Mart Loader

latest update: **120427154429**
svn directory

table of contents:

Introduction

This documentation is mostly intended for developers rather than deployers, the latter should preferably refer to the deliverable page (although some concepts explained here may be useful as well). The idea in this document is that a picture is worth a thousand words, and that the explanations and diagrams presented here should be enough to understand how the *loader* works.

Code

Getting the code:

```
svn co https://code.oicr.on.ca/svn/biomart/martloader/executor loader # directory for runner (executor is the former name), for the time being runner is the only constituent
```

you need Maven >=2 installed, a UNIX-based system and a *MySql* instance

After reading this documentation, looking at the code may be helpful as well. The most important elements are in order:

- *run.sh* (calls everything)
- *LogicalPlanConverter.java* (although its parent class PlanTransformer.java is where most of the logic really is)
- *ConcretePlanConverter.java* (same comment)
- *BatchRunner.java*

- each individual run step: *JvConcatenateStep.java*, *NxJoinStep.java*, ...

These java classes will be explained further down. Also see the development page for comments.

Constituents

Currently the loader is composed of:

1. an *adaptor* component - temporary replacement for the *planner* component - that reproduce part of the *planner* behavior (mostly tailoring an action plan to the input provided)
2. a *runner* component, described in this document

Input

Description

The best way to know what the loader takes as input from a user (deployer)'s perspective is to run:

```
./run.sh --help
```

It essentially takes an input directory containing source datafiles and a number of options to tailor the run. In the background however, *runner* also take 3 more input that are currently shipping with the distribution (see absense of planner section below):

1. a directory containing TSV files describing the input data model initially in files (**what we have**)
2. a directory containing TSV files describing the output data model eventually in a database (**what we want**)
3. an dependency graph file (format: *graphml*) describing the steps necessary to transform the input into the output (**how to get from what we have to what we want**)

A more comprehensive description for those elements can be found on the *planner* to *runner* communication page

Absence of planner

As mentioned above, the current *loader* does not come with the *planner* component which is due for improvements. The input described above is that of the current version. Should *planner* be included, the internal input (on top of the source datafiles) would only be:

- a data model file (format: custom xml) describing both the input data model initially in files (**what we have**) and the output data model eventually in a database (**what we want**)
  It would then figure out the *graphml* (**how to get from what we have to what we want**) part on its own and feed it to the *runner*. It would also extract the information useful to *runner* (that relates to the source and target data model), and create the 2 corresponding directories described above (containing TSV files).

It is therefore supposed to be the *planner* that generates the appropriate set of directories and *graphml* described above. In the absence of it however, we are including a *graphml* describing the current ICGC data model, and use the *adaptor* component to remove unwanted nodes based on user-defined options and available input files. This is a **hack** and is not the original design, hopefully we will address this situation soon.

Dummy input

A dummy input directory is included under *test_data/source/stsm* and should run under 3 minutes. To run it (using the *src* code directly):

```
# still from the loader directory checked out from SVN
export DCC_LOADER_DEV=true # so it knows to use the src code directly rather than the JAR
export DCC_LOADER_SKIP_INDICES=true # quick and dirty way to skip indices creation
./run.sh -hmy_mysql_host -umy_username -pmy_password \
  -s ./test_data/source/stsm -Dmy_target_db -e # "-e" option to skip loading Ensembl tables
```

Output

Mart

The output of a full run (it is possible to specify a subset of run, such as validation only - see "run.sh --help") is a loaded mart.

Console output

The output on the console tries to be as informative as possible. During the "core" run, it shows the following:

- overall progress (as %)
- count of joins performed
- count of reverse joins preformed (understand nix's "join -v")
- count of table loads
- count of index creation
- count of table creation (load + all indices)

Other output

Two other output are created: a working directory and a log directory. The former can safely be removed after run if everything went well. The latter contains warning/error files describing the issues encountered (such issues will show on the console as well, as either the WARN or ERROR levels).

Miscellaneous

Vocabulary

The terms *step* and *task* tend to be used interchangeably throughout the document (and code). Likewise, *graph* (or *graphml*) and *plan* also describe similar concepts. In the future we will try to make this more uniform throughout.

12/03/26 limitations

- context:
    - runner assumes that it runs on a *UNIX*-like machine
    - runner assumes a *mysql* instance is accessible
    - it parallelizes tasks but does not use a distributed environment (although it is written to account for one via an interface)
- no recovery possible
- no graphical monitoring available yet

Important remarks

- the "*sort*" (regular, merge and uniq) *UNIX* command is used extensively as it seems to be the most efficient file sorting tool out there
- headers are externalized as early as possible so as to simplify operations on files: each data file has a counterpart header (*.hd*) file
- source datafiles:
    - acceptable character set for input data files is *UTF-8* (enforced)
    - datafiles must specify a header row describing the columns. this row can be preceded with any number of empty lines and/or comments (lines starting with "[\t]*#")
    - past the headers, data must be compliant with data model; empty lines are however acceptable (but not comments)

Tests

Unit tests

There are unit tests for the following:

- run steps: ensures that each step does what it is expected to do (for instance that *filter* does filter columns)
- objects: ensures helper objects work as expected

To run tests:

```
mvn -DargLine="-ea -Drdbms.host=${RDBMS_HOST?} -Drdbms.port=${RDBMS_PORT?} -Drdbms.user=${RDBMS_USER?}
-Drdbms.passwd=${RDBMS_PASSWD?}" \
    -Dtest=JvProcessStepsTests,JvValidateStepsTests,SqlStepsTests,NxStepsTests,ObjectTests test
```

Integration tests (Jenkins)

They run as follow:

- every 15 minutes: unit tests + full run on a small dataset (runtime ~1 min)
- every 3 hours: full run on simulated data (runtime ~20 mins)
- every night: full run on 6 real datasets (runtime ~ 3 hours)

In all cases, the resulting databases are compared to their reference counterpart (considered to be correct upon manual inspection). The Jenkins tests use Maven Surefire and Emma plug-ins for reports.

