EventStoreToolkit API

API Documentation

January 24, 2006

Contents

1 Module ESAddComment

The ESAddComment function allows add additional comments into EventStore.

1.1 Functions

ESAddComment(*args*)

ESAddComment was designed to add comments into EventStore DB. To add your comment you need to provide old/new grade or data version name and old/new timeStamp. The comment can be added either from command line or can be read from ASCII file.
Please note, ESAddComment is a wrapper shell script around addComment.py module which does the work.

2 Module ESBuilder

EventStore builder supports SQLite/MySQL DBs through sqlite and MySQLdb modules, respectively. All available options are declare below and can be viewed by using -help option. It keep track of users command through esdb.history file Log of all SQL queries are saved into esdb.log Compensation SQL queries can be found in esdb.compensate_YYYYMMDD_HHMMSS_PID

2.1 Functions

ESBuilder(*args*)

ESBuilder is a main injection tool. It supports two types of DBs: MySQL and SQLite. The injection can be done for variety of file formats: pds, bin, idxa. For option information and usage please use '-help' option. For option description '–help'. For specific injection types please use '-examples' option.
Please note, ESBuilder is a wrapper shell script around ESBuilder.py module which does the work.

3 Module ESDB2DB

A set of tools to merge one EventStore DB to another

3.1 Functions

db2db(*args*)

A tool to merge one EventStore DB into another. We use the following algorithm:

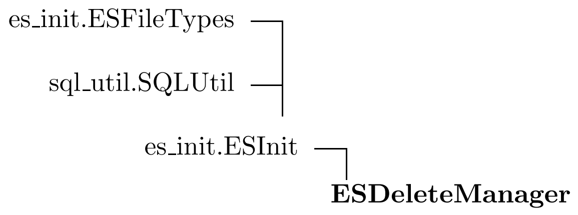
- lock tables of output DB for writing
- loop over available grade,timeStamps in input DB => get graphid
- update Version table in output DB
- loop over files in FileID
- using new graphid and new (fileName,fileType) update KeyFile and Location tables. In the case of location file, change fileIds in its header to new values. In both cases change fileName of index/location files.
- update FileID with new fileName (if any, in the case of data file we don't change its name).
- rename the file.

Please note, we only intend to change a prefix of files and rely on the fact that all files in input DB were properly configured (according with EventStore specs. This routine accepts a list of command line arguments to do the merging. Here an example:
ESDB2DB -dbin EventStore1@/home/vk/sqlite.db -dbout EventStore2@lnx151 -changeprefix /nfs /cdat

4 Module ESDelete

ESDeleteManager handles the case of removing data from EventStore. data files are marked as orphans and associative key/location files are deleted.

4.1 Class ESDeleteManager



ESDeleteManager responsible for deleting entries in EventStore

4.1.1 Methods

__init__(*self*, *db*, *dbType*, *logFile*)

Initialize database pointer, cursor, db type (MySQL or SQLite). Retrieve information about table names from underlying DB.

Overrides: es_init.ESInit.__init__ extit(inherited documentation)

deleteGrade(*self*, *grade*, *timeS*)

'Delete' given grade/timeS in EventStore. Information about such grade is still available but its state is marked as removed and its information moved into OrphanFile table.

doDelete(*self*, *dict*, *table*)

Remove entries in KeyFile and Location tables. When it's done remove physically file from the system.

formDict(*self*, *tup*)

Form a dictionary dict[id]=runList where id is file id

removeElementFromDict(*self*, *dict*, *minR*, *maxR*)

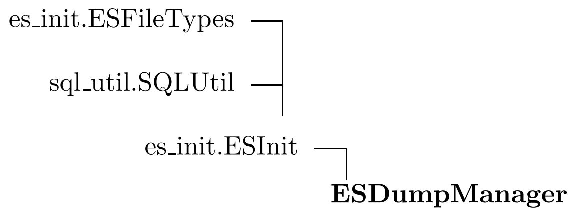
Remove element from dict[id]=runList

Inherited from ESFileTypes: allow, allowToInject, esFileTypes, esInjectionFileTypes, isDatType, isKeyType, isLocType
Inherited from SQLUtil: close, commit, createTables, dropTable, endTxn, fetchAll, fetchOne, findFileForRun, getAllParents, getIsolationLevel, getLastId, getTableNames, getTables, getTableSchema, idx, lockTables, makeESQuery, printDBContent, printESInfo, printRuns, rollback, setCommitFlag, setIsolationLevel, setVerboseLevel, showDepend, startTxn, unlockTables, updateDBAndLog, updateLog, writeToLog

5 Module ESDump

ESDumpManager retrieve information from EventStore tables and provides methods to access it.

5.1 Class ESDumpManager



A dump manager provide utilities to print various information about EventStore DB

5.1.1 Methods

__init__(*self*, *db*, *dbType*, *logFile*)

Initialize database pointer, cursor, db type (MySQL or SQLite). Retrieve information about table names from underlying DB.

Overrides: es_init.ESInit.__init__ extit(inherited documentation)

dumpInfo(*self*, *timeStamp*)

Dump EventStore DB content in user readable format. It uses sql_util module to do the job.

dumpRuns(*self*, *minRun*, *maxRun*)

Prints all available runs in given run range. It uses sql_util module to do the job.

dumpShowDepend(*self*, *childName*)

Show all dependencies for given data version name. It uses sql_util module to do the job.

dumpTable(*self*, *dbTable*)

Prints table content. It uses sql_util module to do the job.

dumpTableSchema(*self*, *tableName*)

Prints table schema. It uses sql_util module to do the job.

findFile(*self*, *searchRun*)

Find file(s) associated with given run. It uses sql_util module to do the job.

Inherited from ESFileTypes: allow, allowToInject, esFileTypes, esInjectionFileTypes, isDatType, isKeyType, isLocType
Inherited from SQLUtil: close, commit, createTables, dropTable, endTxn, fetchAll, fetchOne, findFileForRun, getAllParents, getIsolationLevel, getLastId, getTableNames, getTables, getTableSchema, idx,

lockTables, makeESQuery, printDBContent, printESInfo, printRuns, rollback, setCommitFlag, setIsolation-Level, setVerboseLevel, showDepend, startTxn, unlockTables, updateDBAndLog, updateLog, writeToLog

6 Module ESFixPath

Collection of tools to fix arbitrary file paths wrt EventStore specifications. Please consult https://wiki.lepp.cornell.edu/CleoSW
web page for further reading.

6.1 Functions

ESFixPath(*args*)

Fix paths in EventStoreDB. The CLEOc data path specifications:
/cleo/{detector,simulated}/{event,calibration}/{daq,pass2_version}/123400/123456/{specific_version_path}

formNewPath(*prefix*, *run*, *release*, *svName*, *parentList*, *eventType*=`'event'`)

Form a new path according to CLEOc data path specifications:
/cleo/{detector,simulated}/{event,calibration}/{daq,pass2_version}/123400/123456/{specific_version_path}

getRelease(*relList*, *svName*)

Lookup in release list and find out a match between release name and svName. Return empty string if no match found.

newParentList(*sql*, *run*, *parentList*)

Check every parent if it holds given run

6.2 Variables

Name	Description
RELPATH	**Value:** `'/nfs/solaris2/cleo3/Offline/rel'` *(type=*`str`*)*

7 Module ESGetComment

The ESGetComment function allows get comments from EventStore.

7.1 Functions

ESGetComment(*args*)

ESGetComment was designed to get comments from EventStore DB. In order to use it youm need to provide grade/timeStamp and a date (or date range) of desired comments.
Please note, ESGetComment is a wrapper shell script around getComment.py module which does the work.

8 Module ESManager

Main class which build key/loc files and update EventStore tables in MySQL/SQLite databases.

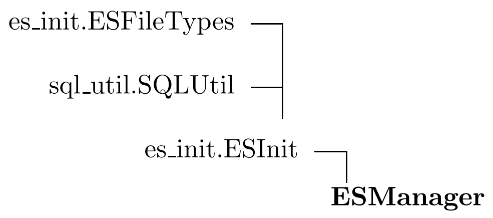
The following update methods are supported:

- update DB using data file or set of files, we support either fileName, file pattern or directory name input
- update DB using IDXA (event list) file

  Data can be entered to EventStore using the following critireas:
- run is not yet present in DB for given grade
- data files shouldn't have overlaping data and sync values
  - data may overlap but have different sync values
  - data may have the same set of sync values but not overlaping proxies

  The same run can be injected only if:
  - data not overlap (e.g. adding D-tagging to pass2)
  - new view is assigned

8.1 Class ESManager



Main class which build key/loc files and update EventStore tables in MySQL/SQLite databases.

8.1.1 Methods

__init__(*self*, *db*, *dbType*, *logFile*, *verbose*=0)

Initialize database pointer, cursor, db type (MySQL or SQLite). Retrieve information about table names from underlying DB.

Overrides: es_init.ESInit.__init__ extit(inherited documentation)

allowStoreToDB(*self*, *iFileList*, *checkFilesInDB*=1)

Check if we can add given files to DB, to be allowed they should contain non-overlaping data among themselves and with EventStore DB. This method also perform data integrity checks on input files. Please consult https://wiki.lepp.cornell.edu/CleoSWIG/bin/view/Main/EventStoreAdministration for more information.

