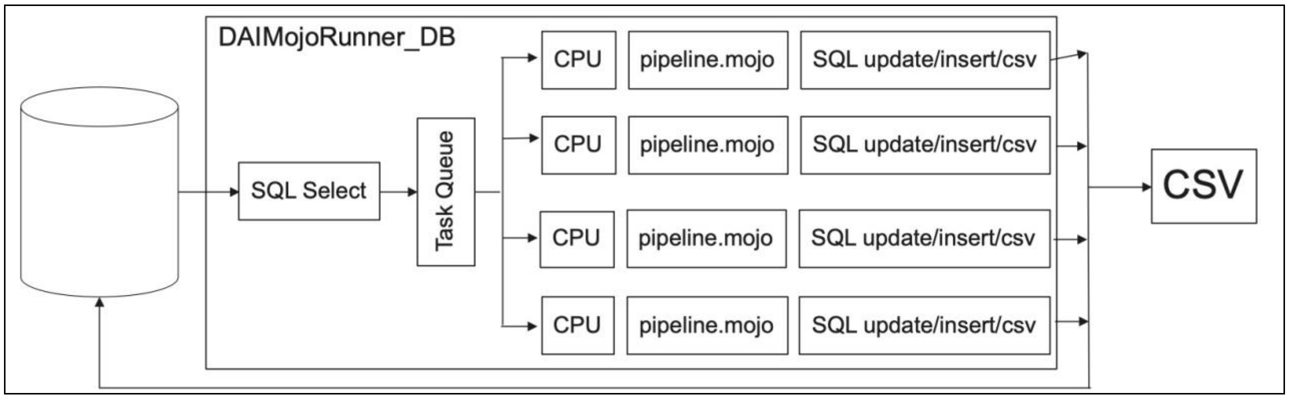
H2O.ai

Database Scorer

The Database Scorer uses a JDBC type 4 to connect to the database and score rows of data. This version support Driverless AI models a sperate implementation supports H2O-3 models.

The scorer executes as a standalone application external to the Database and uses a JDBC driver to connect, as rows are retrieved, they are scored and the results either saved to the original table or inserted to a new table, optionally they can be written to a CSV for bulk uploading or other processing.

When scoring multiple rows are processed at the same time, based on the number of CPU's that are available.



Distribution

The application is provided as a single jar file a properties file is generated so that environment specific options can be used.

Requirement

- Java JDK 1.8
- Driverless AI model, pipeline.mojo
- Driverless AI license
- Database Score https://s3.amazonaws.com/artifacts.h2o.ai/releases/ai/h2o/dai-custom-scorers/DAI-1.8.9/index.html
- Database vendors JDBC driver

Configuration

Using the pipeline.mojo as many of the settings will be auto generated and saved into the properties file so any environment specific changes can be applied.

Create Properties

A properties file should be created for each model, as each model may have different SQL to run, a different database to connect too or a different model name.

When the scorer runs to make predictions the properties file is passed on the command line, but first we must create the file.

```
java -Dmodelname=riskmodel.mojo -Dcreateproperties=true -
Dai.h2o.mojos.runtime.license.file=/Driverless-AI/license/license.sig -cp dist/ai.h2o.mojos.jar
ai.h2o.mojos.db.DAIMojoRunner_DB
Properties file (DAIMojoRunner_DB.properties.riskmodel.mojo) created
```

The command used the model riskmodel.mojo to create a properties file called DAIMojoRunner_DB.properties.riskmodel.mojo the filename can be renamed if needed.

Parameters

The properties file enables specific settings to be set for the environment.