Main algorithm

Logical, Concrete & Atomic plans

It is important to understand what the *logical* plan is first in order to understand the main algorithm.

The best way to understand the main (*logical*) steps involved in *loader* is to refer to the graph diagram. It is not 100% reflecting what actually happens, but is the best visual aid available (and the inacurracy merely fail to illustrate some optimizations)

Alternatively, one can open the actual *graphml* file. This is the directed graph representing the latest transformation needed to be applied on the latest ICGC data model. This file should in theory be generated by planner, but is instead included directly and parsed by the temporary *adaptor* component. It is possible to vizualise the graph described by this file using yEd graph editor.

The separation between *concrete* and *atomic* is somewhat subjective, one could argue it would be best to apply one single such conversion. However keeping them separated was not a big development effort and hardly affects execution time (especially compared to the time it takes to actually run the *atomic* steps themselves). The rational behind it was the following:

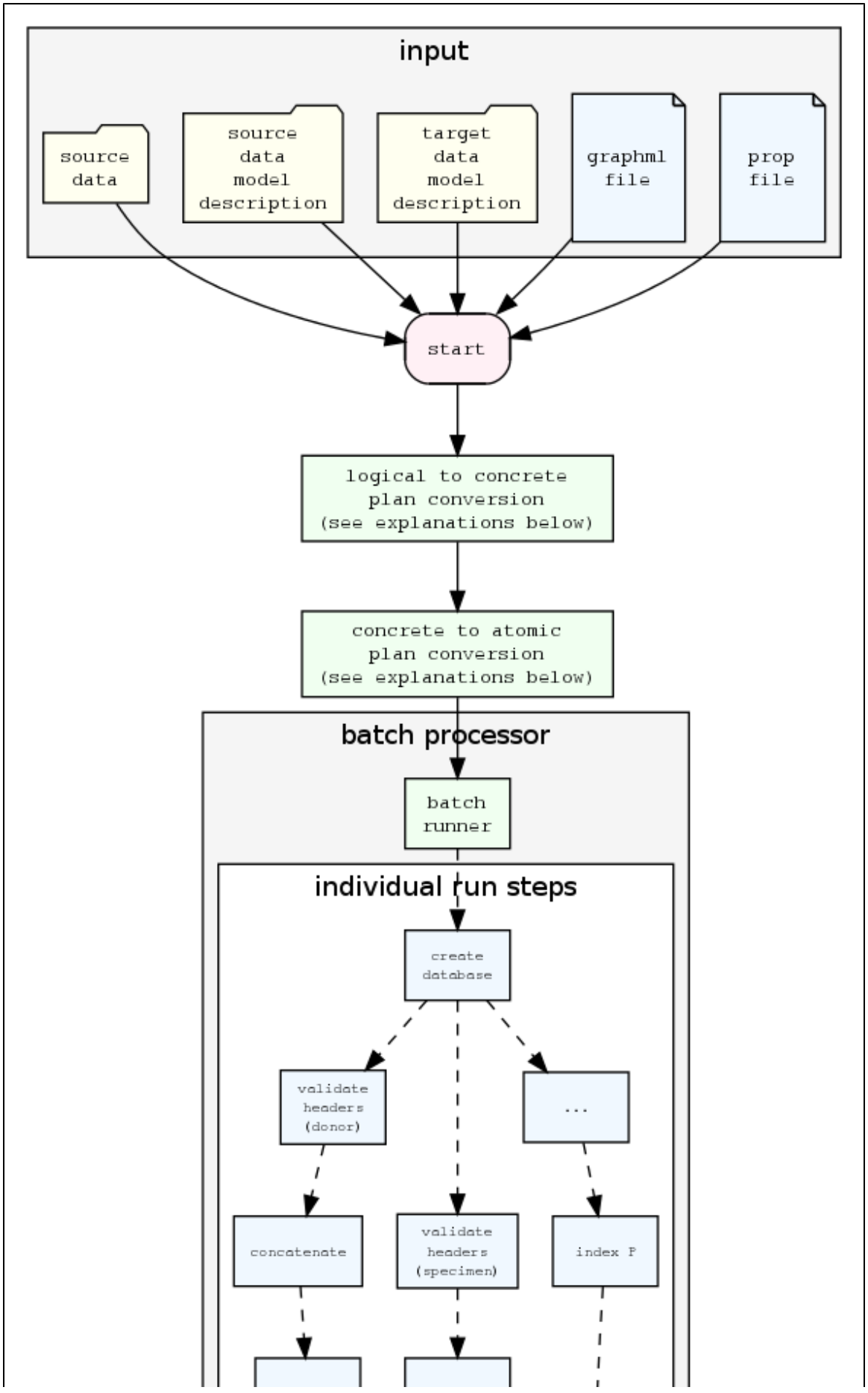
- the *logical* to *concrete* conversion is environment agnostic, it simply break down "logical" steps into what they correspond to (see diagrams)
- the *concrete* to *atomic* conversion is the one that accounts for the environment: what kind of operating system we have and what rdbms type will be used to load the marts (possibly more in the future). It then breaks down *concrete* steps accordingly (see diagram)