Parameters

`iFileList`: list of files
(type=list)

`checkFilesInDB`: flag
(type=integer (default=1))

Return Value

a tuple of (oFileList,refFileType,isGroup) where oFileList is output file list, refFileType file type of iFileList, isGroup=1 if iFileList can be treated as a group or not, e.g. qcd_hot.pds, 2photon_hot.pds, bhaga_hot.pds and unkown_hot.pds should be treated as a group, rather pass2.pds, post-pass2.pds and dskim.pds should be treated as individual input sources.

checkFileList(*self*, *fileList*)

Check if run/data from provided fileList are unique

Parameters

`fileList`: list of files
(type=list)

Return Value

(rDict,dict,isGroup), two dictionaries: (run: fileList), (file: [runList,pList,svList]) pList list of proxies in a file svList list of sync Values presented in a file and isGroup - a flag which tell how to treat input file list.
(type=tuple)

checkParentsInDB(*self*)

First thing during injection we need to check if provided list of parents is already present in DB, otherwise we need to inject parent's information.

Return Value

none
(type=none)

checkVersionInfo(*self*, *iFileList*)

Verify that versioning information either setup throguh command line interface or read from input file list.

Parameters

`iFileList`: list of files
(type=list)

Return Value

status code
(type=integer)

compareLists(*self*, *list1*, *list2*)

Compare two list for their intersection. If use python 2.4 we use 'set' intersection module, otherwise we count every entry from one list into another.

Parameters

`list1`: list
(type=list)

`list2`: list
(type=list)

Return Value

1 if lists overlap and 0 otherwise. For python 2.4 and above we use set(list1) & set(list2), otherwise loop over list1 and search its entries in list2.
(type=integer)

decodeKeyFile(*self*, *keyFileName*)

Decode content of key file. It just invokes `key_dump.dump` method.

Parameters

`keyFileName`: file name
(type=string)

Return Value

a list of (run,evt,uid,stream)
(type=list)

findParents(*self*, *iFile*)

For given input file we find all parents in DB. The fileType of parents should be the same as fileType of input file.

Parameters

`iFile`: file name
(type=string)

Return Value

a tuple (parentDict,presentParents) where parentDict[keyTuple]=[keyFileList,locFileList,locDataDict] keyTuple is (graphid,run,uniqueId) and lists of key and location files as well as locDataDict[(keyFileName,locFileName)]=dataFileList and presentParents is a flag which indicates if parents were found in DB, regardless of input fileType.
(type=list)

generateLocAndKeyFilesFrom(*self*, *iFileList*)

This routine generates location and key files from given list of intput sources (so far iFileList is a list of PDS files). The input sources are analyzed and main ESDB or stand-alone DB is quiering for their parents. Please note that we search for parents of the same data type as input source. For example:

- there is no parents for pass2 injection since pass2 parents are different data type (binary)
- there are parents for post-pass, it is pass2.

If parents are found (data files, key and location files) their key/loc. files can be used (instead input data files) for building output location and key files. Otherwise input data files are used. The common API uses 'readers' to walk through files. Currenly we implemented two types of readers, pds reader (see PDSFileReader) and key/loc. files reader (see KeyLocFilesReader class). The reader access data information and navigates through data.

In order to build output key/loc. files we auto-probe the input sources. If parent's key file is found (and it's only one) its sync. values compared to input sync. values and if differ input sources are declared as a skim (input sources are subset of parents, e.g. DSkim). In the case of a skim we use its list of sync. values to drive the building process, otherwise parent's sync. list is used. In later case if input source doesn't have sync. value which is present in parent sync. values a fake entries are inserted into output location files. In this case the output key file is identical to parent one. In the case of skim, the output key file contains a subset of sync. values wrt parent one.

To build output location header we combine proxies from parents and input sources if we're dealing with a skim case, e.g. DSkim injection. Otherwise we only use input source proxies, e.g. pass2 injection (no parents of the same data type are found) or post-pass2 injection (in this case parents are found (pass2), but sync. values of post-pass2 and pass2 are the same).

Parameters

`iFileList`: list of input sources (pds files), e.g. qcd_hot.pds, 2photon_hot.pds
(type=list)

Return Value

we return three dictionaries

- dict[(run,uid)]=[(fileId,fileName,typeId,view)] this is a list of location files which include newly created loc. file and parents loc. files
- dict[(run,uid)]=[(fileId,fileName,typeId)] this is a list of key files which include either newly created key file (in the case of skim) or key file of the parent (since it's cover the same sync. values) and parent's key files for different views, e.g. qcd view.
- dict[fileId]=fileName, map of fileId vs data file names which include input sources and their parents of the same type, e.g. when we inject post-pass2, we return post-pass2 pds file and its parent's pass2 files.

Those dictionaries are used to update FileID, KeyFile, Location and RunUID tables in `updateDBUsingGroupList` method.
(type=tuple (oFileDict,oKeyDict,oLocDict))

genFile(*self*, *fileName*, *fileId*, *loc*=`0`)

Generate key/location files names. It uses genFileName to do a job.

Parameters

`fileName`: name of the file
(type=string)

`fileId`: file id
(type=integer)

`loc`: flag to generate location file.
(type=integer)

Return Value

file name generated by `genFileName` and files are generated either by `file_util.build_location` for location files or `file_util.build_key` for key files.
(type=string)

genFileName(*self*, *fileName*, *fileId*, *buildType*)

Generate a unique key/location file name. File name is formed by base of data file name, plus an unique file id assgined by FileID table during file id allocation.

Parameters

`fileName`: name of the data file, e.g. run111111.bin or myName.pds. File type is analyzed by `file_util.fileType`.
(type=string)

`fileId`: file id
(type=integer)

`buildType`: we may be ask to construct either 'location' or 'key' file names
(type=string)

Return Value

file name in the following form: run-runNumber-esdb-fileId.extension.
(type=string)

getFileID(*self*, *fileName*)

Lookup in FileID and either return fileID or 0 if file is not present. We use `fetchOne` method to make a query.

Parameters

`fileName`: name of the file
(type=string)

Return Value

file id or 0
(type=string (we use type of returning query))

getFileInfo(*self*, *iFile*)

Form a lists of runs, syncValues and proxies from given file.

Parameters

`iFile`: file name
(type=string)

Return Value

(runList,svList,proxyList), list of runs, sv's and proxies presented in a file
(type=tuple)

getGraphIds(*self*, *all=*`''`)

Return a list of parent graph id's. Can perform nested lookup if all parameter is specified.

Parameters

`all`: optional parameter to check parents by using `getAllParents`
(type=integer or string)

Return Value

list @return list of graph id's

getIds(*self*, *howMany*)

High-level function to retieve new set of unique ids. It uses getNextId to do a job.

Parameters

`howMany`: generate 'howMany' new Id's in FileID table
(type=integer)

Return Value

list of unique id's generated by `getNextId`
(type=list)

getLocAndKeyFromParent(*self*, *fileList*)

Get list of parent pairs (id,fileName) for key/loc. files from provided list of files.

- key file case:
  - we only lookup immediate parents
- loc file case:
  - we lookup all parents in dependency tree

Parameters

`fileList`: list of files
(type=list)

Return Value

a tuple of three lists: (locFileList,keyFileList,viewFileList) where viewFileList is a file list of key files with non-all views.
(type=tuple)

getMaxId(*self*)

Get MAX fileId from master ESDB. First we check MaxMasterID table for any recorded maxId. If it presents there we compare <db>@<host>:<port>:<socket> string to current one and raise exception if they're not match. If MaxMasterID is empty we retrieved information from master ESDB and record it to MaxMasterID.

Return Value

maximum file id from the master db.
(type=long)

getNextId(*self*, *howMany*)

Allocate a new file id in FileID table. We rely on autoincrement feature of underlying DB.

Parameters

`howMany`: generate 'howMany' new Id's in FileID table
(type=integer)

Return Value

list of unique id's generated by autoincrment while executing 'INSERT INTO FileID(fileName,typeId) VALUES(NULL,0)' query. Id's are obtained by using `getLastId` method. Query is logged by `updateDBAndLog` method.
(type=list)

getRunUidListFromFiles(*self*, *fileList*)

Scan all files and return runUidList from them.

Parameters

`fileList`: list of files
(type=list)

Return Value

list of run and uids presented in input file list
(type=list)

getVersionInfo(*self*, *iFileList*)

Parse input file list and try to get version information from 'beginrun' stream according to specifications, please see https://wiki.lepp.cornell.edu/CleoSWIG/bin/view/Main/BeginrunVersioning The real job is done by `pds_dump.decodeVersionInfo` method.

Parameters

`iFileList`: list of files
(type=list)

Return Value

status code
(type=integer)

openDBs(*self*)

Open EventStore tables. If necessary use `createTables` routine to create those. Everything is wrapped in transaction: `startTxn` and `endTxn`.

printAllDB(*self*)

Prints content of all EventStore table by using `printDBContent`

Return Value

none
(type=none)

printESDBContent(*self*, *dbid*)

Prints content of EventStore based on its id.