Description

A friendly description of the model this is used by Model Monitoring and also makes it easier at deployment time to remember the models specific usecase.

```
Description = A model for predictions
```

EnvVariables

This setting enables environment variables to use used to set specific configuration parameters for example:
SQLUSER='$USER' would set the parameter to the value from the environment variable $USER is EnvVariables is set to True

```
EnvVariables = true
```

IgnoreEmptyStrings

When parsing ignores features with empty string

```
IgnoreEmptyStrings = true
```

IgnoreChecks

This verifies that that row from the databased is parsed into the correct number of features, this helps to protect against the wrong data shape being used and other parsing errors. Only set to True with guidance.

```
IgnoreChecks = False
```

IgnoreSQLChecks

The SQL check verifies that the SQL columns requested match the number of features the model requires, as the SQL is auto generated based on the model the order of columns is correct for the model, however, if functions like Average or Coalease are added this may need to be set to True, which would ignore checks and the user is responsible to ensure the order and columns are correct for the model. Only set to True with guidance.

```
IgnoreSQLChecks = False
```

SetNullAs and NullCharacters

Some databases will return the string NULL for a column that have the value null, models use NA to represent NULL. This parameter defines the behavior where the values listed on the NullCharactaters parameter will matched and set to the value specified on SetNullAs

```
SetNullAs = NA
NullCharacters="NA|NULL"
```

ForceConnection and ForceConnectionClass

Enables loading a JDBC specific class for example when debugging connection issues both parameters are used together.

```
ForceConnection = False
ForceConnectionClass =
```

ModelName

Absolute path and filename of model to deploy

```
ModelName = riskmodel.mojo
```

SQLConnectionString

This defines the JDBC connection string to database. This must be changed for each environment, below are a few common examples but any JDBC type 4 driver that support JDK 1.8 should work.

Only one SQLConnectionString can be defined.

Snowflake

```
SQLConnectionString=jdbc:snowflake://myaccount.snowflakecomputing.com/?warehouse=WH
&db=LENDINGCLUB&schema=PUBLIC&application=H2OScoring
```

Postgres and Redshift

```
SQLConnectionString=jdbc:postgresql://192.168.1.35:5432/mydatabase
```

Cassandra

```
SQLConnectionString=jdbc:cassandra://127.0.0.1:9042/lendingclub
```

Microsoft SQL Server

```
SQLConnectionString=jdbc:sqlserver://'$DB_HOST';databaseName='$DB_NAME';user='$DB_
USER';password='$DB_PWD'
```

Microsoft SQL Server with AD security enabled

```
SQLConnectionString=jdbc:sqlserver://192.168.1.173:1433;databaseName=LendingClub;integratedSecurity=true
```

Azure Synapse

```
SQLConnectionString=jdbc:sqlserver://<instance>.sql.azuresynapse.net:1433;database=<database>;user=<username>;password=<password_here}>;encrypt=true;trustServerCertificate=false;hostNameInCertificate=*.sql.azuresynapse.net;loginTimeout=300;
```

Elastic Search

```
SQLConnectionString=jdbc:es://your_es_host:9200
```

Oracle

```
SQLConnectionString=jdbc:oracle:thin:@192.168.1.174:1521:ORCLCDB
```

Teradata

```
SQLConnectionString=jdbc:sqlserver://192.168.1.173/databaseName=ANALYTICS,user=sa,password=h2o123,TMODE=ANSI,CHARSET=UTF8
```

Athena

```
SQLConnectionString=jdbc:awsathena://athena.<region>.amazonaws.com:443;S3OutputLocation=s3://<results-location>/;AwsCredentialsProviderClass=com.simba.athena.amazonaws.auth.InstanceProfileCredentialsProvider;Workgroup=DataAnalyst
```

Databricks

```
SQLConnectionString=jdbc:spark://<instance>.azuredatabricks.net:443/default;transportMode=http;ssl=1;httpPath=<sql/protocolv1/>;AuthMech=3;UID=token;PWD=<personal-token>
```

Google Big Query

```
SQLConnectionString=jdbc:bigquery://https://console.cloud.google.com/bigquery;ProjectId=<project-id>;OAuthType=0;OAuthServiceAcctEmail=<service-account-name>;OAuthPvtKeyPath=/gbq_sevice_account.json
```

SQLUser

The username to use when connecting to the datasource if not defined on the SQLConnectionString

```
SQLUser =
```

SQLPassword

The password for SQLUser this is encrypted in the properties file. See the parameter SQLPrompt for details on how setting the password.

```
SQLPassword =
```

SQLPrompt

This defines how the program will use the password for connecting to the database.