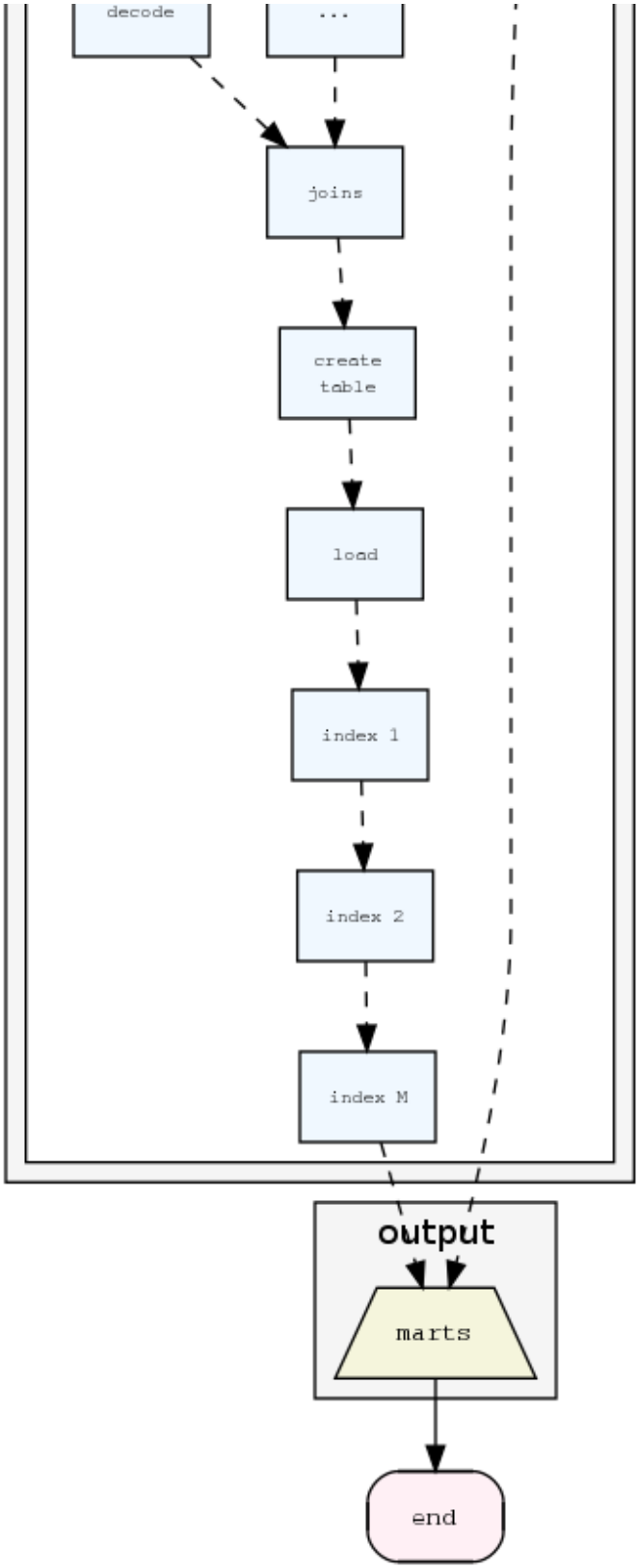
At the moment we only support UNIX-based system that have a *MySql* instance available, so the decision-making is minimum in the *concrete* to *atomic* conversion. However the code is written in such a way that we can offer more options in the future: handling *Oracle* for instance (next on the product backlog), using a basic sort written in Java if running on Windows, ...

Flow chart (simplified)

notes:

- every **box** in the diagram is an independent program (abstraction layer) that can be called separately. The way to exchange information in between layers is via *XML* files: either in the form of *graphml* files for the conversions boxes, or *xstream* files for the individual run steps.
- dashed lines represent dependencies between steps but it must be understood that it is the *batch runner* component that acts as master and triggers steps based on their dependencies


input
source data
source data model description
target data model description
graphml file
prop file
start
logical to concrete plan conversion (see explanations below)
concrete to atomic plan conversion (see explanations below)
batch processor
batch runner
individual run steps
create database
validate headers (donor)
...
concatenate
validate headers (specimen)
index P