Parameters

`dbid`: DB id
(type=integer)

Return Value

none
(type=none)

printMsg(*self*, *fileList*, *Message*, *level*=`'ERROR'`)

Form a general report message with outline of current DB shapshot.

Parameters

`fileList`: list of files
(type=list)

`Message`: message which passed to here
(type=string)

`level`: level of severity
(type=string)

Return Value

none
(type=none)

queryFileIDTable(*self*, *tup*)

Retrieve file names and type ids from FileID table.

Parameters

`tup`: list of file id's
(type=list)

Return Value

list of triplets (fileId, fileName, typeId).
(type=list)

readIDXAFile(*self*, *fileName*)

Read ASCII idxa file and return syncValue and runUid lists.

Parameters

`fileName`: name of the file
(type=string)

Return Value

a tuple of two lists: svList and runUidList
(type=tuple)

requestDataFromDB(*self*, *query*, *whatToRetrieve*=`'all'`)

Send query to the master DB, the master DB may be specified by user, otherwise use EventStore@lnx151.

Parameters

`query:`	SQL query *(type=string)*
`whatToRetrieve:`	a keyword to distinguish what to retrieve, e.g. 'all' or 'one'. *(type=string)*

Return Value

tuple We use either `es_init.requestDataFromDB` methods for quering.
(type=none)

requestDataFromUserDB(*self*, *query*, *whatToRetrieve*=`'all'`)

We may request data either from user or master DBs. Return tuple for given query.

Parameters

`query:`	SQL query *(type=string)*
`whatToRetrieve:`	a keyword to distinguish what to retrieve, e.g. 'all' or 'one'. *(type=string)*

Return Value

tuple We use either `fetchOne` or `fetchAll` methods for quering.
(type=none)

setDBHost(*self*, *dbHost*)

Set EventStore host name.

Parameters

`dbHost:`	user DB hostname *(type=string)*

Return Value

none
(type=none)

setDBName(*self*, *dbName*)

Set EventStore name.

Parameters

`dbName:`	user DB name *(type=string)*

Return Value

none
(type=none)

setDBPort(*self*, *dbPort*)

Set EventStore port

Parameters

`dbPort`: user DB name
(type=string)

Return Value

none
(type=none)

setDBSocket(*self*, *dbSocket*)

Set EventStore socket

Parameters

`dbSocket`: user DB socket
(type=string)

Return Value

none
(type=none)

setGenerateDB(*self*, *newdb*)

Set a flag to generate new database.

Parameters

`newdb`: flag to inform ESManager to generate new DB
(type=string or integer)

Return Value

none
(type=none)

setGrade(*self*, *grade*)

Set grade name

Parameters

`grade`: name of the grade, e.g. 'physics'
(type=string)

Return Value

none
(type=none)

setMasterDB(*self*, *dbName*, *dbHost*, *dbPort=*`''`, *dbSocket=*`''`)

Set EventStore and host names of the underlying master ESDB. If toolkit is runnig in Cornell domain (lnx.cornell.edu) we use default lnx151.lns.cornell.edu as a master DB.

Parameters

`dbName:` user DB hostname
(type=string)

`dbHost:` user DB hostname
(type=string)

`dbPort:` db port, e.g. 3306 is default for MySQL
(type=integer)

`dbSocket:` socket file, e.g /var/log/mysql
(type=string)

Return Value

none
(type=none)

setMaxRun(*self*, *maxR*)

Set upper bound on run range

Parameters

`maxR:` maximum run number of run range
(type=integer)

Return Value

none
(type=none)

setMinRun(*self*, *minR*)

Set lower bound on run range

Parameters

`minR:` minimum run number of run range
(type=integer)

Return Value

none
(type=none)

setNoSkimFlag(*self*, *noskim*)

Set no-skim flag, i.e. inform ESManager to treat input sources as is

Parameters

`noskim:` inform ESManager to use input source as is
(type=string or integer)

Return Value

none
(type=none)

setOutputDir(*self*, *oDir*)

Set location of output directory which would be used to write out key/location files

Parameters

`oDir:` name of output directory
(type=string)

Return Value

none
(type=none)

setParents(*self*, *parents*)

Set list of parents

Parameters

`parents:` list of parents
(type=list)

Return Value

none
(type=none)

setReadDuplicatesSource(*self*, *dupRead*)

Set file name which will be used to resolve data overlap conflicts

Parameters

`dupRead:` file name
(type=string)

Return Value

none
(type=none)

setSkimFlag(*self*, *skim*)

Set skim flag, i.e. inform ESManager to treat input sources as a skim

Parameters

`skim:` inform ESManager that input source is a skim
(type=string or integer)

Return Value

none
(type=none)

setSVName(*self*, *svName*)

Set data version name

Parameters

`svName:` data version name, a.k.a specific version name
(type=string)

Return Value

none
(type=none)

setTimeStamp(*self*, *timeS*)

Set time stamp

Parameters

`timeS`: time stamp to be used, e.g. 20090909
(type=string or integer)

Return Value

none if timeS==-1 we use `gen_util.dayAhead` to set up a day ahead, otherwise we assign self.timeS=timeS
(type=none)

setView(*self*, *view*)

Set view name

Parameters

`view`: view name, e.g. 'qcd'
(type=string)

Return Value

none
(type=none)

uniqueList(*self*, *iList*)

Eliminates duplicates from provided list and return back unique list

updateDB(*self*, *genMode*, *iFileList*, *oHSMDir=''*)

Main method to update EventStore DB. Based on provided file list it decide how to inject data into EventStore. Database is open by using `openDBs`, all input files are checked by `allowStoreToDB`, then we update Version table using `updateVersion`. Finally, based on file type of input files we either use `updateDBFromIDXA`, `updateDBUsingFileList` or `updateDBUsingGroupList` to do actual job.

Parameters

`genMode`: flag to generate files
(type=integer)

`iFileList`: list of files
(type=list)

`oHSMDir`: optional location of HSM directory where files copies will go
(type=string)

Return Value

status code
(type=integer)

updateDBFromIDXA(*self*, *fileName*)

Inject information from idxa file into EventStore. In this case we only create a new index (key) file.

Parameters

`fileName`: name of the file
(type=string)

Return Value

status code
(type=integer)

updateDBUsingFileList(*self*, *generationMode*, *fileList*, *oHSMDir*=`''`)

Inject sequentially files into EventStore. This method is used to inject raw data files to ESDB. Key and location files are generated by using `build_binary_key_loc.build_binKeyAndLoc` in case of binary input files or `genFile`. All tables in DB are updated by using: `updateKeyFile`, `updateRunUID` and `updateLocation`.

Parameters

`generationMode`: flag to generate key/loc. files
(type=integer)

`fileList`: list of files
(type=list)

`oHSMDir`: optional file name of binary file
(type=string)

Return Value

status code
(type=integer)

updateDBUsingGroupList(*self*, *fileList*, *oHSMDir*=`''`)

Inject provided files into EventStore as a logical group, e.g. qcd_hot_runX, 2photon_hot_runX, etc. Key and location files are generated by using `generateLocAndKeyFilesFrom`. All tables in DB are updated by using: `updateKeyFile`, `updateRunUID` and `updateLocation`.

Parameters

`fileList`: list of files
(type=list)

`oHSMDir`: optional HSM location where files copies will go
(type=string)

Return Value

status code
(type=integer)

updateFileID(*self*, *id*, *name*, *typeId*)

Update FileID table entry for given id, name and typeId. We use `startTxn` and `endTxn` as transaction wrappers and `updateDBAndLog` for table update and loging.

Parameters

`id:`	file id *(type=long)*
`name:`	file name *(type=string)*
`typeId:`	type Id *(type=integer)*

Return Value

none
(type=none)

updateFileType(*self*, *fileName*)

Update FileType table. File type is determined by `file_util.fileType`. We use `startTxn` and `endTxn` as transaction wrappers and `updateDBAndLog` for table update and loging.

Parameters

`fileName:`	name of the file *(type=string)*

Return Value

newly allocated or obtained file type id
(type=integer)

updateGraphPath(*self*, *graphid*, *svid*)

Update GraphPath table. We use `startTxn` and `endTxn` as transaction wrappers and `updateDBAndLog` for table update and loging.

Parameters

`graphid:`	graph id *(type=integer)*
`svid:`	specific version id *(type=integer)*

Return Value

none
(type=none)

updateKeyFile(*self*, *graphid*, *view*, *run*, *uid*, *key_id*)

Update KeyFile table. We use `startTxn` and `endTxn` as transaction wrappers and `updateDBAndLog` for table update and loging.

Parameters

`graphid:`	graph id *(type=integer)*
`view:`	view, e.g. 'qcd' *(type=string)*
`run:`	run number *(type=integer)*
`uid:`	unique id *(type=long)*
`key_id:`	key file id *(type=long)*

Return Value

none
(type=none)

updateLocation(*self*, *graphid*, *run*, *uid*, *loc_id*)

Update Location table. We use `startTxn` and `endTxn` as transaction wrappers and `updateDBAndLog` for table update and loging.

