Database and data model

Database links and data model documentation.

- Key links
- Data model

Key links

- SQL database dumps[1]
  - ddds_2026-11_0
- Core data model (v1.18)
  - diagram
  - documentation
- REST API
  - specifications (swagger)
  - specifications (redoc)
  - design
- Other links:
  - Hugging Face

1: Database dumps are created with pg_dump and compressed with zstd.

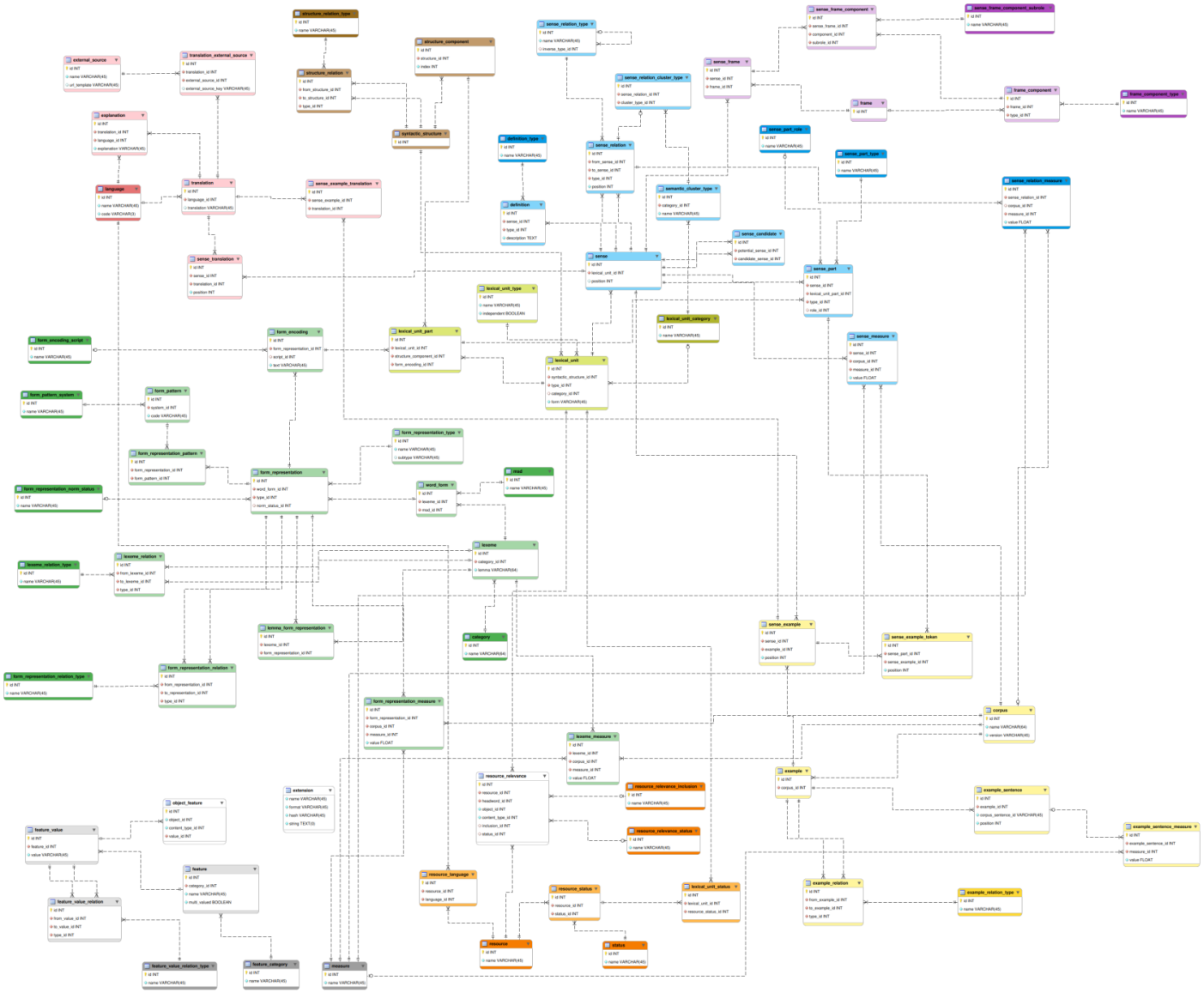
Data model

The central entity types of the datamodel are lexical units and senses. They connect the morpho-syntactic and semantical data in the data model. In essence the model is designed to be a multilingual model, however, currently it is used as a monolingual model that connects with multilingual data (which does not have the same level of granularity) via special entity types.