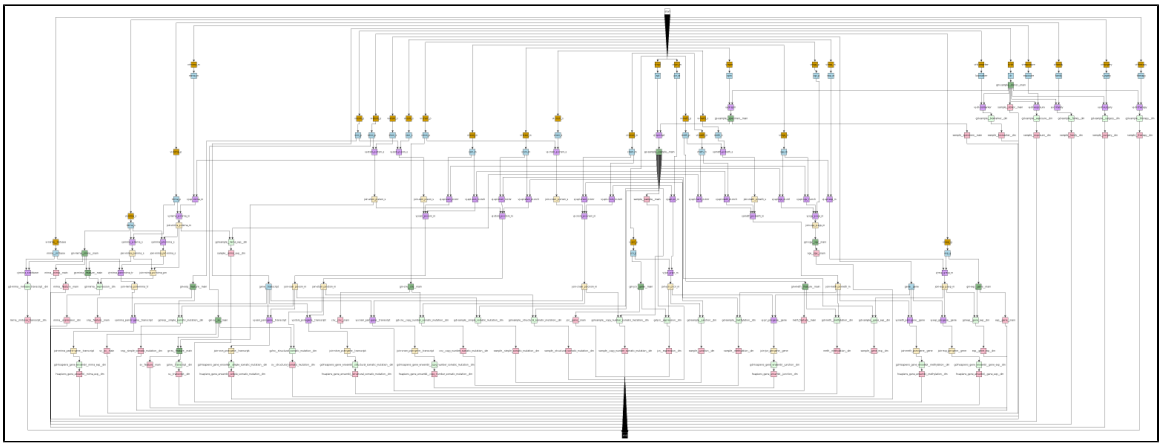

decode
...
joins
create table
load
index 1
index 2
index M
output
marts
end

Pseudocode

```
# ---------------------------------layer separation---------------------------------
# logical2concrete:

read graphml LOGICAL plan into DirectedGraph object
build LOGICAL Plan object from DirectedGraph object (see Plan and Task objects description further
down)
validate original LOGICAL Plan object
transform LOGICAL Plan object into CONCRETE Plan object (for instance LOGICAL "create DB table"
becomes CONCRETE "create table" + "load table" + "create indices", ...)
validate new CONCRETE plan object
write graphml CONCRETE plan

# ---------------------------------layer separation---------------------------------
# concrete2atomic:

read graphml CONCRETE plan into DirectedGraph object
build CONCRETE Plan object from DirectedGraph object
validate original CONCRETE Plan object
transform CONCRETE Plan object to ATOMIC Plan object (for instance CONCRETE "join" becomes ATOMIC
"split left" + "sort left" + ... + "merge" + "join" + "uniq", ...)
validate new ATOMIC Plan object
write graphml ATOMIC plan

# ---------------------------------layer separation---------------------------------
# batch runner:

read graphml ATOMIC plan into DirectedGraph object
build ATOMIC Plan object from DirectedGraph object
validate ATOMIC Plan object
for each ATOMIC Task object in plan:
  serialize parameters as xml file (xstream)
put root Task object in queue
for each Task object in queue:
  if Task object can start:
   new thread for Task object
   deserialize parameters from xml file
   submit step for run as individual program
   wait for it to finish running
   for each successor Task object:
     if successor Task object is ready to start (if all its other predecessor are finished too):
      enqueue successor Task object
```

Steps

Currently the logical steps for ICGC are organized as follow (see corresponding file for better resolution):