Valid settings are:

- SQLPrompt=true causes the program to wait for the password to be typed helpful for scoring were the user interacts with the program directly.
- SQLPrompt=encrypt prompts for the password so it can be saved into the properties file helpful for batch and unattended scoring

Example: with SQLPrompt=encrypt in the properties file.

```
Please enter JDBC password for user :devops

set the parameters in properties file as follows:

SQLPrompt = false

SQLPassword = YWFhYWFhYWFhYQ
```

Copy the value SQLPassword to the properties file and set SQLPrompt to false

```
SQLPrompt = encrypt
```

SQLSelect

The SQL statement to execute, these are the rows that will be passed to the model, this statement is auto generated when the program is run with -Dcreateproperties=true.

Both the <add-table-key-or-unique-field> and <add-table-name> values should be substituted for scoring, see the SQLKey for more details.

```
SQLSelect = select <add-table-key-or-unique-field>, model-feature-1, model-feature-2 from
<add-table-name>
```

SQLAddCols
This parameter defines additional fields to include in the output used for SQLWriteBatch or CSV/CSVWithHeader output.

An example would be, if the output table required the timestamp of the scoring run and the email address column from the database table used in the select but the email address column was not used by the model.

```
SQLAddCol=current_timestamp, email
SQLWriteBatch=SQLWriteBatch=insert into results("ID","TIME_SCORED", "EMAIL",
"BAD_LOAN_0", "BAD_LOAN_1")  values(?, ?, ?, ?, ?)
```

When using SQLAddCols only SQLWriteBatch and CSV output types are supported.

If the output destination is a data, then using StartScripts and EndScripts can also be used for more complex operations, see the section *Pre and Post Processing.*

SQLWrite
This parameter defines how to save the predictions.

Valid options are:
- SQLWrite=update <add-table-name> set where <add-SQLKey-value>=
- SQLWrite=insert into <add-table-name>() values()
- SQLWrite=CSV
- SQLWrite=CSVWITHHEADER

The SQL is dynamically generated at runtime.

```
SQLWrite=update <add-table-name> set where <add-SQLKey-value>=
SQLWrite=insert into <add-table-name>()  values()
SQLWrite=CSV
SQLWrite=CSVWITHHEADER
```

SQLWriteCSVFilename
If SQLWrite is using a CSV file to save the output, this parameter can be used to make the filename a specific name, otherwise the CSV is written to standard out.

The filename can be either a fully qualified path or just the filename.

```
SQLWriteCSVFilename =
```

SQLWriteBatch and SQLWriteBatchSize
If a large number of rows are being used to predict, then enabling SQLWriteBatch= will enable SQL prepared statements to be dynamically generated, these are more efficient for most databases as update and commit operations will occur based on the SQLWriteBatchSize.

The SQLWriteBatchSize is per Thread (see Threads runtime setting)

```
SQLWriteBatch = insert into results("id", "bad_loan.0", "bad_loan.1")  values(?, ?, ? )
SQLWriteBatchSize = 1000
```

Here are a few examples:
Add database timestamp and generation count from environment variable:

```
SQLWriteBatch=insert into resultsUPDATE("ID", "MEMBER_ID", "BAD_LOAN_0",
"BAD_LOAN_1", "SCORED", "GENERATION")  values(?, ?, ?, ?, CURRENT_TIMESTAMP,
'$GENERATION' )
```

If writing Shapley values, then this syntax can be used, as the column names are generated by the model, also see SQLFriendlyName setting.

```
SQLWriteBatch=insert into A.resultshapf()  values()
```

SQLKey

Defines the Table key or some unique value to use in the SQLWrite, this helps to ensure the database operations are efficient by allowing the SQLKey value to be used in a 'where' clause while updating the database and is insert was used on SQLWrite when the SQLKey can be used to join to the original table.