On the top level the model can be divided into clusters (color-coded in the model):

- lexical units (olive green)
- senses (blue)
- word forms (forest green)
- syntactic structures (brown)
- corpus examples (yellow)
- sense translations (red)
- sense frames (violet)
- resource connections (orange)
- generic features (grey)
- entity types that reference other entity types via meta-attributes (white)

The corpus data is not contained in the database itself, but is referenced and accessed via a concordancer. Some data (e.g., structure details) is defined in XML, which is used in existing processing pipelines; this data is stored in a separate generic SQL table.



Contents

- Overview
  - Purpose and state of this document
  - Database technology
  - Core model diagram
- Model clusters
  - Lexical units
  - Senses
  - Word forms
  - Syntactic structures
  - Corpus examples

- Sense translations
- Sense frames
- Resource connections
- Generic features
- Other formats

Overview

Purpose and state of this document

This document is intended for technical users who are working with the DDDS model and/or backups. For a higher-level, more theoretical and more linguistic description of the data model, see here (which is currently very brief, but will be expanded). For programmers who will also use the DDDS Django repository, see the `README_code.md` there (yet to be written, for now see `README_datamodel_old.md`).

This document describes DDDS model v1.18.

Database technology

The Digital Dictionary Database of Slovene is a PostgreSQL database. It contains core tables (with prefix "jedro_"), metadata tables (with prefix "metadata_") and internal tables managed by Django and other integrated packages (with prefixes "django_", "auth_", etc.). The core tables contain the Slovene linguistic data, while the metadata tables refer to the core tables and contain data which are not considered part of the language description, but which are needed for central applications (e.g., longer names for dictionaries to display in the database editor).

We define and manage the database via the Django ORM. Partly for this reason, all tables have a single rather than composite primary key (docs). Also, we make use of Django's content types and generic relations (docs), which allow us to associate some simple extra data related to any table without adding lots of new columns or many-to-many tables. However, this means that special care needs to be taken with this data if using the database outside of Django.

Furthermore, for several reasons, some data is more naturally stored in other formats. However, they are still technically part of the SQL database, so that all the data is in one place and so that SQL transactions can encompass changes to this data. They are stored as whole strings in the "extension" table. Currently we have only two such extensions, both in XML format, and we intend to keep this to a minimum.

This readme focuses on the core data model, which is by far the most complex part of the database.

Core model diagram

MySQL WorkBench diagrams are used to develop and visualise the core of the data model. The project's main Django models.py file is (manually) kept in synch with diagram changes.

The diagram contains several color-coded clusters of tables. For a conceptual explanation of the domain, see here. This readme will provide a more technical explanation and interpretation of these tables and the key relationships between them. Note that the tables with a relatively dark colour shade in the diagram are just basic coding tables (with only id and name), so will not be covered here explicitly, unless they are of particular importance. Also note that every table in this model also has a last_modified timestamp column, but these are not included in the diagram to avoid repetitive clutter.

Model clusters

Lexical units

The main lexical unit table (`LexicalUnit`) is the central table in the database. Lexical units consist of a type (`LexicalUnitType`), a syntactic structure (`SyntacticStructure`) (ref), and one or more parts (`LexicalUnit_Part`). They can also be assigned categories (`LexicalUnitCategory`).

There are six types of lexical unit, which fall under two broader categories: independent (`single_lexeme_unit`, `compound`, `phrase`, `quasi`) and dependent (`collocation`, `combination`). For example, "miza" would be a `single_lexeme_unit`, "okrogla miza" a `compound`, and "velika miza" a `collocation`. Whether a type is independent or dependent is stored explicitly in the `independent` column of `LexicalUnitType`. Independent types are potential headwords with their own entries in dictionaries, while dependent units can be included in the entries of headword units. The partial exception is the `quasi` type, which is used for units that are technically independent, and are needed due to sense relations of other units, but do not have sense definitions, etc.