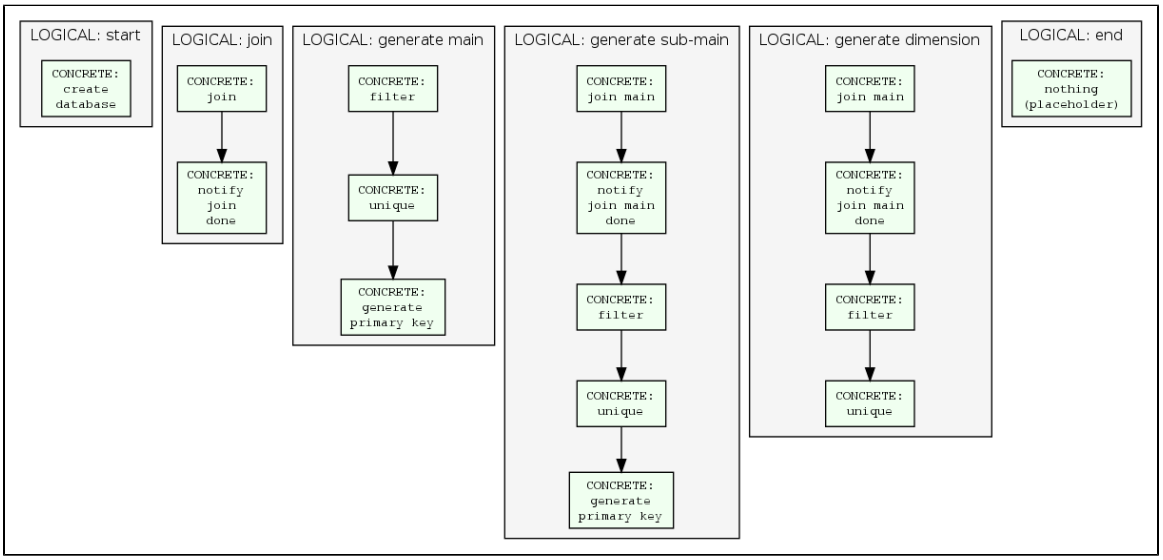


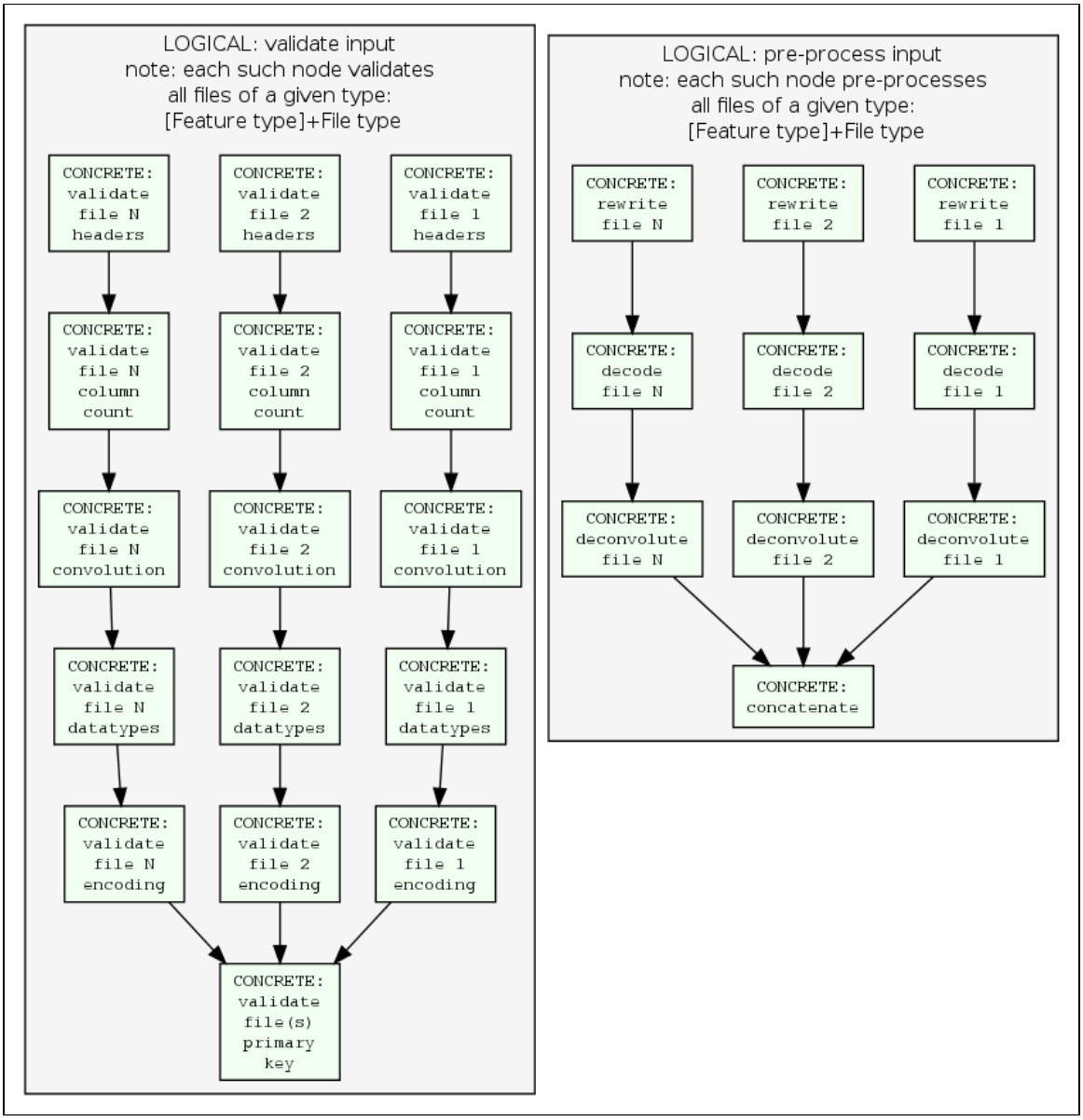
Logical steps

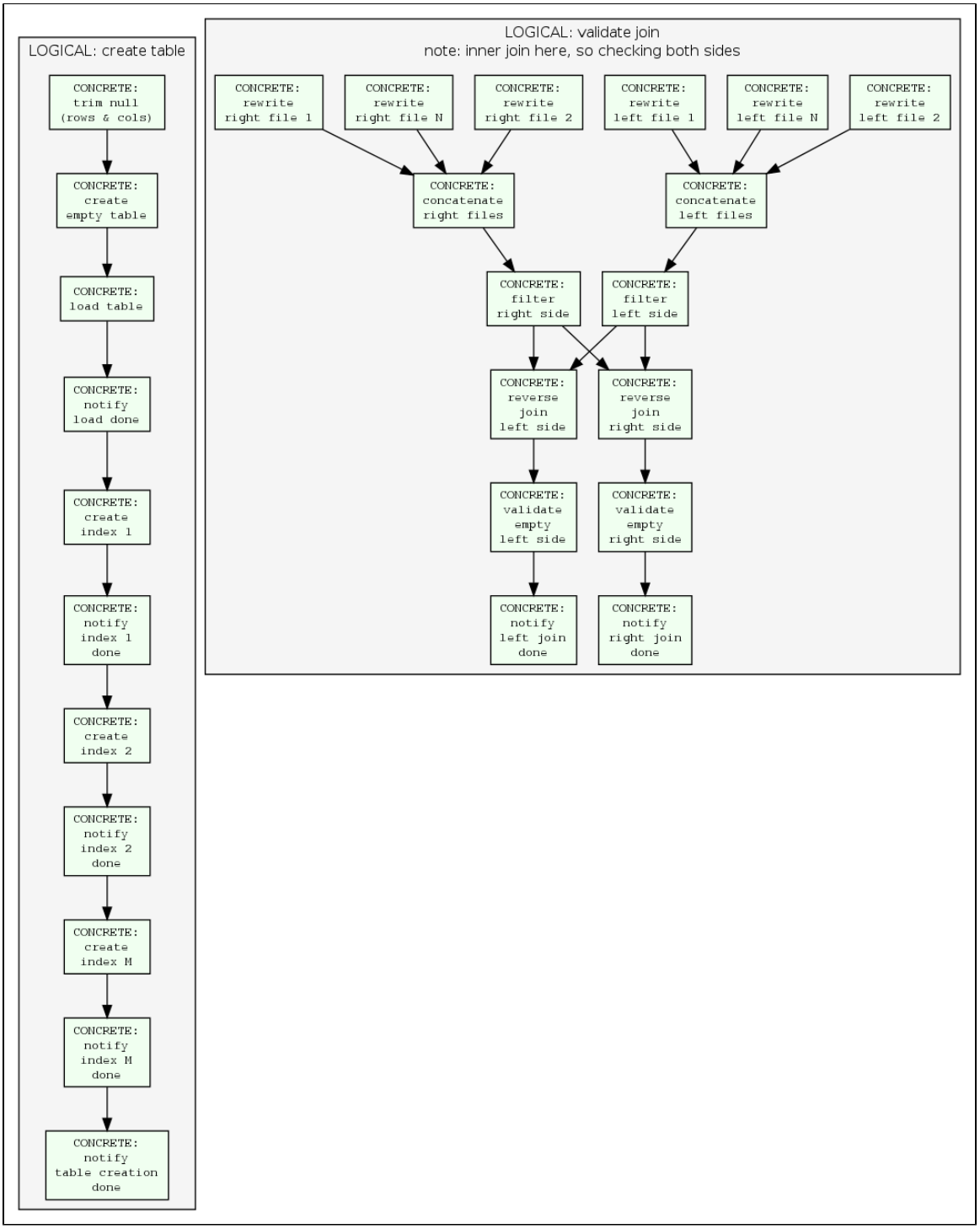
There are 10 of them and they are already described here

