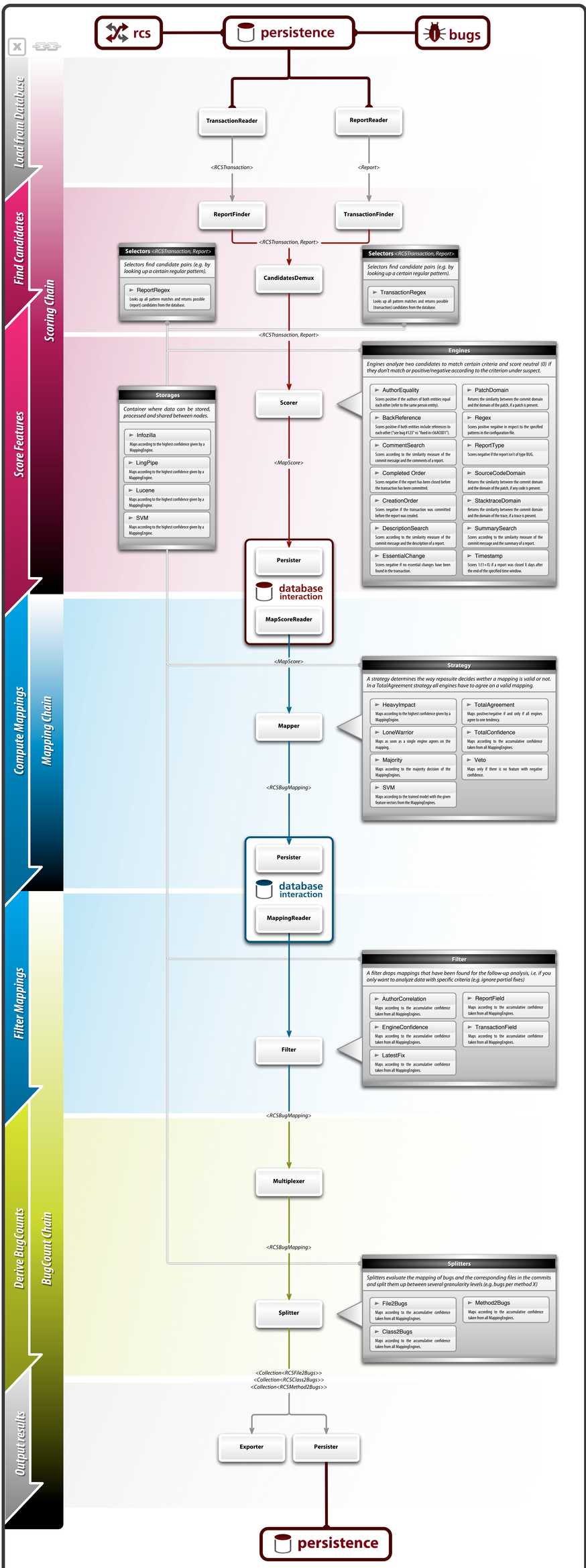
mappings



Description

Loading the data *from the database is the first step in the mappings toolchain. When reposuite is mapping reports to transactions (and vise versa), a source node is used for every possible data that can be used as a starting point. Here, reposuite loads all persistent transactions in one node and all minded reports in a second source node. As soon as one entity is loaded, it is transferred to the corresponding finder node.*

Finding mapping candidates *is done by the corresponding finder nodes (which are transformer nodes per definition—they take data of type A and transform them into data of type B; in this case Report/RCSTransaction to a candidate pair.*
*Finder nodes use all enabled Selector classes. By default, reposuite uses regular expression selectors which scan for certain (simple) patterns (e.g. "id+" in transactions for reports or "*p{IXDigit}{7,}" in comments of reports). If a selector finds possible candidates, reposuite checks if those references are valid, i.e. an entity with this identifier has been stored in the database. If so, the entity is loaded and the pair of transaction/report is considered a valid candidate.*
Candidates from all finders are accumulated by a demultiplexer node and transferred to the scoring node.

Generating feature vectors *for the candidates is a task that is accomplished by the scoring node. In the scoring step, reposuite uses all enabled engines to compute a vector consisting of confidence values. Every engine may have its own configuration options that are required to execute reposuite as soon as the engine is enabled. If the engine depends on certain storages, further config dependencies might be pulled in.*
An engine takes a candidate pair an checks for certain criteria and scores accordingly. Engines can score in three ways: positive, if they consider a pair a valid mapping, negative if they consider the pair to be a false positive or 0 if they can't decide on any of that.
A criterion of an engine should be as atomic as possible, that means an engine shouldn't check for multiple criteria at a time.
After all engines have been execute on one candidate pair, the resulting feature vector is stored to the database (incl. the information what has been scored by each engine and what data was considered while computing the score).

Reposuite relies on a strategy to be used when ***computing the actual mappings****. In this step, reposuite fetches all MapScores from the previous step from the persistence provider and evaluates the feature vector according to the selected strategy. E.g. if a TotalConfidence strategy is used, reposuite will only consider mappings as valid, if and only if the total confidence (the sum of all individual scores from the engines) yields a positive result. In a veto strategies, all mappings that have at least one negative value in the feature vector are dropped.*
Certain strategies rely on storages. E.g. the SVM strategy uses a model that has been build beforehand by having a support vector machine train on already mapped and verified data. If a mapping has passed the strategy checks it is persisted in the database.

Filtering the mappings *is done in another toolchain. Reposuite can post-filter the mappings that have been computed and stored in the database when performing any kind of analysis on this data. E.g. you want consider only mappings that refer to real fixes (no partial fixes) you can filter mappings using the LatestFix filter. You also might be interested by mappings of bugs that have been caused/fixed by certain persons. This kind of filtering is not reflected to the persistence provider since this would require a mapping of the whole dataset every time an analysis is performed.*
The resulting mapping is then transferred to the analysis part of mappings.

When ***counting bugs*** *per unit (where unit can be file, class, method, ...) reposuite first uses a multiplexer node to serve the mappings from the previous step to each available splitter. Splitters are nodes that compute the actual mapping of bugs per unit.*
Reposuite uses the information of the transaction to determine which files have been touched in the fix. It also filters out non-essential changes (such as renamings...) which are not relevant to the fix itself. There are further algorithms in reposuite modules that determine which part of an commit is relevant to the fix. Reposuite breaks down the remaining changes from the diff into classes and methods and uses the resulting information to compute the representing counts.

In the last step reposuite ***outputs the results****. Reposuite can either export the data directly (e.g. to a CSV file) or store the results back in the database. However, BugCounts that have been stored in the database are considered non-reliable data by reposuite since those counts may change according to the filters that have been used beforehand.*