At the level of the lexical unit tables, the type of lexical unit is generally irrelevant, but as we will see, the types do impact how associated data in some other tables are interpreted (e.g., senses, resources).

Lexical unit parts correspond roughly to tokens in corpora. Usually these are words, but not always (the 2nd part of "francosko-slovenski slovar" may correspond to the punctuation character `-`). In the data model, `LexicalUnit_Part` connects the lexical unit of the part (`LexicalUnit`), the component of that unit's syntactic structure the part corresponds to (`StructureComponent`), and the

form of the lexeme (`FormEncoding`) of that part. For instance, the first (of two) parts of "okrogla miza" connect the lexical unit "okrogla miza" with the first component of the common adjective-noun syntactic structure (ref) and the orthographic form of the appropriate form (feminine singular nominative) of "okrogel" (ref).

A lexical unit is uniquely determined by its type, structure and sequence of parts. Therefore, we cannot have multiple units which have the same combination of these properties, but we can have multiple lexical units which only partially match. For example, we may have two units "švicarski nož" (a `compound` and a `phrase`), or two noun `single_lexeme_unit`s for two forms of the same lexeme ("oblast" and "oblasti"). For convenience, each lexical unit also stores its canonical string (`form`) explicitly (e.g., "okrogla miza"), although this field is redundant and must be recalculated whenever the unit's parts or structure change.

Independent lexical units can be assigned a lexicographic category (`LexicalUnitCategory`). The values are similar to those of `Category` ([ref][#bkmrk-word-forms]), and for most units are thus unnecessary. However, this table makes it possible to assign a multi-part unit to a single category, which is helpful for technical reasons. For example, "covid-19" is stored as a 3-part ("covid", "-", "19") lexical unit (of type "compound") in the database, and thus cannot be assigned a single lexeme-level category, but by using `LexicalUnitCategory` it can still be conceptually labeled as a noun.

Senses

While lexical units may be the most central unit in the data model, senses (`Sense`) are probably the level to which the most data is attached. Lexical units have 1 or more senses, and each sense belongs to a particular lexical unit. Senses can have definitions (`Definition`), they contain parts (`SensePart`) and they can be related in various ways (`SenseRelation`). Certain relations can be grouped into predefined clusters (`SemanticClusterType`). We can also store measures of sense occurrences in corpora (`Sense_Measure`). And if we don't know which of several senses is appropriate for some data, we can group alternatives (`SenseCandidate`).

Definitions are string descriptions of a sense. They are only used for independent lexical units, for which they are the main aid in identifying senses for lexicographers and users. Definitions can be of different types, among which `indicator` is the most common and important.

Each lexical unit has 1 or more senses with a particular (possibly null) position. However, their interpretation depends on whether the lexical unit is dependent or independent (ref). For independent lexical units, `Senses` with a non-null position are "real" senses, normally equipped with further lexicographic data such as definitions or labels. The positions determine the order of a lexical unit's senses, normally via lexicographers' explicit decisions. But every independent lexical unit is also given a so-called "dummy" sense, which has null position and is used when we want to associate sense-level data with a lexical unit, but we don't yet know under which sense (which is a common situation because of the challenging nature of automatic semantic categorisation etc.). In addition, if we know that some data corresponds to one of a particular proper subset of a lexical

unit's senses, we can also have a sense with null position and candidate pairings (`SenseCandidate`), which relate particular senses (as `candidate_sense`) to that sense (`potential_sense`). However, this sense candidate support has not yet been put to use.

In (lexicographic) theory, dependent lexical units do not have senses, as indeed the main reason they are "dependent" is that their meanings derive somehow from the meanings of their parts (compare "okrogla miza" and "velika miza"). But from a technical point of view, since many kinds of data that are attached to senses (e.g., translations, examples, labels) are relevant for both independent and dependent units, dependent lexical units do have senses as well. For dependent lexical units, all senses have null positions (their ordering in particular contexts is determined by calculable criteria), and we do not anticipate to need sense candidates. However, a dependent lexical unit can still have multiple senses, such as the literal and figurative meanings of the collocation "svinjski jezik" (which will will have different translations, for example). Different senses of the same dependent lexical unit are distinguished by their parts (`SensePart`) and dependency relations (`SenseRelation`) (see below).