1. *start*
2. *validate input*
3. *preprocess input*:
4. *validate join*:
5. *join*
6. *generate main*:
7. *generate submain*
8. *generate dimension*
9. *create database table*
10. *end*

Under typical circumstances they are broken down into *concrete* steps as follow:




LOGICAL: validate input
note: each such node validates
all files of a given type:
[Feature type]+File type
CONCRETE: validate file N headers
CONCRETE: validate file 2 headers
CONCRETE: validate file 1 headers
CONCRETE: validate file N column count
CONCRETE: validate file 2 column count
CONCRETE: validate file 1 column count
CONCRETE: validate file N convolution
CONCRETE: validate file 2 convolution
CONCRETE: validate file 1 convolution
CONCRETE: validate file N datatypes
CONCRETE: validate file 2 datatypes
CONCRETE: validate file 1 datatypes
CONCRETE: validate file N encoding
CONCRETE: validate file 2 encoding
CONCRETE: validate file 1 encoding
CONCRETE: validate file(s) primary key
LOGICAL: pre-process input
note: each such node pre-processes
all files of a given type:
[Feature type]+File type
CONCRETE: rewrite file N
CONCRETE: rewrite file 2
CONCRETE: rewrite file 1
CONCRETE: decode file N
CONCRETE: decode file 2
CONCRETE: decode file 1
CONCRETE: deconvolute file N
CONCRETE: deconvolute file 2
CONCRETE: deconvolute file 1
CONCRETE: concatenate



Some *concrete* steps may be skipped based on the input, for instance:

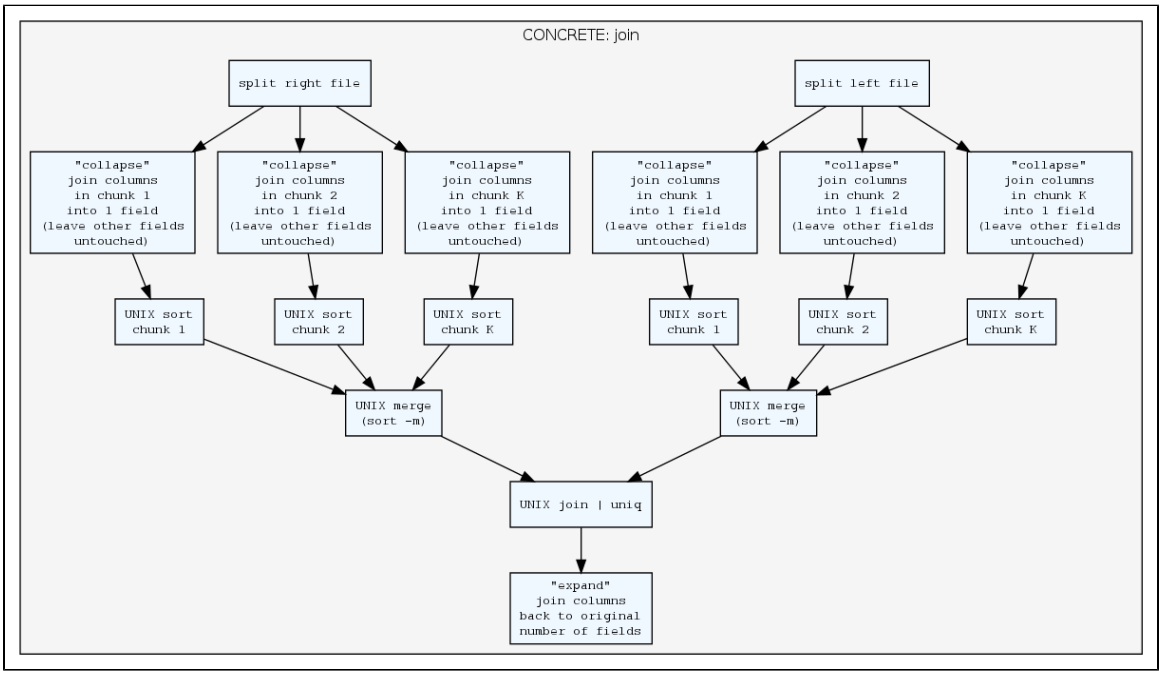
- no decoding step if there is no field to decode in a given datafile (the decision being based on source data model description as described above)
- no database creation if running in validation-only mode (see "./run.sh --help")
- ... refer to code for more detailed control flows (for instance see pre-process input helper)

Concrete steps

There are 22 of them; for now they mostly lead to their *atomic* counterpart, with *join* being the one notable exception

1. *validate header* : leads to atomic counterpart; ensures input file's header matches what is expected (order of columns does not matter)
2. *validate column count* : leads to atomic counterpart; self-explanatory