Parameters

`graphid:`	graph id *(type=integer)*
`run:`	run number *(type=integer)*
`uid:`	unique id *(type=long)*
`loc_id:`	location file id *(type=long)*

Return Value

none
(type=none)

updatePathDepend(*self*, *svid*)

Update PathDepend table.

Parameters

`svid:`	specific version id *(type=integer)*

Return Value

none
(type=none)

updateRunUID(*self*, *run*, *uid*)

Update RunID table. We use `startTxn` and `endTxn` as transaction wrappers and `updateDBAndLog` for table update and loging.

Parameters

`run`: run number
(type=integer)

`uid`: unique id
(type=long)

Return Value

none
(type=none)

updateSpecificVersion(*self*, *svName*, *svid*)

Update SpecificVersion table. We use `startTxn` and `endTxn` as transaction wrappers and `updateDBAndLog` for table update and loging.

Parameters

`svName`: data version name, a.k.a specific version name
(type=string)

`svid`: specific version id
(type=integer)

Return Value

none
(type=none)

updateVersion(*self*)

Update Version table. The procedure is to check if processed run fall in existing run range for given grade/timeStamp. Otherwise form new graphid, update GraphPath,SpecificVersion,PathDepend table(s)

Return Value

if nothing wrong we return graphid which always positive or self.error which is always 0.
(type=none)

Inherited from ESFileTypes: allow, allowToInject, esFileTypes, esInjectionFileTypes, isDatType, isKeyType, isLocType

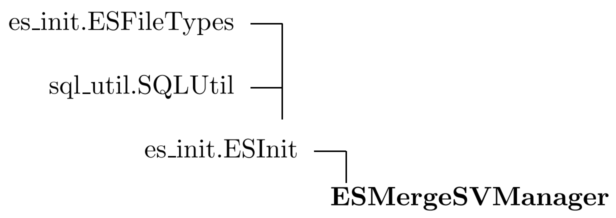
Inherited from SQLUtil: close, commit, createTables, dropTable, endTxn, fetchAll, fetchOne, findFileForRun, getAllParents, getIsolationLevel, getLastId, getTableNames, getTables, getTableSchema, idx, lockTables, makeESQuery, printDBContent, printESInfo, printRuns, rollback, setCommitFlag, setIsolationLevel, setVerboseLevel, showDepend, startTxn, unlockTables, updateDBAndLog, updateLog, writeToLog

9 Module ESMergeSVManager

This tool designed to help merge multiple data version names into new graphid Algorithm: upon request to merge multiple svName's:

- resolve their svId's
- add entry into GraphPath newGraphId->svid1, newGraphId->svid2, etc.
- add entry into Version with all run-ranges from gid1, gid2, etc.
- copy entries from KeyFile with gid1->newGraphId, gid2->newGraphId
- copy entries from Location with gid1->newGraphId, gid2->newGraphId

9.1 Class ESMergeSVManager



ESMergeSVManager is in charge of merging multiple svNames into new graphid

9.1.1 Methods

__init__(*self*, *db*, *dbType*, *logFile*)

Initialize database pointer, cursor, db type (MySQL or SQLite). Retrieve information about table names from underlying DB.

Overrides: es_init.ESInit.__init__ extit(inherited documentation)

merge(*self*, *svList*)

Main routine to merge and update all tables using given data version list:

- resolve their svId's
- add entry into GraphPath newGraphId->svid1, newGraphId->svid2, etc.
- add entry into Version with all run-ranges from gid1, gid2, etc.
- copy entries from KeyFile with gid1->newGraphId, gid2->newGraphId
- copy entries from Location with gid1->newGraphId, gid2->newGraphId

setGrade(*self*, *grade*)

Set grade to be used

setTime(*self*, *timeS*)

Set time stamp to be used

updateKeyFile(*self*, *newgid*, *gidList*)

Update KeyFile table

updateLocation(*self*, *newgid*, *gidList*)

Update Location table

updateVersion(*self*, *svList*)

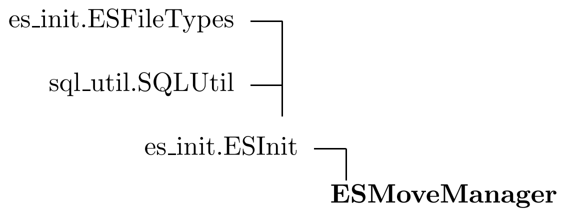
Update Version and GraphPath tables

Inherited from ESFileTypes: allow, allowToInject, esFileTypes, esInjectionFileTypes, isDatType, isKeyType, isLocType
Inherited from SQLUtil: close, commit, createTables, dropTable, endTxn, fetchAll, fetchOne, findFileForRun, getAllParents, getIsolationLevel, getLastId, getTableNames, getTables, getTableSchema, idx, lockTables, makeESQuery, printDBContent, printESInfo, printRuns, rollback, setCommitFlag, setIsolationLevel, setVerboseLevel, showDepend, startTxn, unlockTables, updateDBAndLog, updateLog, writeToLog

10 Module ESMove

ESMoveManager is in charge of moving data in EventStore.

10.1 Class ESMoveManager



ESMoveManager is in charge of moving files in EventStore

10.1.1 Methods

__init__(*self*, *db*, *dbType*, *logFile*)

Initialize database pointer, cursor, db type (MySQL or SQLite). Retrieve information about table names from underlying DB.

Overrides: es_init.ESInit.__init__ extit(inherited documentation)

moveFileInES(*self*, *fileIn*, *fileOut*)

Move a single fileIn to fileOut destination. First update FileID table and then physically copy file to new location.

Parameters

`fileIn:`	file name *(type=string)*
`fileOut:`	file name *(type=string)*

Return Value

(status code, query, compensatingQuery), where query is SQL query how to update FileID table. It's printed out by `moveFilesInES` in the case of failure for debugging purpose.
(type=tuple)

moveFilesInES(*self*, *iFileList*, *fileOut*)

High-level method to move a list of files to fileOut destination. All job is done by using `moveFileInES` method.

Parameters

`iFileList:`	list of files *(type=list)*
`fileOut:`	file name *(type=string)*

Return Value

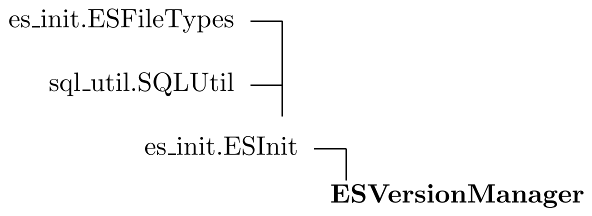
status code
(type=integer)

Inherited from ESFileTypes: allow, allowToInject, esFileTypes, esInjectionFileTypes, isDatType, isKeyType, isLocType
Inherited from SQLUtil: close, commit, createTables, dropTable, endTxn, fetchAll, fetchOne, findFileForRun, getAllParents, getIsolationLevel, getLastId, getTableNames, getTables, getTableSchema, idx, lockTables, makeESQuery, printDBContent, printESInfo, printRuns, rollback, setCommitFlag, setIsolationLevel, setVerboseLevel, showDepend, startTxn, unlockTables, updateDBAndLog, updateLog, writeToLog

11 Module ESVersionManager

ESVersionManager is in charge of moving grades within EventStore

11.1 Class ESVersionManager



ESVersionManager is in charge of moving grades within EventStore

11.1.1 Methods

__init__(*self*, *db*, *dbType*, *logFile*)

Initialize database pointer, cursor, db type (MySQL or SQLite). Retrieve information about table names from underlying DB.

Overrides: es_init.ESInit.__init__ extit(inherited documentation)

checkRunsInDB(*self*, *grade*, *tStamp*)

Check if new list overlap with a list of run-ranges in DB

diffLists(*self*, *list1*, *list2*)

Compare two list of run-ranges. Order lists and look first if for intersection between them, if found show difference between list1 and list2

findLatestTimeStamp(*self*, *iGrade*)

Search for latest timeStamp in Version table, it will assign a day ahead for non existing grade.

formNewRunList(*self*, *iGrade=''*, *iTime=''*)

Form a new run-range list for given grade/timeStamp from badRunList or use goodRunList for make it

formQuery(*self*, *iGrade*, *iTime*)

Form a query for given grade/time to lookup in Version table

moveGrade(*self*, *iGrade*, *oGrade*, *iTime*, *oTime*)

Main method to move a grade in EventStore. It accepts old/new grade and timeStamp. Based on user settings it can make a duplicates, exclude some runs while moving one grade into another.

setExcludeRunList(*self*, *runList*)

Set list of runs to be excluded

setGoodRunList(*self*, *runList*)

Set list of runs to be excluded

setMaxRun(*self*, *maxR*)

Set upper run range bound to be used

setMinRun(*self*, *minR*)

Set lower run range bound to be used

setSVName(*self*, *svName*)

Set data version name to be used

Inherited from ESFileTypes: allow, allowToInject, esFileTypes, esInjectionFileTypes, isDatType, isKeyType, isLocType
Inherited from SQLUtil: close, commit, createTables, dropTable, endTxn, fetchAll, fetchOne, findFileForRun, getAllParents, getIsolationLevel, getLastId, getTableNames, getTables, getTableSchema, idx, lockTables, makeESQuery, printDBContent, printESInfo, printRuns, rollback, setCommitFlag, setIsolationLevel, setVerboseLevel, showDepend, startTxn, unlockTables, updateDBAndLog, updateLog, writeToLog