Sense parts (`SensePart`) serve two functions, identified by two different types. `within_other` parts indicate which lexical unit parts of a dependent unit correspond to the sense of a particular independent unit. For instance, the compound "okrogla miza" ("miza") is found in the 2nd and 3rd parts of the collocation "organizirati okroglo mizo". (The reason that sense parts refer to senses of independent units rather than the lexical units themselves is to make it easier to handle example tokens (ref).) `within_self` parts, on the other hand, allow us to indicate the `role` of the part in the sense (which is null for `within_other` parts). The roles are defined, managed and assigned by lexicographers. In `within_self` parts, we are always connecting lexical unit parts of a lexical unit with that lexical unit's own sense, which is redundant, but it does simplify our data model as it prevents the need for creation of two similar tables.

Sense relations (`SenseRelation`) relate pairs of senses with a certain relation type (`SenseRelationType`). Only one relation can exist between a given pair of senses (in a given direction). Senses of two independent lexical units can be related with classic semantic relations (e.g. `synonym`, relating a particular sense of "mali" with a particular sense of "majhen"). These relations always go in pairs, and the type of the reverse relation is stored explicitly in `SenseRelationType` as `inverse_type`. For example, `hyponym` and `hypernym` are each other's inverse types, so if "avto" is a hyponym for "vozilo", then "vozilo" is a hypernym for "avto". Symmetric types (e.g. `synonym`, `antonym`, `gender_pair`) are their own inverse type. Types without an inverse type (e.g. `dependency`, see below) are not paired in this way. In addition, a `subsense` relation indicates that one sense (`to_sense`) can be understood to be a subsense of another (`from_sense`). Basic statistical data for a relation (e.g., degree of synonymity) can also be stored (`SenseRelation_Measure`).

There is also a special relation type (`dependency`) which relates a dependent unit's sense (`to_sense`) with an independent unit's sense (`from_sense`). For example, the literal sense of "svinjski jezik" can be related to the physical senses of "svinjski" and "jezik", while its metaphorical sense can be related to more abstract senses of "svinjski" and "jezik". We can also have a sense of "svinjski jezik" which is not related to any independent unit senses, which would be the "dummy" sense for the dependent lexical unit. Therefore, sense relations between senses of independent lexical units

give additional information about senses, while dependency sense relations help define a dependent sense.

Dependent unit senses can also be organised in clusters under the independent senses which they are related to. A set of cluster types (`SemanticClusterType`) is pre-defined for each lexical unit category (`LexicalUnitCategory`). Each dependency sense relation can be assigned (`SenseRelationClusterType`) to one of the cluster types of the independent unit's category. For example, if the "noun" category has a "what material?" cluster type, then the "lesena miza"-"miza" relation would be assigned to this category for the appropriate sense of the lexical unit "miza".

Sense measures (`Sense_Measure`) record basic statistical measure values for a sense in a particular corpus. For instance, we would use this table to record that the physical sense of "svinjski jezik" occurs in a particular corpus 157 times. If we are dealing with a corpus which has not been semantically disambiguated, we can use the lexical unit's dummy sense (i.e., the position-less sense of an independent unit or the relation-less sense of a dependent unit).

Some sense-dependent tables have a `position` column which identifies the relative order of an object under its sense relative to other objects of the same type. This includes relations (`SenseRelation`), translations (`Sense_Translation`), examples (`Sense_Example`) and senses themselves (`Sense`). Positions start at 1. For sense relations, they are counted separately for each relation type (i.e. per `from_sense` and `type`), while for sense translations, they are counted separately for each language and translation cluster (the "cluster" feature of `Sense_Translation`). For examples, they must be unique per sense. Unlike `Sense.position`, which is null for some senses (ref), these positions are always set.