If no SQLKey is specified and threads is greater than 1, then the following message will the shown in standard out and no scoring will occur.

```
No SQLKey specified and Threads greater than 1, this means:
SQL updates will force a table scan
SQL inserts will be in a random order.
```

```
SQLKey = <add-table-key-or-unique-field>
```

SQLKeyType

This is used with the -Dinspect=true so that the generated DDL for the database table can be created correctly it is also used when using the ShapPredictContrib=true setting.

```
SQLKeyType = <any-valid-DDL-type>
```

SQLAddCols

If additional columns are required for saving the results but those columns are not required by the model, then they can be specified as a comma separated list, for example:
SQLAddCols=member_id, CURRENT_TIMESTAMP, '$GENERATION'

This would add these three fields into the output for the SQLWrite target.

```
SQLAddCols=<column-names>
```

SQLPrediction

Specify the prediction to return from the scorer, 0, 1 or blank for ALL

```
SQLPrediction = 0
```

SQLPredictionLabel

Save prediction to this column name in the table, blank used model target name, this is used when for example the target name used in the model, needs to be saved to a specific column name.

```
SQLPredictionLabel =
```

SQLFieldSeperator

The character that is used to separate the columns that are returned from the database. This defaults to a comma (,)

```
SQLFieldSeperator =,
```

InternalFieldSeperator

InternalFieldSeperator should be used if the data contains the default comma character, for example if a column contained a comma, then parsing from column to features would fail.

If “1st Street, Apt A” was in a single column, then the default would be a split into two fields (“1st Street” and “Apt A”) setting the InternalFieldSeperator=@ would split the column into one field (“1st Street, Apt A”).

```
InternalFieldSeperator = ,
```

SQLWriteFieldSeperator

When writing a CSV output (SQLWrite) this defines the field separator

```
SQLWriteFieldSeperator = ,
```

SQLFieldCaseName

This option determines what case to convert the feature names to uppercase or lowercase, the default is the case in the model.

```
SQLFieldCaseName = <uppercase | lowercase>
```

SQLWriteFieldFriendlyNames

Feature names that contain invalid characters, such as “.: “ are replaced with “_” so they are compatible with the database.

```
SQLWriteFieldFriendlyNames = true
```

SQLWriteIgnoreNames

When the target column names in SQLWriteBatch statement need to be overridden from the model target names to a table specific name, specify the column names on the SQLWriteBatch paraemeter and set this parameter to True.

```
SQLWriteIgnoreNames=false
```

SQLWriteIgnoreNamesClean

If the prediction returned by the model is enclosed in quotes some databases may not cast it to the correct column type. This parameter can be used to remove specific characters before saving the prediction to the table, usually this is used in conjunction with SQLWriteBatch.

```
SQLWriteIgnoreNamesClean='
```

SQLStartScript

This is a database script to run before scoring, ideal for clearing a result table

```
SQLStartScript =
```

SQLEndScript

This is a database script to run after scoring, ideal for comparing row counts

This can also be very handy way to join the results to the original data using the SQLKey, see the section Scoring Post Processing.

```
SQLEndScript =
```

Threshold

This sets a string to write if the prediction if larger than the specified value

```
Threshold=Yes,>=,0.9,No
```

This example sets the returned prediction to Yes if the score is equal or greater then 0.9 otherwise it returns No.

ShapPredictContrib

If Shapley contributions should be returned with the prediction, then enable this option.

```
ShapPredictContrib=true
```

Once enabled only the SQLBatchWrite function is available and the SQLWriteBatch only needs to specify the target table and the SQLKeyType must be specified. An example SQLWriteBatch statement:

```
SQLWriteBatch=insert into resultshap()  values()
```

This would save the prediction and shapley contributions to the table resultshap, use the -Dinspect=true to generate an example DDL for the specific model.