12 Module binary_dump

A set of utilities to dump content of binary (raw) files

12.1 Functions

dump(*fileName*, *verbose*=`0`)

Dump content of binary file. The verbosity level can be specified

13 Module binary_reader

A set of tools to read the content of binary (raw) files

13.1 Functions

binaryParser(*iFile*, *what*='')

Binary parser scan a file and return a list of runs, uids, proxies and sync. values presented in a file

binaryRunParser(*iFile*, *what*='')

Binary run parser scan a file and return a list of run and sync. values presented in a file

fileInfo(*fileName*)

Helper function to print file binary file content. It uses binaryParser to do a job

printProxiesInAllStreams(*streamProxyList*)

Print list of proxies presented in binary file

14 Module binary_utils

Defines global event data types. See, /nfs/cleo3/Common/src/EventDefs/EventTypes.h

14.1 Functions

beginRunRecordIsMC(*recordType*)

Return a type of begin run record (MC or data)

recordTypeToStreamId(*recordType*)

Convert given record type into stream id

streamIdToName(*id*)

Convert stream id into stream name

15 Module build_binary_key_loc

A set of tools to build key and location files from binary (raw) data file

15.1 Functions

build_binKeyAndLoc(*iBinaryFileName*, *iFileID*, *locFileName*, *keyFileName*, *binFileName*=`0`)

Build simulteneously key and location files from a binary file.

Parameters

`iBinaryFileName:`	input binary file name *(type=string)*
`iFileID:`	input file Id *(type=long)*
`locFileName:`	output location file name *(type=string)*
`keyFileName:`	output key file name *(type=string)*
`binFileName:`	output binary file name (optional). Caliper is used this to write out binary file to HSM. *(type=string)*

16 Module build_binary_location

A set of tools to build binary location file

16.1 Functions

build_binary_location(*iBinaryFileName*, *iFileID*, *iLocationFileName*)

Build binary location file.

Parameters

`iBinaryFileName:`	input binary file name *(type=string)*
`iFileID:`	input file Id *(type=long)*
`iLocationFileName:`	location file name *(type=string)*

Return Value

none
(type=none)

changeFileIdsInLocFile(*locFileName*, *fileIdList*)

Change fileIDs in location header to provided id list

17 Module build_key_from_binary

A set of tools to build key file from binary (raw) data file

17.1 Functions

build_key(*iFile*, *oFile*, *oFileID*)

Build a key file from a binary file

18 Module build_key_from_pds

A set of tools to build key file out of PDS data file

18.1 Functions

build_key(*iFile*, *oFile*, *oFileID*)

Build a key file from a pds file

19 Module build_pds_location

A set of tools to build location file from PDS data file

19.1 Functions

build_pds_location(*iPDSFileName*, *iFileID*, *iLocationFileName*, *allList*=`[]`)

Build location file from PDS data file

buildLocationFileContent(*iPDSFileName*, *iFileID*)

Instead of writing directly to file return information about location file back to the user. It used by old code of `ESManager` when building combined location file out of multiple PDS files

buildLocationHeader(*iPDSFileName*, *iFileID*)

Build a PDS location header, from given pds file name and file id

buildLocationHeaderFromDict(*streamDataKeysDict*, *fileIDList*)

Build a PDS location header, from given components

changeFileIdsInLocFile(*locFileName*, *fileIdList*)

Change fileIDs in location header to provided id list

getStreamDataKeyDictFromPDS(*iPDSFileName*)

Extract from PDS file streamDataKey dictionary

20 Module convert

A simple convertor of MySQL<->SQLite databases for cleo3 EventStore

20.1 Functions

convert(*args*)

convert module allow user to convert EventStore DB from MySQL into SQLite format and vice versa.
Examples:

- SQLite into MySQL
  - convert.py -in sqlite sqlite.db -out mysql lnx151 -esdb EventStoreTMP
- MySQL into SQLite:
  - convert.py -out sqlite sqlite.db -in mysql lnx151 -esdb EventStoreTMP

21 Module createBackupSQL

A helper script which take a latest snapshot of EventStore DB and creates SQL statements how to rollback to that state.

22 Module es_init

Set of helper classes to initialize ES toolkit

22.1 Functions

checkArg(*optValues*)

Check provided list of arguments for leading '-'.

Parameters

`optValues`: list of argument values provided by user
(type=list)

Return Value

exit 1 if given value is not allowed, otherwise return nothing.
(type=none)

checkPythonVersion(*version*)

Check if provided version is greater then current version of python.

Parameters

`version`: python version, e.g. 2.4
(type=string)

Return Value

true or false

connectToMasterDB(*dbName*, *dbHost*, *port*, *socket*, *verbose*=`0`)

Connect to master DB

Parameters

`dbName`: name of underlying DB, e.g. EventStore
(type=string)

`dbHost`: hostname or file name for DB, e.g. lnx151 or sqlite.db
(type=string)

`port`: port number, e.g. 3306 is default for MySQL
(type=integer)

`socket`: file name of the socket
(type=string)

Return Value

db pointer and db type, e.g. "sqlite"
(type=tuple)

decodeHostNameString(*hostName*)

Decode provided string to dbName@dbHost:dbPort:dbSocket

ESDBConnector(*dbHost*, *dbName*, *userName=''*, *userPass=''*, *isolationLevel=''*, *dbPort=''*, *dbSocket=''*)

ESDBConnector connect to given host(MySQL) and/or file(SQLite) using dbName. In the cae of SQLite user may pass isolationLevel:

- default mode, if no isolationLevel is passed will start db with BEGIN
- autocommit mode, isolationLeve='None'
- you may also pass: DEFERRED, IMMEDIATE or EXCLUSIVE levels
  - DEFERRED deffer locks until first access
  - IMMEDIATE implies lock for first BEGIN IMMEDIATE and doesn't allow anything to write to DB, but allow to read
  - EXCLUSIVE will lock entire DB.

We use IMMEDIATE for SQLite and READ COMMITTED isolation level for MySQL. We also turn off autocommit.

Parameters

`dbHost:`	name of the host, lnx151.lns.cornell.edu or db file name /sqlite.db *(type=string)*
`dbName:`	DB name, obsolete in the case of SQLite *(type=string)*
`userName:`	user name *(type=string)*
`userPass:`	password *(type=string)*
`isolationLevel:`	valid only for SQLite, setup isolation level *(type=string)*
`dbPort:`	port number, e.g. default for MysQL is 3306 *(type=integer)*
`dbSocket:`	socket name, e.g. /var/log/mysql= *(type=string)*

Return Value

return object to underlying DB and its type, e.g. "sqlite" or "mysql"
(type=tuple (db, dbType))

ESExamples()

Contain a list of usefull examples how to add/modify data to EventStore.

Return Value

none
(type=none)

ESInput(*userCommand*, *outputLog*, *dbType*)

Write userCommand information into EventStore db. userCommand is a command with their option invoked by user.

Parameters

`userCommand:`	command invoked by user *(type=string)*
`outputLog:`	open file descriptor *(type=file descriptor)*
`dbType:`	type of underlying DB, e.g. ”sqlite” or ”mysql” *(type=string)*

Return Value

none
(type=none)

ESOptions(*userCommand*, *optList*, *usage=*`''`, *usageDescription=*`''`, *examples=*`''`)

Analyse common options of EventStore toolkit: [-help] [–help] [-examples] [-profile] [-verbose] [-historyfile <filename>] [-db <name@host:port:socket or fileName>] [-user <username> -password <password>] [-logFile </path/filename or ’stdout’ or ’stderr’>]

Return Value

oList=[dbName,dbHost,userName,userPass,dbPort,dbSocket,histFile,logFile,logDir, verbose,profile,userCommand] dictOpt[option]=value
(type=tuple (oList,dictOpt))

ESOutput(*status*, *userCommand*, *historyFile*, *outputLog*, *globalLog*)

Write out final information about job status to EventStore db log.

Parameters

`status:`	status code of injection *(type=integer)*
`userCommand:`	command invoked by user *(type=string)*
`historyFile:`	file name *(type=string)*
`outputLog:`	open file descriptor *(type=file descriptor)*
`globalLog:`	open file descriptor *(type=file descriptor)*

Return Value

status code
(type=integer)

ESOutputLog(*logFile*)

Setup EventStore db log. It is either stdout, stderr or file based.

Parameters

`logFile`: name of the log file
(type=string)

Return Value

outputLog is a open file descriptor to the output log file globalLog is a open file descriptor for global EventStore log, e.g. multiple jobs can write to global log file and it's local own log file
(type=tuple (outputLog,globalLog))

helpMsg(*tool*, *localOpt*)

Form usage message. Defines a list of common options for every tool used in EventStore toolkit. Among them '-help', '-esdb', etc.

Parameters

`localOpt`: list of options
(type=list)

Return Value

help message how to use toolkit
(type=string)

requestDataFromDB(*dbName*, *dbHost*, *port*, *socket*, *query*, *whatToRetrieve*=`'all'`, *verbose*=`1`)

Send query to specified DB. Return tuple for given query.