Word forms

Slovene is a highly inflectional language, where words have many forms with different sets of features, so the data model includes a hierarchy of tables for morphological data. From top to bottom, there are word grammatical categories (`Category`), form-independent lexemes of particular categories (`Lexeme`), the set of morphosyntactic descriptions (`Msd`) for each lexeme (`WordForm`), concrete forms for such combinations (`FormRepresentation`) and actual string representations of those concrete forms (`FormEncoding`). Lexemes can be related to each other (`LexemeRelation`) and have basic corpus statistics (`Lexeme_Measure`). Word forms have a hierarchy of form representations of different types, encoded as relations (`FormRepresentationRelation`). We can also store basic form representation corpus statistics (`FormRepresentation_Measure`), and classify form representations by their paradigm patterns (`FormRepresentation_Pattern`). Finally, lexemes also have canonical form representations for each type (`Lemma_FormRepresentation`).

Lexemes (`Lexeme`) represent a word (or punctuation) consisting of a lemma (the basic or dictionary form of the word) (e.g., "miza"), a category (e.g, "noun", "preposition", "punctuation") and a set of category-dependent lexeme-level MSD (see below) features (e.g., noun gender) (ref). This combination almost uniquely determines a lexeme, so we can have two different lexemes with the same lemma (e.g., "dolg"), or even with the same lemma and category (e.g., "klop"), but normally

not with the same lemma, category and lexeme-level MSD features. The exception is if we have differences in non-orthographic form representations (e.g., lesen (accentuation="lesén") and lesen(accentuation="lésen"). When these are encountered during matching and we have to make a choice automatically, we can exclude the lexemes marked with the feature "lexeme_homonym_secondary".

For a given lexeme, word forms (`WordForm`) are in effect abstract nodes for grouping together forms under particular MSDs (`Msd`). MSDs are linguistic codes which encode a particular combination of morphosyntactic features morphosyntactic description. For instance, Slovene nouns typically have 18 word forms (6 cases x 3 numbers), so there are 18 Slovene MSDs for nouns. Each `Msd` contains such a code, along with the features that are specific to particular forms (e.g., adjective gender), but the word category and lexeme-level features are not stored at this level, as word forms inherit them from their lexemes.

For each abstract word form, there is a hierarchy of form representations (`FormRepresentation`). We currently have three different types of form representations: `orthography` (e.g., "dekan"), `accentuation` (e.g., "dekàn") and `pronunciation` (e.g., "dɛ'kan"). Accentuation representations fall under particular orthography representations, and pronunciation representations fall under particular accentuation representations (e.g., "dɛ'kan" falls under "dekàn", not "dekán", although they are all form representations corresponding to the MSD "Ncmsn"). These inter-type relationships are encoded as relations (`FormRepresentationRelation`). In case there are multiple representations of the same type for a given form, `norm_status` can be used to indicate the representation's relative status (e.g., "non-standard", "variant").

The actual string representations of form representations are stored as form encodings (`FormEncoding`). This is a separate level, because there can be multiple encodings for the same representation using different encoding scripts. For instance, pronunciations can be encoded using SAMPA or IPA, and in some languages even orthographic forms are commonly written with different scripts (e.g., Cyrillic and Latin for Serbian).

As for senses, we can store basic corpus statistics at the level of form representations (`FormRepresentation_Measure`). For instance, this table can record that the single genitive variant "Shakespeareja" of the masculine lexeme "Shakespeare" occurs 123 times in Gigafida 2.0.

Also, form representations tend to follow certain paradigms (as typically described in grammar books). These are managed by lexicographers and represented with pattern codes (`FormPattern`). Individual form representations can then be assigned to particular patterns (`FormRepresentation_Pattern`).

Lexeme relations (`LexemeRelation`) relate two lexemes with a particular relation type (`LexemeRelationType`). Lexeme relations are symmetric and are therefore always stored twice, once in each direction, with the same type. The business logic creates and deletes both directions together. For instance, "izkoreninjen" and "izkoreniniti" are related (in both directions) with the `word_formation` relation.

