A Schema Representation of Jmdict

by Stuart McGraw <jmdictdb@mtneva.com>

This note describes how JMdict data[1] is represented in the database developed for the JMdictDB project[2]. It assumes you understand the how the JMdict data is represented in the JMdict XML file[3] and the basics of relational database design.

Source code to create, load, and access a JMdict database is available at the JMdictDB project website[4]. The current implementation uses Postgresql[5], but the schema design attempts to avoid dependence on features unique to Postgresql, so the schema should be implementable using other relational database products after adjusting the scripts' syntax to the appropriate flavor of SQL.

A relationship diagram of the database is available in the file schema.png[6].

The scripts that create the database (and are thus the ultimate authority regarding the schema) are available in the source code distribution in directory pg/. The tables described in this document are created in mktables.sql. Foreign keys and indexes are defined in fkcreate.sql which is an auto-generated file created by pg/Makefile. The JMdict database also includes a number of views, functions, and other database objects that are not discussed in this paper but are defined in mkviews.sql.

Table of Contents

1. Overview

Roughly, each element type in the JMdict XML is modeled by a separate table in the database. The hierarchical structure of the XML is mirrored by the tables being linked in parent-child relationships using foreign keys in child tables that refer to the primary keys of parent tables. Table *entr* is at the top of the hierarchy and each row in *entr* corresponds to an <entry> element in the JMdict XML.

The correspondence of tables and JMdict XML elements is not absolute however; in some cases where no more than one sub-element is possible, the sub-element's information may be contained in a column of the table corresponding to the containing element. In some cases the JMdict XML semantics have been modified to better fit relational data semantics, or to allow more representational capabilities.

1.1. Parent/Child Tables and Primary/Foreign keys.

Each JMdict entry is represented by a row in table *entr* and each row has a integer field *id* containing an arbitrary but unique integer that identifies the row (and the entry). Each *entr* row contains some information about its entry, but most of the entry's information (for example readings or senses) is in other tables connected to the *entr* table by primary key / foreign key relationships.

Each of these "child" tables has a column *entr* that is a foreign key to the primary key column *id* in their parent table, *entr* (abbreviated *entr.id*). It is primary key / foreign key relations like this that tie together all the information for a single entry even though that information is distributed over multiple tables. In addition, the child tables of *entr* have another column, usually given the same name as the table itself, whose value is a small integer that serves to both disambiguate the child table's rows within the entry, and to order them. Thus, the table *sens* has two columns, *entr*, and *sens*, and all rows representing senses of a given entry will have the same value in the *entr* field, and values like 1, 2, 3, etc, in the *sens* field. The *entr* and *sens* columns together form the primary key of the *sens* table.

Some of the child tables in turn have their own child tables. The table *gloss* is a child table of *sens*. It has columns *entr* and *sens* which relate its rows to particular senses in table *sens*, and a column *gloss* to disambiguate and order the glosses within each sense. The table's primary key consists of the columns *entr*, *sens*, *gloss*.

The *entr.id* numbers are used by the database to connect rows to related rows in other tables. They can change over time and will probably be different in different database instances, and thus should never be saved outside the database as row identifiers.

The following table shows the database tables as a hierarchy with parent-child relationships denoted by indentation of the table names:

Database table name	JMdict XML element	Information
entr	<entr>	Entries
hist	<audit>	Entry change history
kanj	<ke_ele>, <keb>	Kanji text
kinf	<ke_inf>	Kanji supplementary info
rdng	<re_ele>, <reb>	Reading (kana) text
rinf	<re_inf>	Reading supplementary info
sens	<sense>, <s_info>	Sense
pos	<pos>	Part-of speech
misc	<misc>	Misc. sense info
fld	<field>	Field of use
gloss[1]	<gloss>	English and other language translations
dial	<dial>	Dialect
lsrc	<lsource>	Source language and word
There are some tables that have two parents and thus don't fit neatly into the hierarchical picture above:		
restr	<restr>	Invalid reading-kanji pairs. [parents: (*rdng*, *kanj*)]
stagr	<stagr>	Invalid sense-reading pairs. [parents: (*sens*, *rdng*)]
stagk	<stagk>	Invalid sens-kanji pairs. [parents: (*sens*, *kanj*)]
freq	<re_pri>, <ke_pri>	Frequency-of-use info for reading-kanji pairs. [parents: (*rdng*, *kanj*)]

1 The name "gloss" for this table is a misnomer but it is too much work to change at this point. In addition to glosses it contains several types of non-gloss translational entities such as literal translations. The *ginf* column value indicates the type of translational entity in each row.

xref	<xref>, <ant>	Cross-references to other entries. [parents: (*sens*, *sens*)]

Table 1: Entr database tables hierarchy

The *restr* table, for example, contains information about invalid combinations of readings and kanji and therefore needs to reference rows in both of the tables *rdng* and *kanj*. The primary keys of those two tables are (*entr,rdng*) and (*entr,kanj*) respectively. Since the referenced readings and kanji are always in the same entry, the *restr* table need only reference the entr column once, so the columns in *restr* will be (*entr,rdng,kanj*) and its foreign keys are (*entr,rdng*) and (*entr,kanj*). The *stagr* and *stagk* tables are similar.

The table *xref* also has two parents, both senses. The first is the sense the cross-reference is from, and the second, the sense (in a different entry) the cross-reference is to. Because the senses exist in different entries, the *xref* table uses the columns (*entr*,*sens*,*xentr*,*xsens*) to form the foreign keys (*entr,sens*) and (*xentr,xsens*) for the "from" and "to" senses respectively.

