**Epipog (v1.02 , Oct. 29, 2016)**

***Prelude***

EpiPog is an “open source” program for developing design models for NoSQL databases. It’s target audience includes both academic researchers/students as well as those looking to deploy a NoSQL database for commercial uses.

***License***

Licensed under the create commons license: CC-BY

**Initial Release**

This initial release v1.02 was developed as a demonstration of the application of modern design and programming paradigms to build NoSQL databases that can be both lightweight and highly-scalable and adaptable. It is written entirely in Java and can be run on a wide array of platforms. Its light-weightiness makes it well suitable for mobile processing platforms such as phones and tablets.

It incorporates an OOP design methodology, using abstraction and interfaces, and was developed under a test driven development process.

***Compiling and creating Jar executable file***

To produce an executable file, compile all the \*.java files within the same directory. The application can now be run within the same directory (that now contains the compiled \*.class files) by issuing ‘java epipog’:

* javac \*.java # compiles into \*.class files
* java epipog # will run in the same directory

To make the compiled version portable, you need to create a jar file, as follows:

* jar cfe epipog.jar epipog \*.class # archives into jar file
* java –jar epipog.jar # can run anywhere now

***Command Interface***

Usage: epipog <options>

-i inputfile # import input file

-s field(s) # select fields ( use ‘\*’ for all)

-o field(s) # order by fields

-d datastore # type of data store (binary,psv,csv,json), default: binary

-F format # format of input file (psv,csv,tsv), default: csv

-C collection # name of the data collection

-S schema # the schema (key,type pairs)

-t storage # storage type (single,multi), default: single

-I index # index type (linked,binary), default: linked

-P primary # primary keys for indexing

-O sort # sorting algorithm (insert,quick), default: insert

-n # no header in input file, used retained/specified schema

-f filter # filter (where clause)

-c cache # size of cache [STUBBED]

-V # vacuum (remove deleted items) from collection [STUBBED]

**Importing a Dataset**

Version v1.02 can import a variety of datasets:

Character Separated Values (SV):

CSV ( comma separated values)  
 PSV ( pipe separated values)  
 TSV ( tab separated values)

Datasets are imported using the **-i** option. By default, the input format is assumed to be CSV. Alternate input formats can be specified with the **–F** option.

Example: Import a CSV file

* ***java epipg –i input.csv***

Example: Import a TSV file

* ***java epipog –I input.tsv –F tsv***

**Setting a Data Store Representation**

By default, imported data sets are stored in binary fixed record data representation (i.e., RDBMS), which is written to the default collection in the temp directory ( /tmp).

The –d option is used to specify the data store representation, when not using the default.

Version v1.02 supports the following data store representations:

Character Separated Values (SV):

CSV ( comma separated values)  
 PSV ( pipe separated values)

Document Oriented

JSON (Javascript Object Notation)

Binary Fixed Record

Binary (RDBMS)

Example: Import a dataset and store in a CSV data store:

* ***java epipog –i input.txt –d csv***

Example: Import a dataset and store in a JSON data store:

* **java epipog –i input.txt –d json**

**Setting a Named Collection**

By default, all data is written to a single collection in the temporary directory (i.e., /tmp) under the name ‘tmp’. The **–C** option is used to specify a named collection.

Example: import a first dataset to the collection cars and a second dataset to the collection sales

* ***java epipog –i cars.txt –C cars***
* ***java epipog –i sales.txt –C sales***

**CSV Input files with and without headings**

When importing a CSV data set, it is assumed that the first line is a heading. Use the –n option for CSV (and PSV/TSV) datasets that do not have a heading on the first line.

Example: Import a csv file without a heading

* ***java epipog –i input.csv –F csv -n***

**Schema**

While version v1.02 does not support a fully schema-less data store, it does support dynamic schemas. Unlike a traditional RDBMS database, a schema does not need to be predefined. Instead, on the very first import you can specify the schema with the –S option. The schema will then be retained and does not need to be re-specified on subsequent imports of data sets. Schemas are specified in name:type sequence separated by commas:

-S field1:string16,field:integer

The following types are supported:

string16 (16 byte string)  
 string32 (32 byte string)  
 string64 (64 byte string)  
 string128 (128 byte string)  
 char (1 byte character)  
 short (16-bit integer)  
 integer (32-bit integer)  
 long (64-bit integer)  
 float (32-bit float)  
 double (64-bit float)  
 date (date string in format: yyyy-mm-dd)  
 time (time string in format: hh:ss)

*Note: Only ANSI strings are supported in version v1.02.*

Example: Import a first CSV dataset with the columns as strings for country, state and city, and the import a second CSV data set with the same columns.