Finally, a lexeme's lemma can be explicitly associated with a subset of its form representations (`Lemma_FormRepresentation`). This normally consists of all the form representations which fall under a particular abstract word form, which is usually determined by the lexeme's category. For example, for noun lexemes, this would be the representations falling under the singular nominative word form. The lexeme's `lemma` should match one of the lexeme's orthography lemma representations.

Syntactic structures

Syntactic structures describe the structure of the canonical forms of lexical units. Each syntactic structure (`SyntacticStructure`) defines a sequence of components (`StructureComponent`), their properties, and dependencies between them. Structures can also be related to each other (`StructureRelation`). Each lexical unit falls under a particular syntactic structure. However, most of the details of syntactic structures are not stored in the SQL database, but rather in a related XML extension file (static/extensions/structures.xml). There are several reasons for this (see wiki). The format and contents of this XML extension will not be covered here.

Structures (`SyntacticStructure`) have only an id, which serves primarily to connect the SQL core and XML extension. New lexical units are normally assigned to particular syntactic structures with a dedicated pipeline. The pipeline uses a standard parser (CLASSLA) together with scripts which match the lexical unit's sequence of parts to a syntactic structure, and creates a new XML structure if necessary. Such new structures are then added to the SQL database separately.

While most of the details of syntactic structures are kept in the XML, we do also register the components in SQL (`StructureComponent`). The main reason for this is so we can efficiently access the position (`index`) of the component within the structure, which is relevant when working with `LexicalUnitParts` (ref).

We can associate structures with each other as relations (`StructureRelation`). For instance, lexicographers may want to explicitly relate two structures which are similar except that the verb is reflexive in one structure but not in the other (e.g., consider lexical units "umivati roke" in "umivati si roke").

Corpus examples

Senses of lexical units can be associated with corpus text to demonstrate real usage. Corpora (`Corpus`) are registered in the database. Examples (`Example`) always come from a particular corpus and are comprised of sentences (`ExampleSentence`). A single example can apply to different lexical units in particular senses (`Sense_Example`), and we track the tokens of those lexical units within the examples (`SenseExampleToken`). Examples can be related to each other (`ExampleRelation`).

Corpora (`Corpus`) are external resources of parsed text or speech and identified with a name and version (e.g., Gigafida 2.0).

Examples (`Example`) are a sequence of sentences from a corpus that have been chosen to exemplify one or more lexical units. In most cases, they have only one sentence, but sometimes consist of more, when more context is needed.

The sentences of an example (`ExampleSentence`) have an id internal to the corpus, and a position within the example. With the use of an external API, the id can be used to fetch the structured sentence from the corpus in TEI format. Basic statistical data (e.g., quality of example) can also be stored (`ExampleSentence_Measure`).

Senses of lexical units can be associated with a particular example (`Sense_Example`). The same example can be used for different lexical units (e.g., "Organiziral je okroglo mizo." could be an example for senses of "organizirati", "okrogla miza", "organizirati okroglo mizo", etc.).

We also track which tokens of an example represent the lexical unit (`SenseExampleToken`), which is useful when visualising examples, such as marking the lexical unit in bold. For instance, if "Organiziral je okroglo mizo." is used as an example for "okrogla miza", then in this table we would note positions 3 and 4. However, for dependent lexical units (ref), we may also want to differentiate between one of its independent lexical units and the rest. For instance, if "Organiziral je okroglo mizo." is used as an example of "organizirati okroglo mizo" and we are considering it as a collocation for "okrogla miza", then we might want to put, say, "okroglo mizo" in bold and "organizirati" in italic. For this reason, the table also includes a `SensePart` (ref): if the SensePart is of type `within_other`, then we can use it to make this distinction.

Sense translations

The data model supports storage of translations from Slovene to other languages (`Language`) for senses (`Sense`) and examples (`Sense_Example`) and may be in the form of an ordinary translation (`Translation`) and/or explanation (`Explanation`). If the translations come from external sources (`ExternalSource`), such as monolingual dictionaries of other languages, they can be associated with translations (`Translation_ExternalSource`) along with an external id.

Since translations are used for both senses and examples, a more abstract table is also used (`Translation`). A translation may be empty, in which case it should have at least one associated explanation.