Parameters

`dbName:`	name of underlying DB, e.g. EventStore *(type=string)*
`dbHost:`	hostname or file name for DB, e.g. lnx151 or sqlite.db *(type=string)*
`port:`	port number, e.g. 3306 is default for MySQL *(type=integer)*
`socket:`	file name of the socket *(type=string)*
`query:`	SQL query *(type=string)*
`whatToRetrieve:`	how to retrieve query, a single row or all matched rows *(type=string)*
`verbose:`	verbosity flag *(type=integer (default is yes))*

Return Value

result's tuple associated with requested query
(type=tuple)

22.2 Class ESFileTypes

Known Subclasses: ESInit

Base class which defines file types accepted by EventStore

22.2.1 Methods

__init__(*self*)

Initialize file types supported in EventStore. Write now all of them are hard-coded: pds, ikey, lpds, bin, lbin. Among them a separate list of files which allowed to be injected: pds, bin, idxa.

allow(*self*, *type*)

Check if given file type is allowed in EventStore.

Return Value

true or false.
(type=integer)

allowToInject(*self*, *type*)

Check if given file type is allowed for injection in EventStore.

Return Value

true or false.
(type=integer)

esFileTypes(*self*)

Return a list of known file types supported in EventStore.

Return Value

list of known file types supported in EventStore
(type=list)

esInjectionFileTypes(*self*)

Return a list of known file types supported in EventStore which allowed for injection.

Return Value

list of known file types supported for injection to EventStore
(type=list)

isDatType(*self*, *iFileType*)

Check if given file type is data file.

Parameters

`iFileType:` file type
(type=string)

Return Value

non-zero value if given file type is one of the data types, e.g. pds
(type=integer)

isKeyType(*self*, *iFileType*)

Check if given file type is index file.

Parameters

`iFileType`: file type
(type=string)

Return Value

non-zero value if given file type is the key file type
(type=integer)

isLocType(*self*, *iFileType*)

Check if given file type is location file.

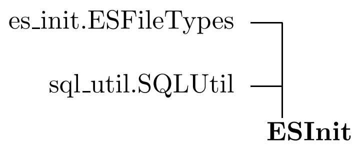
Parameters

`iFileType`: file type
(type=string)

Return Value

non-zero value if given file type is one of the location types, e.g. lpds
(type=integer)

22.3 Class ESInit



Known Subclasses: ESDeleteManager, ESDumpManager, ESManager, ESMergeSVManager, ESMoveManager, ESVersionManager

Base class which establish connection with EventStore

22.3.1 Methods

__init__(*self*, *db*, *dbType*, *logFile*)

Initialize database pointer, cursor, db type (MySQL or SQLite). Retrieve information about table names from underlying DB.

Overrides: es_init.ESFileTypes.__init__

Inherited from ESFileTypes: allow, allowToInject, esFileTypes, esInjectionFileTypes, isDatType, isKeyType, isLocType
Inherited from SQLUtil: close, commit, createTables, dropTable, endTxn, fetchAll, fetchOne, findFileForRun, getAllParents, getIsolationLevel, getLastId, getTableNames, getTables, getTableSchema, idx, lockTables, makeESQuery, printDBContent, printESInfo, printRuns, rollback, setCommitFlag, setIsolationLevel, setVerboseLevel, showDepend, startTxn, unlockTables, updateDBAndLog, updateLog, writeToLog

23 Module es_logger

Simple logger base class for EventStore

23.1 Class ESLogger

Simple logger base class for EventStore

23.1.1 Methods

__init__(*self*, *logName*, *logType*=`()`, *formatType*=`''`)

Set EventStore logger with logType tuple, e.g. (”stream”,) or (”fileName”,’a’) It also support to choose formatType: short or long

addHandler(*self*, *paramTuple*)

Add new handler to the logger. Available handlers are: stream, file, socket, http

critical(*self*, *msg*)

Invoke logger critical printout

debug(*self*, *msg*)

Invoke logger debug printout

error(*self*, *msg*)

Invoke logger error printout

info(*self*, *msg*)

Invoke logger info printout

logSQL(*self*, *msg*)

Invoke logger SQL printout

setLevel(*self*, *level*=`''`)

Set logger level. Available levels are: CRITICAL, ERROR, WARNING, INFO, DEBUG, NOTSET

warn(*self*, *msg*)

Invoke logger warn printout

24 Module esdb_auth

This module provides `md5crypt` authentication to EventStore MySQL DB

24.1 Functions

authToESMySQL(*mysqlHost*, *userMySQL=''*, *passwordMySQL=''*)

Provides authentication with EventStore DB.

dissassembleDBName(*db*)

Dissassemble the input db string.

Parameters

`db`: input db string, e.g. EventStore@lnx151:3306:/var/log/mysql
(type=string)

Return Value

DB name, hostname of DB, its port and socket.
(type=tuple (dbName,dbHost,dbPort,dbSocket))

readConfigFile()

Read and parse content of $HOME/.esdb.conf configuration file The following syntax is supported in configuration file: # ESDB configuration file login:password # ESDB Master #ESMASTER=db@host:port:socket or /path/my/sqlite.db Comments are started with '#' letter. User can specify login and password to provide access to MySQL DB, if SQLite is used they're ignored. Also, user may specify which master DB to use by providing its host and DB names.

25 Module executeSQL

A stand-alone script which executes SQL statements from ASCII file

26 Module feedMetaDataDB

MetaData injector tool. It is based on httplib to form a SOAP message For full description of WSDL and related stuff please visit http://cougar.cs.cornell.edu/CLEO/CLEO_admin.asmx

26.1 Functions

constructSOAPEnvelope(*method*, *aList*)

Construct a soap envelop for given method and argument list

endEnvelope()

Add end statement to soap envelop

headerEnvelope(*userName*=`'CLEOadmin'`, *password*=`'CLEOpassword'`)

Form a header of soap envelop which include user authentication

parseWSDL(*wsdl*=`'http://cougar.cs.cornell.edu/CLEO/test_CLEO_admin.asmx?W...`)

Parse a wsdl file. So far we use urllib to do a job to read content of the file

sendSOAPMessage(*method*, *envelope*, *test*=`1`, *debug*=`0`)

Send soap message to cougar.cs.cornell.edu. Right now we use httplib to do a job

soapBody(*method*, *argList*)

Form a body of soap envelop. Construct appropriate array of items to retrieve.

soapEnvelope()

Form a soap envelop in a form of CS web service wants. For service description, please consult
http://cougar.cs.cornell.edu/CLEO/CLEO_WS.asmx

26.2 Variables

Name	Description
HOST	**Value:** `'cougar.cs.cornell.edu'` *(type=`str`)*
NS	**Value:** `'http://cleo.lepp.cornell.edu/CLEO/'` *(type=`str`)*
PORT	**Value:** `80` *(type=`int`)*
WSDL	**Value:** `'http://cougar.cs.cornell.edu/CLEO/test_CLEO_adm-in.asmx?WSDL'` *(type=`str`)*

27 Module fileContent

A general wrapper over format dependent dump routines to print content of supported files in EventStore, e.g. pds, key, location, binary

27.1 Functions

fileContent(*args*)

fileContent is a wrapper over format dependent dump routines It dump content of all supported file formats in EventStore.

28 Module file_util

Set of high-level functions to build key/location files regardless from input data format. All EventStore file types are determined by fileType method.

28.1 Functions

build_key(*fileIn*, *fileOut*, *oFileID*)

A high-level method to build key files in EventStore

build_location(*fileIn*, *fileID*, *fileOut*, *allList*=`[]`)

A high-level method to build location files in EventStore

changeFileIdsInLocFile(*locFileName*, *fileIdList*)

A high-level method to change fileIds in location file to given list

fileParser(*fileName*, *what*=`''`)

A high-level method to parse data files in EventStore. Based on a file type it propagates a request to appropriate module.

fileType(*fileName*)

A high-level method to determine file type. It uses file signature:
- PDSSIGNATURE=3141592
- KEYSIGNATURE=2718281
- LOCSIGNATURE=2951413,
- BINARYSIGNATURE="RAW"
- LOCBINARYSIGNATURE="RAWL"
- SWAPPEDBINARYSIGNATURE="WAR",
- SWAPPEDLOCBINARYSIGNATURE="LWAR".

It is endian complaint.

getFileIds(*locFileName*)

A high-level method to get a list of fileIds from location file

getProxies(*fileName*)

A high-level method to get data (proxies) from given file

locationFileParser(*fileName*)

A high-level method to parse location files in EventStore

runParser(*fileName*)

A high-level method to parse files in EventStore and return run content.

29 Module fixTruncated

A helper script to fix truncated run in EventStore. The original key file entry is copied to view='all-problem', then new IDXA file is used to create a new key file. Its entry is replaced in KeyFile table of EventStore DB.