3. *validate convolution* : leads to atomic counterpart; ensure every "convoluted" column respect the convolution syntax
4. *validate datatypes* : leads to atomic counterpart; ensure every column respects its declared datatype (also checks charset for *string*-like fields)
5. *validate encoding* : leads to atomic counterpart; ensure every encoded column contains known codes
6. *validate primary key* : leads to atomic counterpart; ensure columns acting as primary key (possibly a composite one) are unique and not NULL
7. *validate emptiness* : leads to atomic counterpart; ensures a file is empty (as the result of a reverse join for orphan key check for instance)
8. *rewrite* : leads to atomic counterpart
9. *filter* : leads to atomic counterpart
10. *trim NULLs* : leads to atomic counterpart
11. *decode* : leads to atomic counterpart
12. *deconvolute* : leads to atomic counterpart
13. *concatenate* : leads to atomic counterpart
14. *generate key* : leads to atomic counterpart
15. *uniq* : leads to atomic counterpart
16. *join*: see diagram below
17. *create empty database* : systematically points to the mysql atomic counterpart as it is the only RDBMS we currently support
18. *create empty table* : same as above
19. *load table* : same as above
20. *create index* : same as above
21. *notify* : leads to atomic counterpart
22. *nothing* : placeholder mostly



Most other *concrete* steps won't need a helper. They'll simply refer to an *atomic* step counterpart (and will usually add a few more low level parameters to the task).

Atomic steps

There are 29 (21 + 8 join-specific ones as illustrated above) of them at the moment, although it is expected to grow as we offer more flexibility (for instance we will soon have four *oracle* counterparts to the *mysql* ones listed below):

1. *validate header* : see *concrete* step counterpart
2. *validate column count* : see *concrete* step counterpart
3. *validate convolution* : see *concrete* step counterpart
4. *validate datatypes* : see *concrete* step counterpart
5. *validate encoding* : see *concrete* step counterpart
6. *validate primary key* : see *concrete* step counterpart
7. *validate emptiness* : see *concrete* step counterpart
8. *rewrite* : externalizes headers and rewrites -777/-888/-999 codes to internal null value (later rewritten to '\N' prior to loading)
9. *filter* : removes unwanted columns
10. *decode* : proceeds with decoding
11. *deconvolute* : proceeds with "deconvolution"
12. *concatenate* : concatenates multiple files as one

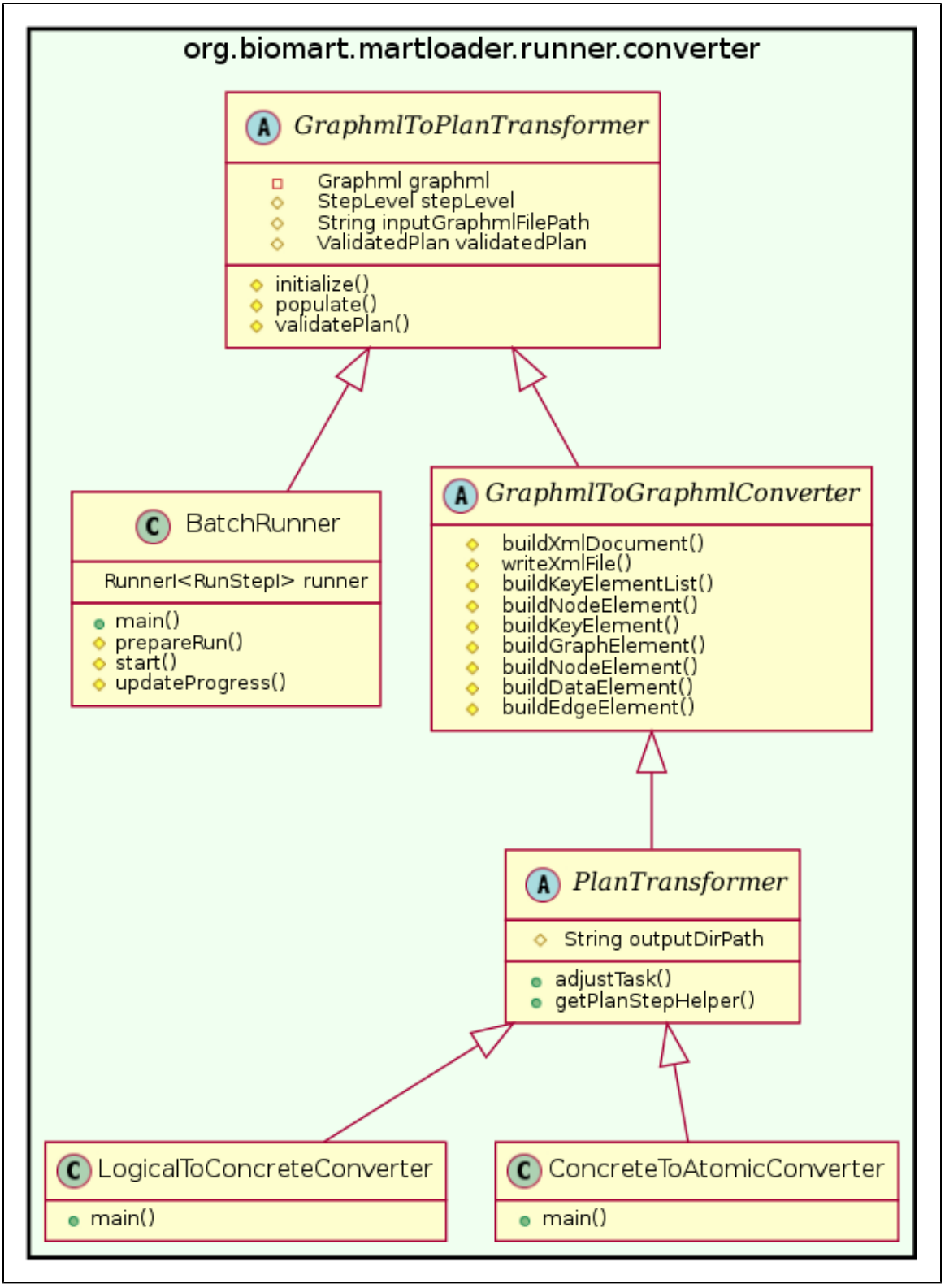
13. *generate key* : adds a generated primary key field to a file
14. *split* : splits a file into several smaller files
15. *collapse* : invented term to describe to action of putting together multiple fields as one (necessary for unix join)
16. *expand* : reverse operation of collapse
17. *unix sort basic* : self-explanatory
18. *unix sort merge* : merging/sorting of the files from several smaller sorted files
19. *unix sort uniq* : removes duplicate rows after sorting
20. *unix join* : self-explanatory
21. *mysql create empty database* : self-explanatory
22. *mysql create empty table* : self-explanatory
23. *mysql load table* : self-explanatory
24. *mysql create index* : self-explanatory
25. *nothing* : placeholder mostly