Explanations (`Explanation`) are alteratives to translations, and are normally a longer description. They do not need to be in the same language as the `Translation` (this may depend, for example, on the target users).

Translations are stored for senses (`Sense_Translation`) and examples (`SenseExample_Translation`). Note that translations are not symmetric. For senses, the translation will be analogous to the string of a lexical unit in the other language (and not a particular sense of that unit). For examples, the text of the translation is stored directly in the database and the lexical unit is not marked (ref).

Sense frames

Frames provide a formal description of a verb's semantic arguments. Frames (`Frame`) have components (`FrameComponent`). The same abstract frame can be used by senses of different verbs (`Sense_Frame`), for which components may have specific subroles (`SenseFrame_Component`). As this part of the data model has not yet been used, it may well undergo further development.

In the model, frames (`Frame`) themselves are abstract elements with only ids. In addition to providing foreign keys for related tables, these ids will also be used in a new xml extension for semantic frames (analogous to ref).

Each frame has a set of frame components (`FrameComponent`). The components are of different types corresponding to semantic roles (e.g., agent, experiencer, goal).

Senses of independent lexical units can be assigned to particular frames (`Sense_Frame`). For example, a sense of `dati` might be associated with a frame which has an agent component for the giver, a patient component for the object given and a recipient component for the recipient.

For particular verbs, frame components take on more specific roles than prescribed by their component type (`SenseFrame_Component`). For instance, while agent components may generally include any kind of animate objects, a verb like "plavati" is restricted to humans and animals.

Resource connections

In addition to storing lexical units and their diverse associated data, the data model also supports the means to group (roughly speaking) subsets of this data for particular purposes (e.g., a Slovene-Hungarian dictionary portal) and assign them statuses in that context. There are registry tables for resources (`Resource`) and statuses (`Status`), and we can assign headword lexical units to resources (`LexicalUnit_Status`) for valid combinations (`Resource_Status`), as well as specifying relevant translation languages (`Resource_Language`). We can also specify if certain data under headwords should be included in a resource and with what status (`ResourceRelevance`).

The resource table (`Resource`) just registers resources with a short name or acronym (e.g., "vsms" for the Slovene-Hungarian dictionary). Each resource is typically associated with a relatively large project or particular perspective on the database (model and/or data), and lexicographers may want various imports or exports specific to that resource. There will often also be a particular portal for a particular resource (VSMS), for which a simplified resource-specific database is normally generated from the central database. If a resource is a bilingual or multilingual dictionary, then it is assumed that the source language is Slovene, and the target translation languages are explicitly stored (`Resource_Language`) (e.g., Hungarian and Serbian for a dictionary resource with translations of Slovene units in Hungarian and Serbian).

The status table (`Status`) registers the global set of string statuses that are potentially available to use for resource headwords (e.g., "manual"). The actual statuses available for a particular resource are then a subset of that (`Resource_Status`).

Independent lexical units can be "included" in a resource by assigning them a status of that resource (`LexicalUnit_Status`). Doing so effectively makes them headwords for the resource. For instance, if "miza" is assigned a particular status (e.g., "automatic") associated with the kssj resource, then lexicographers will expect that "miza" will be a headword in the (generated) Collocations Dictionary portal, perhaps marked in a particular way to signal that particular status. Note that dependent lexical units cannot be assigned to a resource in this way, because by definition they cannot be headwords but rather fall under them.

In order to specify if and how data subordinate to headwords should be included for particular resources, a special Django feature is used (generic relations, which allows us to have one table (`ResourceRelevance`), rather than a separate one for each type of subordinate data, to specify resource relevance. The content type (`content_type_id`) identifies the appropriate table and the object's id (`object_id`) specifies its id within that table. Since some types of objects (e.g., dependent lexical units) could fall under different headwords, `headword_id` is used to explicitly specify the headword (e.g., we may want to include the collocation "velika miza" under "miza" but not under "velik" for a particular resource). The `inclusion_id` and `status_id` columns specify whether the data should be included and with with what status, respectively.