ShapPredictContribOriginal

If original Shapley contributions should be returned with the prediction, then enable this option.

```
ShapPredictContribOriginal=true
```

This as the same output requirement as ShapPredictContrib. It's also possible to write the results to a CVS with a header by setting: SQLWrite=csvwithheader

Runtime Parameters

In addition, the properties file that define the model and datasource, the command line pararmeters control how the runtime will execute in the environment.

Parameter Summary

Command line -D Parameter	Default Value
propertiesfilename	DAIMojoRunner_DB.properties
verbose	false
override	false
stats	false
inspect	false
createproperties	false
compatible	false
stoponmismatch	false
capacity	Number of threads + 75%
threads	Number of available CPU's
errors	false
health	false
healthhost	localhost
healthport	8082
modelmonitor	false
Modelname	pipeline.mojo

Propertiesfilename

This allows the name of a custom properties file to be used. Usually each model will have a unique properties file.

```
propertiesfilename=
```

Verbose

Enabling this option, prints a lot of runtime diagnostic information to standard out.

```
verbose=
```

Override

If the number of threads is greater than 1 and no SQLKey is in the properties file, then this option will allow scoring to continue, however the input row and output order of rows will not match, this should only be used with guidance.

```
override=
```

Stats

If enabled, then when scoring is completed statistics on the number of rows read from the database and the per thread statistics are displayed.

```
Total selected rows 39029
Thread-59 Rows Read 1142 Scored 1142 Error 0 Queue Empty false
```

Inspect

This parameter prints to standard out details about the model, suggested DDL's for the tables and details of the environment.

```
Inspect=
```

This can be helpful when deploying to check a model or the expected features is uses.

```
Details of Model: pipeline.mojo
UUID: b2fce6c1-6ddc-4c78-8355-f437945a613c
Input Features:
0 =  Name: loan_amnt Type: Float32
1 =  Name: term Type: Str
2 =  Name: int_rate Type: Float32
3 =  Name: installment Type: Float32
4 =  Name: emp_length Type: Float32
5 =  Name: home_ownership Type: Str
6 =  Name: annual_inc Type: Float32
7 =  Name: verification_status Type: Str
8 =  Name: addr_state Type: Str
9 =  Name: dti Type: Float32
10 =  Name: delinq_2yrs Type: Float32
11 =  Name: inq_last_6mths Type: Float32
12 =  Name: pub_rec Type: Float32
13 =  Name: revol_bal Type: Float32
14 =  Name: revol_util Type: Float32
15 =  Name: total_acc Type: Float32
```

Output Features

0 = Name: bad_loan.0 Type: Float64

1 = Name: bad_loan.1 Type: Float64

Example SQL Table DDL

this is generic SQL, verify this for the specific database.

Verify the SQLKey and the column data types.

```
CREATE TABLE add-table-name (
        id INTEGER,
        loan_amnt FLOAT,
        term VARCHAR(1024),
        int_rate FLOAT,
        installment FLOAT,
        emp_length FLOAT,
        home_ownership VARCHAR(1024),
        annual_inc FLOAT,
        verification_status VARCHAR(1024),
        addr_state VARCHAR(1024),
        dti FLOAT,
        delinq_2yrs FLOAT,
        inq_last_6mths FLOAT,
        pub_rec FLOAT,
        revol_bal FLOAT,
        revol_util FLOAT,
        total_acc FLOAT
);

CREATE TABLE results (
        id INTEGER,
        bad_loan.0 FLOAT,
        bad_loan.1 FLOAT
);

Suggested configuration for properties file:

select <add-table-index>, loan_amnt, term, int_rate, installment, emp_length, home_ownership,
annual_inc, verification_status, addr_state, dti, delinq_2yrs, inq_last_6mths, pub_rec, revol_bal,
revol_util, total_acc from <add-table-name>

update <add-table-name> set where <add-table-index>=

Change the values in <> above and manually test before using them in the program.

Use the option SQLPrompt=encrypt to help encrypt the JDBC password, you will be prompted
to type in the password.
That output can then be saved into the properties file.
Use SQLPrompt=true to always get prompted and not save the encrypted password.

The System has 16GB available physically. This program is using 0GB Consider adjusting -Xms
and -Xmx to no more than 12GB
The System has 16 Processors.
```

