# AsciiIngest

Adrian Partl

Version 0.91beta

## Part I
## Introduction / Intent

AsciiIngest is build on the DBIngestor database ingestion library. This software can be used, to ingest large datasets into various databases. The data has to be in ASCII format and needs to be row based. The databases that are supported are given by the capabilities of the DBIngestor library.

AsciiIngest is largely inspired by the MS SQL Server *bcp* tool. Similar to *bcp*, AsciiIngest uses a *structure file* to describe the structure of the ASCII file and how it should be read. Prominent features of AsciiIngest are the ability to read data from a possible header section in the ASCII file, apply constant values to columns, and apply assertion and convertion functions to the data. Assertion and convertion functions need to be implemented in the DBIngestor library and can be easily added.

This document describes basic usage of AsciiIngest and gives a brief overview of all its capabilities. However at the current state, this document is far from complete and should be extended to be more precise.

## Part II
## Compilation

AsciiIngest uses the following dependancies, that must be met before compilation:

- cmake
- Boost Library (especially Program Options)
- DBIngestor library (https://github.com/adrpar/DBIngestor)
- One or more of the following database drivers:
  - Sqlite3
  - mysql C AIP
  - ODBC (tested with UnixODBC and MS ODBC driver)
  - in order to connect to SQL Server from Unix: FreeTDS

When these dependancies are met, you can easily compile AsciiIngest by:

1. cd into AsciiIngest directory. Make sure that you have installed the DBIngestor library appropriately.
2. edit CMakeLists.txt or use the cmake gui tools
   You can turn any of the DB interfaces on and off, by commenting out any of SQLITE3_BUILD_IFFOUND, MYSQL_BUILD_IFFOUND, and ODBC_BUILD_IFFOUND options in the CmakeList.txt file.
3. mkdir build
4. cd build
5. cmake \
   -DDBINGESTOR_LIBRARY_PATH:PATH=/usr/local/DBIngestor/lib \
   -DDBINGESTOR_INCLUDE_PATH:PATH=/usr/local/DBIngestor/include ..
6. make
7. done

## Part III
# Command line options

The way AsciiIngest behaves, can be heavily customised through command line options. With *AsciiIngest –help* or *AsciiIngest -?* you can get an overview of all the possible options.

- **–structFile:** The path to the structure file used to read the data file. See the chapter below on how to write structure files.
- **–data, -d:** Path to the ASCII data file that will be ingested into the database.
- **–system, -s:** The data base system to be used for the ingest. This depends on the available dependencies and libraries on your system. The following systems are supported up to now:
  - **sqlite3**: the SQLite3 API is used. This means, that –path needs to be specified to specify the path to the database file
  - **mysql**: the MySQL API is used to connect to a MySQL database
  - **unix_sqlsrv_odbc**: the UnixODBC together with the FreeTDS driver is used, to connect to a remote MS SQL Server database
  - **sqlsrv_odbc**: the MS ODBC SQL Server driver is used to connect to a SQL Server database (version 10.0 driver)
  - **sqlsrv_odbc_bulk**: same as above, however using ODBC bulk operations. (WARNING: Slow method... Why not clear yet)
  - **cust_odbc**: using a ODBC connection to connect to a database. Use the –socket option to specify the ODBC connection string for the driver to be used. These should contain any options except user name, password, host, port. Don't forget to end the statement with ';'! Example:

    ```
    --socket="SQL Server Native Client 10.0;"
    ```

  - **cust_odbc_bulk**: same as above, however using ODBC bulk operations. (WARNING: Can be slower than the normal ODBC connection - at least for MS SQL Server).

- –**bufferSize, -B:** [default: 128]: The size of the ingestion buffer that is used for the ingest. This determines the number of rows that are transmitted at one time to the database server. Limits by the database system exist and will be checked by AsciiIngest if you specify a larger than supported buffer. Performance can vary alot on this parameter.
- –**outputFreq, -F:** [default: 10000]: Number of rows that are ingested, before a status information and the time needed to ingest the specified amount of rows is printed on the command line.
- –**dbase, -D:** Name of the database/schema where the data is ingested into
- –**table, -T:** Name of the table where the data is ingested into
- –**comment, -C:** [default: #]: The string with which comments in the data file are marked and thus skipped. WARNING: This does not specify the comment string to determine comments in the structure file!
- –**socket, -S:** Socked used to connect to the database
- –**user, -U:** User name
- –**pwd, -P:** Password
- –**port, -O:** [default: 3306 (mysql)]: The port through which to connect to the database. MySQL uses 3306, MS SQL Server uses 1433.
- –**host, -H:** [default: localhost]: Address to the database server
- –**path, -p:** Path to the database file (only used for SQLite3)
- –**disableKeys:** [default: 0]: Disable all keys/indexes on the server side before ingesting the data. This can take some time to complete. WARNING: Disabling keys on large data sets can take a long time to regenerate. WARNING: Not disabling keys on the table can slow down ingestion of data substantially.
- –**enableKeys:** [default: 0]: Reenable all the keys/indexes on the server after ingestion fininshed. WARNING: This operation can take a long time.
- –**userVal, -V:** [default: 1]: If information is read from a header section in the ASCII file (as specified in the structure file), ask the user first if the data has been properly extracted (1 = TRUE, 0 = FALSE). Also any other assumption that the code makes will be first checked with the user.
- –**greedyDelim, -G:** [default: 0]: Should multiple equal delimiters be merged into one (1 = TRUE, greedy read) or not (0 = FALSE)