Diagrams

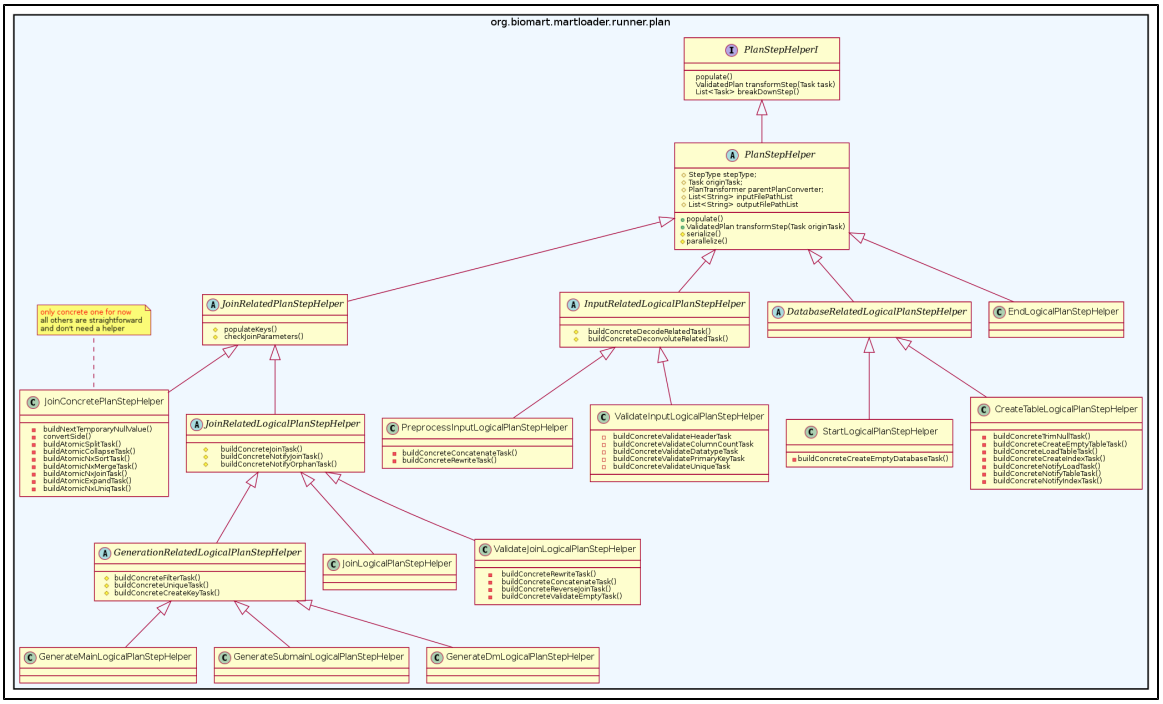
Class diagrams

Converters:

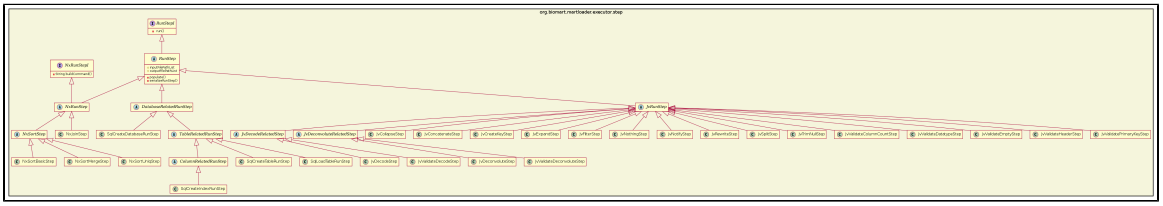
They handle the transformation of steps from one layer to the other. For instance *CreateTableLogicalPlanStep* generates a sub-plan composed of: create table, load table, create index 1, create index 2, ...


org.biomart.martloader.runner.converter
GraphmlToPlanTransformer
Graphml graphml
StepLevel stepLevel
String inputGraphmlFilePath
ValidatedPlan validatedPlan
initialize()
populate()
validatePlan()
BatchRunner
RunnerI<RunStepI> runner
main()
prepareRun()
start()
updateProgress()
GraphmlToGraphmlConverter
buildXmlDocument()
writeXmlFile()
buildKeyElementList()
buildNodeElement()
buildKeyElement()
buildGraphElement()
buildNodeElement()
buildDataElement()
buildEdgeElement()
PlanTransformer
String outputDirPath
adjustTask()
getPlanStepHelper()
LogicalToConcreteConverter
main()
ConcreteToAtomicConverter
main()

Plan steps:



Run steps:



Atomic graph

Graphical representation of the atomic graph on entire set of marts. This would be the result of converting the *logical graphml* file into an *atomic* one (via a *concrete* one as intermediate). Each yellow dot represents one of the 29 *atomic* steps