30 Module gen_util

A set of usefull utilities independent from EventStore

30.1 Functions

addToDict(*iDict*, *key*, *value*)

Add value as a list to the dictionary for given key. If dictionary contains such key, update its list with given value. Return dictionary itself.

changeFileName(*fileName*, *fromField*, *toField*)

Change portion of file name from 'fromField' to 'toField'. It uses string replace mechnism to make a change. Return original name if no 'fromField' and 'toField' provided.

dayAhead()

Form a day ahead in the YYYYMMDD format. To form such day we ask for seconds since epoch time.time(), add one day 60*60*24, convert to tuple in UTC format and send it for formating to time.strftime:
int(time.strftime("%Y%m%d",time.gmtime(time.time()+60*60*24)))

form64BitNumber(*lower*, *upper*)

Form a 64-bit number from two 32-bit ones

lowerUpperBitsOfUID(*uid*)

Return lower and upper bits of 64-bit number

printExcept()

print exception type, value and traceback on stderr

printListElements(*iList*, *msg=''*)

Loop over elements in a list and print one in a time on stdout

31 Module idxa_reader

Base class to read IDXA files

31.1 Class IDXAFileReader

Base class to read information from IDXA files.

31.1.1 Methods

__init__(*self*, *fileName*)

Base class to read information from IDXA files. You may access information from data members: svList, stramNames

Parameters

`fileName`: name of the file
(type=string)

Return Value

none
(type=none)

getRunUidList(*self*)

Return a list of run,uid pairs read it from IDXA file.

Return Value

list of (run,uid) pairs read it from IDXA file
(type=list)

getSVList(*self*)

Return a list of sync. values read it from IDXA file.

Return Value

list of sync. values read it from IDXA file
(type=list)

getSVStreamList(*self*)

Return a list of (sv,stream) pairs read it from IDXA file.

Return Value

list of (sv,stream) pairs read it from IDXA file
(type=list)

32 Module key_dump

Dump content of key (index) file. Also contain countEvents routine for counting number of events present in key file.

32.1 Functions

countEvents(*fileName*)

Event counter method, it counts a number of sync. values presented in a key file

decodeKeyRecord(*keyFile*, *needToSwap*, *nRecordTypes*)

Decode a record in key file at current position of key file

dump(*keyFileName*, *verbose*=`1`, *whereToWrite*=`'std'`)

Dump content of a key file. The output can be redirected to /dev/null or stdout.

dump_old(*fileName*, *verbose*=`1`, *whereToWrite*=`'std'`)

Dump content of a key file. The output can be redirected to /dev/null or stdout.

keyFileHeaderReader(*keyFileName*)

Read key file header and return file offset to Records

keyFileParser(*keyFileName*)

Parse key file and read back all syncValues

recreateKeyFile(*keyFileName*, *newFileId*, *fakeSVList*)

From given key file name, new file id and fake sv list recreate key file. Return new file name and write out the file.

stripKeyFile(*keyFileName*, *newFileId*, *skimKeyFileName*)

From given key file name, new file id and fake sv list recreate key file. Return new file name and write out the file.

33 Module key_fixer

Repairs a broken key file

33.1 Functions

fix(*fileName*, *uid*=`''`, *whereToWrite*=`'std'`)

fixes problems in key file. The output can be redirected to /dev/null or stdout.

34 Module lbin_dump

Binary location file dump and parser routines

34.1 Functions

dump(*fileName*, *verbose*=`1`)

Dump content of binary location file

getFileIds(*locFileName*)

Return a list of fileIds from location file header

getProxies(*fileName*)

Get proxies from a file. Since we know that raw files contains only RawEventData, we just return it for every known stream.

locationFileParser(*locFileName*)

Parse header of binary location file and read data types

35 Module locKeyReader

Location and key reader class which allow to read data from location and key files simultaneuosly (it allow to read data from one key file and a number of location files, e.g. pass2, post-pass2).

35.1 Class KeyLocFilesReader

Location and key reader class which allow to read data from location and key files simultaneuosly (it allow to read data from one key file and a number of location files, e.g. pass2, post-pass2).

35.1.1 Methods

__init__(*self*, *keyFileName*, *locFileList*)

Class constructor which initialize key and location file descriptions and reads their headers.

backToFirstRecord(*self*)

Return key and location files descriptors to the position of first records in those files

dataInStreams(*self*)

Return dictionary[stream]=[proxies] list of proxies in a stream

fileIDs(*self*)

Return list of file Id's from location file

formTupleTriplet(*self*, *recordInfoList*)

Read (sv,fileOffset,streamIdx) from each entry of given recordInfoList and form combined (sv,fileOffset,streamIdx). This method is used in `readRecordInfo` and readLastRecordInfo.

locFileList(*self*)

Return list of location files associated with given class

newOldIndecies(*self*)

Build conversion from new stream index to old one

numberOfSyncValues(*self*)

Return number of sync values from key file

readLastRecordInfo(*self*)

Read last record from key and location files and return tuple (syncValue, list of fileOffsets, stream index)

readRecordInfo(*self*)

Read one record from key and location files at their current position and return tuple (syncValue, list of fileOffsets, stream index). Internally it reads file offsets from all location files associated with key file and forms a combined files offset list.

streamNames(*self*)

Return list of streams in location file

35.2 Class LocKeyReader

Location and key reader base class which allow to read data from location and key files simultaneuosly.

35.2.1 Methods

__init__(*self*, *keyFileName*, *locFileName*)

Class constructor which initialize key and location file descriptions and reads their headers.

backToFirstRecord(*self*)

Return key and location files descriptors to the position of first records in those files

dataInStreams(*self*)

Return dictionary[stream]=[proxies] list of proxies in a stream

fileIDs(*self*)

Return list of file Id's from location file

listOfDataKeys(*self*)

Return a list of data keys in streams

newOldIndecies(*self*)

Build conversion from new stream index to old one

numberOfSyncValues(*self*)

Return number of sync values from key file

readLastRecordInfo(*self*)

Read last record from key and location files and return tuple (syncValue, list of fileOffsets, stream index)

readRecordInfo(*self*)

Read one record from key and location files at their current position and return tuple (syncValue, list of fileOffsets, stream index)

streamNames(*self*)

Return list of streams in location file

36 Module lpds_dump

PDS location file dump and parser routines

36.1 Functions

decodeLocationRecord(*locFile*, *needToSwap*, *recordSize*, *nFiles*)

Decodes a single location file record.

Parameters

`locFile:`	location file name *(type=string)*
`needToSwap:`	flag to inform if we need to swap bytes in in put loc. files *(type=integer)*
`recordSize:`	size of the record *(type=integer)*
`nFiles:`	number of PDS files which this location files describe *(type=integer)*

Return Value

(fOffsetList,streamIdx,dataKeyList) where fOffsetList is a list of file offsets, streamIdx is an index stream and dataKeyList is a list of proxies for this record.
(type=tuple)

dump(*fileName*, *verbose*=`1`)

Dump content of location file

dump_old(*fileName*, *verbose*=`1`)

Dump content of pds location file to stdout

find_datakeys_in_streams(*iRestOfHeader*, *iStreamNames*)

Search for data keys in all available streams

getFileIds(*locFileName*)

Return a list of fileIds from location file header

getProxies(*fileName*)

Return list of proxies (data) stored in pds location file

locationFileParser(*locFileName*)

Parse header of location file and read data types

read_tags(*iRestOfHeader*, *iIndex*)

In given header search for all available usage and production tags

36.2 Variables

Name	**Description**
LOCSIGNATURE	**Value:** `2951413` *(type=`int`)*
PDSSIGNATURE	**Value:** `2951413` *(type=`int`)*
pdsSignature	**Value:** `0` *(type=`int`)*

37 Module makeIDXA

IDXA file maker from provided PDS file

38 Module makePDSSkim

Make a skim file out of input PDS and IDXA files

39 Module md5crypt

md5crypt.py - Provides interoperable MD5-based crypt() function

SYNOPSIS

import md5crypt.py

cryptedpassword = md5crypt.md5crypt(password, salt);

DESCRIPTION

unix_md5_crypt() provides a crypt()-compatible interface to the rather new MD5-based crypt() function found in modern operating systems. It's based on the implementation found on FreeBSD 2.2.[56]-RELEASE and contains the following license in it:

"THE BEER-WARE LICENSE" (Revision 42): <phk@login.dknet.dk> wrote this file. As long as you retain this notice you can do whatever you want with this stuff. If we meet some day, and you think this stuff is worth it, you can buy me a beer in return. Poul-Henning Kamp

apache_md5_crypt() provides a function compatible with Apache's .htpasswd files. This was contributed by Bryan Hart <bryan@eai.com>.

39.1 Functions

apache_md5_crypt(*pw*, *salt*)

Provides a function compatible with Apache's .htpasswd files.

md5crypt(*pw*, *salt*, *magic*=`None`)

Provides a crypt()-compatible interface to the rather new MD5-based crypt() function found in modern operating systems.

to64(*v*, *n*)

unix_md5_crypt(*pw*, *salt*, *magic*=`None`)

Provides a crypt()-compatible interface to the rather new MD5-based crypt() function found in modern operating systems.