# Part IV
# Structure Files

Structure files determine how the ASCII data file is read and mapped onto the database. The follwing gives an example of the structure of a structure file:

**Algorithm 1** Example structure file

```
numCols
numHeaderRows
#numCol DType    PrefixLen        DataLen Separator       SchemaColName    AssertAndConvList
D1       CHAR    0                10       ','            haloName         ASRT_ISNOTNULL, CONV_CAPITALIZE,CONV_TRIM
D2       INT4    1                40       ','            numParticles     ASRT_ISNOTNULL, ASRT_ISNOTNEGATIVE, CONV_MULTIPLY(D2)
#D2      INT4    1                40       ','            numParticles     ASRT_ISNOTNULL, ASRT_ISNOTNEGATIVE, CONV_MULTIPLY(D2; 12.0)
#These are examples of header items
#numCol DType    LineNum:offset  DataLen Separator       SchemaColName    AssertList
H3  INT4    0    0    'Redshift:'  redshift    ASRT_ISNOTNULL,CONV_FLOOR
H4  REAL4    10:5   1    ''   expFact ASRT_ISNOTNULL,CONV_FLOOR,CONV_MULTIPLY(3.1415)
#These are examples of constant items
C5  REAL4    0    0    '3.14156'    pi   ASRT_ISNOTNULL
C6  INT 4    0    0    '2' someMagicNumber
```

- Comments are marked with '#'.
- The first line should state the number of columns to be read
- The second line states the number of rows in the ASCII file that represent the header

Each row in the structure file that follows represents a column in the ASCII file. In each data row, the following information that is delimited by whitespaces need to be supplied:

## Data columns:

- numCol: The number of this column. Data columns from the ASCII file should start with a 'D', header colums with 'H' and constant items with 'C'.
- DType: The data type of this column in the ASCII file. Supported are: CHAR, INT2 (integer with 2 bytes, 16 bits), INT4 (integer with 4 bytes, 32 bits), INT8 (integer with 8 bytes, 64 bits), REAL4 (float, 32 bits), REAL8 (double, 64 bits). All datatypes are signed. Unsigned data types are NOT YET supported.
- PrefixLen: If separator is not specified: The number of characters to be skiped, before the data field begins.
- DataLen: If separator is not specified: The number of characters to be read for the data field.
- Separator: If a separator is specified, the field delimited by this separator is read, starting from the position of the last column in the file. Separators have to be specified between singe quotes '. If an empty separator is defined (i.e. ''), then PrefixLen and DataLen will be used to read the data field.
- SchemaColName: Name of the column in the database table on the server.
- AssertAndConvList: A comma delimited list of asserter and converter functions. Assertion functions are marked by ASRT_, Converters by CONV_. Converters can be functions, where the function parameters are stated in parenthesis. Function parameters can either by data columns (i.e. any numCol except ones own) or numerical values. If a Converter supports multiple function parameters, they are delimited by ';'. The asserters and converters are applied from left to right.

## Header columns:

Any numCol that starts with a 'H' is a header column. I.e. information from the header of the ASCII data file is read. Header rows are differently structured:

- numCol: as in data column, only starting with an 'H'
- DType: as in data column
- LineNum:offset: If separator is not specified: Number of the line where to read the information from (1 is the first line), offset marks the character from the left where to start reading
- DataLen: If separator is not specified: The number of characters to be read for the data field
- Separator: Separator in a header row can be a word/string that exists in the header and the data delimited by whitespaces right of that string will be read as a data field. Given between two quotes.
- SchemaColName: as in data column
- AssertAndConvList: as in data column

### Constant columns:

Constant columns are columns that hold the same value (or using Converters are results of these):

- numCol: as in data column, only starting with an 'C'
- DType: as in data column
- PrefixLen: this field is ignored
- DataLen: this field is ignored
- Separator: the constant value between two quotes
- SchemaColName: as in data column
- AssertAndConvList: as in data column

### Storage columns:

Storage columns are constant columns that safe their value into memory for further use after evaluation of all Converters. This way, a calculated value can be taken over to the next row and further modified (an evident example of this is incrementing a variable from row to row by one) (or using Converters are results of these):

- numCol: as in data column, only starting with an 'C'
- DType: as in data column
- PrefixLen: this field is ignored
- DataLen: this field is ignored
- Separator: the constant value between two quotes
- SchemaColName: as in data column
- AssertAndConvList: as in data column

# Part V
# Asserters

This is a list of implemented assertion functions:

- ASRT_ISNEGATIVE
- ASRT_ISNOTINF
- ASRT_ISNOTNAN
- ASRT_ISPOSITIVE

# Part VI
# Converters

This is a list of implemented convertion functions:

- CONV_ADD (one param)
- CONV_DIVIDE (one param)
- CONV_FLOOR (no param)
- CONV_MADD: multiply and add: first parameter will be multiplied and second is then added
- CONV_MULTIPLY (one param)
- CONV_SUBTRACT (one param)
- CONV_IFTHENELSE: if "first parameter == 1" then "second parameter" else "third parameter"
- CONV_ISEQ: (two params): tests first param == second param (1 if true, 0 if false)
- CONV_ISGE: (two params): tests first param >= second param (1 if true, 0 if false)
- CONV_ISGT: (two params): tests first param > second param (1 if true, 0 if false)
- CONV_ISLE: (two params): tests first param <= second param (1 if true, 0 if false)
- CONV_ISLT: (two params): tests first param < second param (1 if true, 0 if false)
- CONV_ISNE: (two params): tests first param != second param (1 if true, 0 if false)