Createproperties

Setting this option to true and will create a properties file based on the model specified on the modelname parameter.

```
createproperties=
modelname=
```

Compatible

This setting attempts to check the version of the runtime with the version used to create the model.

```
compatiable=
```

example output of execution.

```
Checking versions

Built using Driverless AI version "1.8.1"

Model version created using "2.1.12" and version "1.8.1" of Driverless AI
```

Stoponmismatch

Used in conjunction with compatible, if the versions to not match or version information is not available execution stops.

```
stoponmismatch=
```

example of execution

```
Checking versions

Built using Driverless AI version "1.5.5"

Model version created using "0.13.18" and version "1.5.5" of Driverless AI

This mojo was created with a different version of the mojo2-runtime.jar halting.
```

Capacity

This is the size in entries of the buffer used to pass data from the database reader thread to the scoring thread.

The default size is the number of threads +75%

```
capacity=
```

Threads

This is the number of threads to use for scoring. The default is the number of available CPU’s.

```
threads=
```

Errors

If errors occur during scoring, then the error is displayed in standard out.

```
errors=
```

Example output when an error happens.

```
WARNING input row (not, verified) contains internalFieldSeperator ,
```

Health

Setting this parameter to true enables a http listener during scoring so that details are available live. This is useful with large scoring jobs to monitor progress.

```
curl http://127.0.0.1:8082/health

Driverless AI Database Scorer v3.0

Time: Fri May 07 15:56:42 EDT 2021

Server: prodsvr-2

Model name: pipeline.mojo

Rows selected: 6168

Rows with errors: 0

Rows processed: 4912
```

Healthhost

This is the host to use to bind for the health listener, if the environment has a separate management interface, use that value here.

```
healthhost=
```

Healthport

This is the port for the health listener.

```
healthport=
```

ModelMonitor

If the H2O.ai model monitoring component is available, enabling this option will create data that can be used by model monitoring.

```
modelmonitor=
```

DelayStart

This parameter if set (defaults to off) will not start the scoring threads until the first set of rows has been read from the data source, this can reduce overall elapsed time, by prepopulating the some of the data to be scored into the process.

```
delaystart=
```

Pause

This is used to indicate if the scoring thread should pause for a random number of milliseconds between 0 and 50 when a commit to the database is performed, this is used to help reduce the impact to the database in high concurrency envionments.

```
pause=
```

Execution

The scorer can be used on any platform that has a JDK 1.8 JVM and access to the database with a type 4 JDBC driver.

Linux example:

```
java -Dhealth=true -Dstats=true -Derrors=true -Xms10g -Xmx10g -
Dpropertiesfilename=riskmodel.properties -cp
/sqljdbc_6.0/enu/jre8/sqljdbc42.jar:ai.h2o.mojos.jar ai.h2o.mojos.db.DAIMojoRunner_DB
```

Windows example:

```
java -Dthreads=3 -Xms4g -Xmx4g -Dpropertiesfilename=riskmodel.properties -
Dai.h2o.mojos.runtime.license.file=c:\DB\license.sig -
Djava.library.path=C:\sqljdbc_6.0\enu\auth\x64 -cp c:\sqljdbc42.jar; ai.h2o.mojos.jar
ai.h2o.mojos.db.DAIMojoRunner_DB
```

Using Windows Active Directory

Some scoring environments want to use the credentials of the user executing the scoring rather than a service account.