39.2 Variables

Name	Description
ITOA64	**Value:** `'./0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefgh-ijklmnopqrstuvwxyz'` *(type=`str`)*

40 Module os_path_util

Set of high-level utils to manipulate with arbitrary files

40.1 Functions

checkPermission(*dir*)

Check permission to directory

formAbsolutePath(*file*)

Form absolute path to given file

isFile(*fileName*, *verbose*=1)

Check if given fileName is a file. If it's a link print out a file name it points to

40.2 Class NullDevice

Redirect stdout to /dev/null

40.2.1 Methods

write(*self*, *s*)

41 Module pds_dump

PDS file dump tools

41.1 Functions

decodeVersionInfo(*fileName*)

Decode VersionInfo from beginrun record. VersionInfo consists of:

- softwareRelease : string
- specificVersionName : string
- configurationHash : string
- ordinal : unsigned int no packing
- ancestors : container of string

So, it alwasy grow, e.g. post-p2 file will contains two VersionInfo's, one for itself and one for it's parent. So, the underlying algorithm creates a list of VersionInfo's in the following format: [(childTag,[softRel,svName,hash,id,parent1,parent2,...]),(parentTag,[...])] This method returns a (svName,[listOfParents])

dump(*fileName*, *verbose*=`0`, *iStream*=`'event'`)

Dump content of pds file to stdout

formVersionInfoWord(*list*)

Form version info word from provided list of attributes

getPDSHeader(*fileName*)

Return pds header in form of array.array

getPDSRecordInfo(*pdsFile*, *needToSwap*)

Read information from one PDS record

Parameters

`pdsFile:`	pds file descriptor *(type=file descriptor)*
`needToSwap:`	flag which indicates if we need to swap bytes in pdsFile *(type=integer)*

Return Value

(syncValue, fileOffsetList, recordIndex)
(type=tuple)

pdsHeaderParser(*fileName*)

Parser header of PDS file

proxyReader(*fileDesc*, *needToSwap*, *proxyLength*)

Proxy reader from provided file descriptor

readPDSRecord(*pdsFile*, *needToSwap*)

Read one record from PDS file, input parameter 'pdsFile' is open PDS file at position of first record

readPDSRecord_old(*pdsFile*, *needToSwap*)

Read one record from PDS file, input parameter 'pdsFile' is open PDS file at position of first record

41.2 Class PDSFileReader

Base class to read PDS file

41.2.1 Methods

__init__(*self*, *fileName*)

backToFirstRecord(*self*)

PDS file descriptor of the first record in a file.

Return Value

seek to the first record in a file
(type=none)

dataInStreams(*self*)

Return a dictionary of stream:data

fileDesc(*self*)

PDS file descriptor.

Return Value

file descriptor return: pds file descriptor

needToSwap(*self*)

Return a flag if we pds file was produced on another endian node

newOldIndecies(*self*)

Build conversion from new stream index to old index

readLastRecordInfo(*self*)

Return last record in pds file. Record info is retrieved by `getPDSRecordInfo` method

readRecord(*self*)

Read one record in pds file, we use `readPDSRecord` method

readRecordInfo(*self*)

Return current record information, we use `getPDSRecordInfo` method

setPosition(*self*, *pos*)

Set position of pds file descriptor.

Parameters

`pos`: position of the file
(type=integer)

Return Value

seek to provided position in a file
(type=none)

streamNames(*self*)

Return stream names

42 Module pds_merger

PDS files merger

43 Module pds_reader

PDS file reader include two parser: pdsRunParser only scan PDS files and collect all stored runs pdsParser is a full parser which collect syncValues and all proxies

43.1 Functions

fileInfo(*fileName*)

Dump content of pds file to stdout

pdsParser(*file*, *what*=`''`)

PDS file parser. Return a list of run/uid/sync.values/proxies in given file

pdsRunParser(*file*, *what*=`''`)

PDS run parser. Return run/uid list

printProxiesInAllStreams(*streamProxyList*)

Print proxies in all streams

44 Module pds_utils

PDS file utilities (read number of proxies, etc.)

44.1 Functions

charArrayOfStreams(*streamNames*)

Return an array Int32 words corresponding to provide stream names

could_be_start(*iRestOfHeader*, *iStartIndex*, *iNumStreams*, *iNumProxies*, *iNumSeenStreams*)

Determine if position in given header can be a start of new record Start of list should have

- a number less than # proxies followed by another number less than # proxies followed by a value that could be a valid string
- or a 0 followed by a condition 1 or 2 BUT the number of times this condition is applied must be less than the number of streams

could_be_string(*iRestOfHeader*, *iStartIndex*)

Determine if position in a header could be a string

find_probable_start(*iRestOfHeader*, *iStartIndex*, *iNumStreams*, *iNumProxies*)

Find a new start in pds file: Start of list should have

- a number less than # proxies followed by another number less than # proxies followed by a value that could be a valid string
- or a 0 followed by a condition 1 or 2 BUT the number of times this condition is applied must be less than the number of streams

find_proxies_in_streams(*iRestOfHeader*, *iStreamNames*, *iProxyNames*)

Find data proxies in a data streams

name_list(*iArray*)

Construct a name list from given array

name_list_from_file(*fileDesc*, *withNumberOfWords*=`''`)

Form a name list

pdsDumpTime(*file*)

Dump unique id (UID) of pds file, in old notation it was called time

printProxies(*pList*)

Print proxies in pds file

read_tag(*iCharArray*, *iIndex*)

Read usage/production tag from given array

read_tags(*iRestOfHeader*, *iIndex*)

Read usage/production tags from given header

44.2 Variables

Name	Description
ex_found_bad_word	**Value:** `'found bad word'` *(type=`str`)*
MAGIC_NUMBER	**Value:** `4294967296L` *(type=`long`)*

45 Module sql_util

SQLUtil class defines high-level API for EventStore tables. Use this class to create/drop/access/print content of ES tables.

45.1 Class SQLUtil

Known Subclasses: ESInit

SQLUtil class defines high-level API for EventStore tables.

45.1.1 Methods

__init__(*self*, *db*, *dbType*, *dbLog*=`''`)

close(*self*)

Close cursor and db connector

commit(*self*)

Explicit commit

createTables(*self*, *table*=`'all'`)

Create table(s) in EventStore database. For complete list of tables please visit: https://wiki.lepp.cornell.edu/CleoSWIG/bin/view/Main/EventStoreDesign

dropTable(*self*, *table*)

Drop table in EventStore

endTxn(*self*, *msg*=`''`)

Update EventStore db log and invoke COMMIT transaction. In the case of SQLite we use DB-API db.commit() to commit our transactions. Please note that SQLite only support DB locking.

fetchAll(*self*, *query*)

Update a EventStore log and retrieve all rows for given query

fetchOne(*self*, *query*)

Update a EventStore log and retrieve one row for given query

findFileForRun(*self*, *run*, *time*=`0`)

Find location of the data file holding given run

getAllParents(*self*, *childName*)

Get all parents for given child name

getIsolationLevel(*self*)

Get isolation level of SQLite DB

getLastId(*self*, *table*)

Get last autoincemented id for given table

getTableNames(*self*)

Return a list of table names in EventStore DB

getTables(*self*)

Return list of tables from DB

getTableSchema(*self*, *tableName*)

Return EventStore table schema

idx(*self*, *name*)

lockTables(*self*, *tableName*=`''`)

Lock tables in EventStore. If no tableName is provided, then lock all tables

makeESQuery(*self*, *timeStamp*=`-1`)

Return the following dictionary: dict[time]=[(grade,minR,maxR,svName)]

printDBContent(*self*, *table*)

Print content of given table in EventStore DB

printESInfo(*self*, *timeStamp*=`-1`)

Dump EventStore DB content in user readable format.

printRuns(*self*, *minR*, *maxR*)

Print list of runs for given run range

rollback(*self*)

Rollback transaction

setCommitFlag(*self*, *flag*=`1`)

Set flag which will inform underlying DB what to do with transaction commits.

setIsolationLevel(*self*, *level*=`None`)

Set isolation level for internal DB (necessary to perform SQLite transactions.

setVerboseLevel(*self*, *verbose*)

Set verbose level, 0 low, 1 is high. In future may verbose levels can be added.

showDepend(*self*, *childName*)

Print all dependencies for given data version name.

startTxn(*self*, *msg*=`''`)

Update EventStore db log and invoke BEGIN transaction. In the case of SQLite we rely on IMMEDIATE transaction mechanism which locks DB for that transaction.

unlockTables(*self*)

Unlock all locked tables in EventStore

updateDBAndLog(*self*, *iQuery*, *cQuery*=`None`)

updateLog(*self*, *iQuery*, *cQuery*=`None`)

Update EventStore log and execute given query. The log is written in the following format: hh:mm:ss pid query

writeToLog(*self*, *msg*)

Update EventStore db log with time/pid/msg signature

46 Module syncToMaster

Synchronize slave MySQL server up to certain position of the master one

46.1 Functions

connect(*dbHost*, *dbPort*, *dbName*, *userName*='', *userPass*='')

Establish connection to MySQL db

getSlaveInfo(*curS*)

Retrieve slave information

syncToMaster(*masterHost*, *slaveHost*, *dbName*, *userName*='', *userPass*='')

Try to synchronize slave to certain position in a master DB server

Index