* ***java epipog –i mexico.csv-Scountry:string64,state:string64,city:string64***
* ***java epipog –i canada.csv***

A schema can also be dynamically extended when importing a subsequent data set that has extended fields. In the above example, we could import a 3rd dataset that has a fourth column for postal.

* ***java epipog –i usa.csv -Scountry:string64,state:string64,city:string64,postal:integer***

**Primary Key and Indexing**

By default, there are no primary keys. A primary key can be specified using the **–P** option for a single field (e.g., column) or a combination of fields. The primary key information is then retained and does not need to be re-specified on subsequent imports.

When importing, an index will be built for the primary key (or combination). Duplicates matching the primary key are eliminated. The pre-existing entry is marked as dirty and the new entry is inserted to replace it (ie., update on duplicate). For example, if the primary key is country and state, then the second data entry below will replace the first entry:

Country,State,Pop  
 United States,Oregon,1500000  
 United States,Oregon,2000000

Example: Import a CSV file and index for primary key combination country and state.

* ***java epipog –i input.txt –S country:string32,state:string32,pop:integer –P country,state***

By default, a linked list is used as the indexing method. The **–I** option is used to specify alternate indexing methods.

*Note: While –I binary option is accepted for a binary tree index, it was not implemented in v1.02*

*Note: The linked list index uses the java hash() function to hash the index strings. Collisions though are not handled in v1.02.*

**Select**

The **–s** option is used to select one or more (or all) fields in a search from a data store.

Example: Select the country and state from a collection named cities.

* ***java epipog –s country,state –C cities***

Example: select all fields from a collection named cities.

* ***java epipog –s “\*” –C cities***

**Sort (Order By)**

The **–o** option is used to sort the results from a select.

Example: Select the country and state from a collection named cities and alphabetically sort by state.

* **java epipog –s country,state –C cities –o state**

By default, sorting is done using a insertion sort algorithm. The –O option can be used to select alternate sorting algorithms:

-O insert (insertion sort)  
 -O quick (quick sort)

*Note: while the command line syntax supports it, verison v1.02 does not support subgroup sorting (ie., first sort by field1 and then within field1 sort by field2).*

**Filter (Where)**

The –f option is used to specify a filter (where) clause on a select. One or more filters can be specified as field<op>value pairs separated by a comma. The following operators are supported:

= (equal)  
 != (not equal)  
 > (greater than)  
 < (less than)  
 >= (greater than or equal)  
 <= (less than or equal)

Example: Select all entries where the field state is equal to Oregon

* ***select –s “city,postal” –f state=Oregon –C cities***

**Storage**

By default, the collections are stored as a single monolithic file. The –S option can be used to specify other file storage methods:

-S single (single monolithic file)  
 -S multi (multi-files: sharding)

*Note: In version v1.02 only single monolithic file storage is supported.*

***File Structure***

Each component (public class) is contained wholly within its own file and corresponding file name.

Query.java – command line interface  
  
DataStore.java – abstract layer for data stores  
BinaryStore.java - derived class for representing data store as a fixed-length records.  
SVStore.java – abstract layer for representing data stores as character delimited file.  
PSVStore.java – derived class for representing data store as PSV format.   
CSVStore.java – derived class for representing data store as PSV format.  
JSONStore.java – derived class for representing data store as JSON objects.

Storage.java – abstract layer for on-disk storage  
SingleFileStorage.java – derived class for storing collection as a single monolithic file  
MultiFileStorage.java – derived class for storing collection as multiple files (stubbed)

Index.java – abstract layer for indexing.  
LinkedIndex.java – derived layer for representing index as an unsorted linear index.  
BinaryTreeIndex.java – derived layer for representing index as a sorted binary tree (stubbed).

Parse.java - abstract layer for parsing input file and inserting into the datastore  
SVParse.java - abstract layer for parsing a character delimited input file and inserting into datastore  
PSVParse.java - derived layer for parsing a PSV input file and inserting into the datastore  
CSVParse.java - derived layer for parsing a CSV input file and inserting into the datastore  
TSVParse.java - derived layer for parsing a TSV input file and inserting into the datastore

Data.java – abstract layer for returned data types from queries  
DataString.java – abstract layer for returned data string types from queries  
DataString16.java, … DataString128.java – derived layer for returned data types from queries  
DataShort.java,DataInteger.java,DataLong.java - derived layer for returned data types from queries  
DataFloat.java,DataDouble.java – derived layer for returned data types from queries  
DataDate.java,DataTime.java – derived layer for returned date types from queries

Sort.java – abstract layer for sorting results  
InsertSort.java – derived layer for sorting results  
QuickSort.java – derived layer for sorting results

QueryException.java – custom exception handler for the data store layer  
StorageException.java – custom exception handler for the storage layer.

Schema.java – schema handling