However, in order to avoid the need for exhaustively adding and updating a `ResourceRelevance` row for every single piece of subordinate data under every single headword in every single resource, a set of agreed upon rules and defaults is applied. First, only specific preestablished kinds of data can be selectively included in resources - at present these are headword senses, dependent unit senses (which in practice means collocations and combinations), and sense examples. Second, the default is that headword senses are included (`ResourceRelevanceInclusion`:`include`), while dependent unit senses and sense examples are excluded (`ResourceRelevanceInclusion`:`exclude`). Third, it is assumed that a default status is defined for each resource (note that these statuses are different than the headword-levels ones in `Resource_Status`). Under these assumptions, resource relevances only need to be defined for data if it is of one of the specified types, and its inclusion and/or status are not the defaults. If the columns contain null or the default for both columns, then it is the same (in the business logic) as if the row is not included.

The `Resource` table also includes a resource called "ddd", which is interpreted as a special kind of meta-resource that includes "all data". This means that it includes every object of the preestablished kinds that can be selectively included in resources (as defined in the previous paragraph), as well as all available languages (`Resource_Language`).

Generic features

In order to enable associating objects with particular features without promoting them to columns in their tables (which may or may not be relevant for all objects in the table), the data model also provides some tables for more generic purposes. Features (`Feature`) can be defined which take on

a set of values (`FeatureValue`) and are grouped into categories (`FeatureCategory`). Values can be associated with objects (`Object_Feature`) and enter into relations (`FeatureValueRelation`). Also, playing technically separate but conceptually similar function to features, the `Measure` table registers statistical measures for data.

Every feature (`Feature`) has a name and belongs to a category (`FeatureCategory`), the combination of which must be unique. Feature categories serve to group related features. In practice, we've used categories with names that match the table that they are used with (e.g., `sense_translation` for features which are only used with `Sense_Translations`), or the generic category `general` if a feature is used with objects from multiple tables.

Feature values (`FeatureValue`) list all the allowed values (as strings) of a feature. The model does not distinguish between theoretically close-ended features (e.g., "gender") and open-ended features (e.g., "latin_name"); for the latter, extra feature values are just created as needed.

While features with their values are ultimately more or less equivalent to basic name-value pairs, they can also be related to each other if needed. A particular case of this is with labels, where there are two hierarchies of features: `label_type` and `label_value`. For instance, the `label_type` feature has a value "domain", which is related to a subset of the `label_value` feature (e.g., "zgodovina", "kemija", "organska kemija"), and these values have hierarchical relations (e.g., "kemija" -> "organska kemija").

Using the same Django generic relation mechanism as `ResourceRelevance` (ref), objects of *any* table in the core data model can then be associated with one or more values of one or more features (`Object_Feature`). This is enabled with a combination of two columns: `content_type_id` and `object_id`. `content_type_id` refers to django's `django_content_type` table, where ids (`id`) are associated with table names (`model`), while the `object_id` column specifies the id in that table. For example, if an `Object_Feature` has content_type_id=18 and object_id=71, and the `django_content_type` table has model="sense" for id=18, then the `Object_Feature` applies for the `Sense` with id=71. Note that an object can also have multiple feature values. Whether it can have more than one value of the *same* feature is controlled by the feature's `multi_valued` flag (`Feature`). For instance, an object can have several "label_value" values, but at most one "status" value.

Finally, there is a separate registry table of measures (`Measure`), which can be used to record basic statistical information for different kinds of data in particular corpora. The model supports this for senses (`Sense_Measure`), lexemes (`Lexeme_Measure`), form representations (`FormRepresentation_Measure`), example sentences (`ExampleSentence_Measure`) and sense relations (`SenseRelation_Measure`).

Other formats

A special table (`Extension`) is used for data in other formats. The columns specify a unique `name`, `format` (currently only XML is used), `string` with the string representation of the data, and `hash` for that string representation. For example, one row in our current database contains the details of the definitions of all syntactic structures, with @id attributes in synch with `SyntacticStructure` ids.

The hashes are derived using SHA512, which allows for more efficient checks if data has changed.

These extensions allow for more flexibility in data representation and prevent a proliferation of special-purpose tables and columns, but we try to keep them to a minimum so that the vast majority of the database can be queried and manipulated via standard Django and SQL operations.