The score can use the AD for this, follow these steps:

1. Add to the command line: `-Djava.library.path=\sqljdbc_6.0\auth`
2. Add to the SQLConnectionString `;integratedSecurity=true`

For example the command line would look like this:
```
java -Xms4g -Xmx4g -Dlogging=true -
Dpropertiesfilename=DAIMojoRunner_DB.properties-MSSQL-Auth -
Dai.h2o.mojos.runtime.license.file=c:\DB\license.sig -
Djava.library.path=C:\sqljdbc_6.0\enu\auth\x64 -cp
c:\\sqljdbc42.jar;dist/DAIMojoRunner_DB.jar daimojorunner_db.DAIMojoRunner_DB
```

The SQLConnectionString in the properties file would look like this:
```
SQLConnectionString=jdbc:sqlserver://192.168.1.173:1433;databaseName=LendingClub
;integratedSecurity=true
```

The SQLUser and SQLPassword parameters in the properties file would be commented out.

Pre and Post Processing

Before scoring starts or nnce scoring has completed, sometimes additional processing is required, in many cases this can be achieved using the SQLStartScript or SQLEndScript functions.

This function executes a set of SQL commands, the following gives some simple examples.

```
SQLStartScript=ClearTimeStamp.sql

#SQLEndScript=DisplayTable.sql

#SQLEndScript=ResultsJoin.sql
```

Many databases have SQL that is unique to a specific vendor, or they have features that can make tasks like this very simple, use the following as an example and modify for your specific database and needs.

Be very careful, the database executes these scripts using the authenticated user, any valid SQL statement will be executed in the database, make sure you test and validate the results before using in production.

Pre-Process: Clear Results

If a table is updated with the prediction results, the existing results you might want to remove the existing scores.

```
UPDATE Customer SET SCORED = NULL, BAD_LOAN_0 = NULL, BAD_LOAN_1 =
NULL;
```

Post-Process: Display Number of Results

As a manual verification count the input and output rows.

```
SELECT getdate();
SELECT COUNT(*) FROM Customer;
SELECT COUNT(*) FROM RESULTS;
```

Post-Process: Display Results

In addition to saving the results into the database table, a copy of the results is required for downstream processing as a file.

```
File: DisplayTable.sql
SELECT * FROM RESULTS;
```

Post-Process: Join with Original Table

This example creates a table using the original data and the results, using the SQLKey=id as the common field to join on.

```
File: ResultsJoin.sql
DROP table RESULTSJOIN;

CREATE TABLE RESULTSJOIN (
        BAD_LOAN_0 DOUBLE,
        MEMBER_ID INTEGER,
        LOAN_AMNT  DOUBLE,
        TERM varchar(20)
);

INSERT into RESULTSJOIN
SELECT RESULTS.BAD_LOAN_0,
          LOANSTATS4.MEMBER_ID,
          LOANSTATS4.LOAN_AMNT,
          LOANSTATS4.TERM
FROM LOANSTATS4
INNER JOIN RESULTS ON LOANSTATS4.ID = RESULTS.ID;
```

Post-Process: Snowflake Upload using Stage Table

Depending on the size of the data it can be more efficient to update the data using a bulk load approach. This example is for Snowflake, but other databases such as RedShift also can use similar bulk update scripts.

```
SELECT GETDATE();
SELECT CURRENT_REGION();
SELECT CURRENT_WAREHOUSE(), CURRENT_DATABASE(), CURRENT_SCHEMA();
DELETE FROM "RESULTS";
SELECT COUNT(*) FROM "RESULTS";
PUT FILE://RESULTS.CSV @~/STAGED1;
COPY INTO RESULTS FROM @~/STAGED1;
SELECT COUNT(*) FROM "RESULTS";
```

Monitoring

If the health parameter is passed on the command line, then a http listener is available to show the process of scoring. See the command line parameters health, healthhost and healthport for details about these parameters.

Available Metrics

The metrics that are collected are:

Metric	Description
Time	Current time of reported metrics
Server	Host name where the scoring is executing
Model name	Name of model being executed
Rows selected	Number of rows read from the database
Rows with errors	Number of parsing errors detected during scoring
Rows processed	Number of rows scored
Rows committed	Number of rows that have been committed to the database