1.2. Keyword Tables

In the JMdict XML file, XML entities and abbreviations are used to encode reoccurring text strings. For example the entity "&io;" stands for "io", and its longer form, "irregular okurigana usage".

In the database, small integers are used for this purpose. Each set of related keywords is placed in its own table, and other tables that need to refer to a keyword store the appropriate id number. The keyword tables are all named with the prefix "kw" and we sometimes collectively refer to them as "the kw* tables". (Such tables are also sometimes called "lookup tables" in database literature.) We also use the words “keyword” and “tag” somewhat interchangeably, although the former tends to occur in the context of the database, and the latter in the context of the JMdict XML.\

Here are the contents of the *kwkinf* table, which contains the keywords used in the *kinf* table (which in turn corresponds to the <ke_inf> elements in the JMdict XML file).

```
select * from kwkinf;
id | kw | descr
---+----+--------------------------------------
1  | iK | word containing irregular kanji usage
2  | io | irregular okurigana usage
3  | oK | word containing out-dated kanji
4  | ik | word containing irregular kana usage
```

kw* tables that correspond to entities and abbreviations used in JMdict xml are: *kwdial*, *kwfld*, *kwfreq*, *kwkinf*, *kwlang*, *kwmisc*, *kwpos*, *kwrinf*. There are several kw* tables that don't correspond to anything in JMdict: *kwsrc*, *kwstat*, *kwxref*.

The keyword tables' primary function is to enforce the use of valid keyword values in other tables by means of referential integrity. For this purpose only the values in the keyword tables' *id* columns are significant. The *kw* and *descr* texts are available for the use of applications to use for display, but applications are free to map their own texts to the keyword id numbers.

The keyword tables are colored green in the schema diagram.

1.3. Keyword Lists

Some information in JMdict and the database is essentially lists of keywords. For example, a kanji text might have one of more keywords from the *kinf* table attached to it. These lists are stored in tables that have a foreign key to the table with the elements the list is attached to, and a foreign key to the appropriate keyword table. For example table *kinf* holds the ke_inf tags for the kanji elements. It has columns (*entr, kanj, ord, kw*). *entr* and *kanj* together identify a specific kanji element that a keyword applies to, and *kw* identifies the appropriate ke_inf keyword in table *kwkinf*. The *ord* column contains a small integer and is used to maintain the order of the items in the list.

A few keyword list tables have some other information in addition to the keywords. The table *freq* for example contains freq keywords, and also a numeric value associated with each.

The tables that contain such keyword lists are: *dial*, *fld*, *freq*, *kinf*, *misc*, *pos*, *rinf*.

Not all references to the kw* tables occur from these keyword list tables though. In some cases where only one instance of a tag is possible, a reference may occur from one of the other tables. For example, the *gloss* table contains a column *lang* that references *kwlang*, This is appropriate since a gloss can never have more than one language tag.

The use of integers rather that the short text strings for keywords might seem cumbersome initially, but in practice, one quickly learns the equivalences. This separation allows the short and long text, which is really only relevant for display to users, to change, while the numeric id values, relevant to application code, remain invariant. Application code seldom needs to execute joins with the kw* tables, because they will usually read the keyword tables into memory at startup and resolve numeric id's to strings (and the reverse) programmatically using the in-memory tables. Finally, if one really does want to work with the text strings in joins, it is easy to define views on the keyword list tables that have the kw* tables pre-joined and present an additional column containing the keyword text.

The keyword list tables are colored yellow in the schema diagram.

1.4. Restrictions: restr, stagr, stagk

In the JMdict XML file, these elements describe restrictions on the use of reading-kanji, sense-reading, and sense-kanji (respectively) pairs by enumerating the *valid*

pairs.

In the database, those restrictions are given by listing *invalid* pairs. Although this creates an impedance mismatch when displaying such information (which will generally be displayed as valid combinations), it is more consistent, simplifies queries that incorporate this information, and eliminates the need for explicitly representing related information such as the <re_nokanji> element.

The restriction tables are colored light blue in the schema diagram.

1.5. ke_pri, re_pri Generalization

The ke_pri and re_pri elements in the JMdict XML are used to convey information about frequency-of-use metrics using values such as "ichi1", "ichi2", "ns14", etc. Rather than mimicking JMdict's keyword based approach, the database schema generalizes the notion of frequency-of-use by using a scale indicator such as "ichi" or "nf", and a metric such as "1", "14", etc. Thus "ichi1" is replaced by the keyword, value pair ("ichi",1) and "nf14" by ("nf",14). This allows for the inclusion of other finer-grained metrics in the future such as page counts from Internet search engines like Google.

Most ke_pri and re_pri elements in JMdict xml occur as equi-valued pairs, that is there might be a ke_pri element with the value "nf32" on a particular kanji element, and a re_pri tag also with the value "nf32" on a particular reading element. The equal values imply that the frequency-of-use indicator refers to the tagged kanji and reading as a pair, not to each independently. There are some ke_pri or re_pri that are not paired and they apply to the kanji or reading alone.

In the database we represent ke_pri and re_pri elements in table *freq*. Each row contains columns *rdng* and *kanj* which references a reading and a kanji, thus a single row represents a reading-kanji pair. Either (but not both) may be NULL should the value apply to a reading or kanji alone. Two other columns are *kw* which identifies the scale (by referring to table *kwfreq*), e.g. "nf", and *value* that gives the metric, e.g. 34, on the scale.

1.6. Cross-references

In the JMdict XML file, there are two kinds of cross references, <xref> ("see also") and <ant> ("antonym"). Both are located within senses, and specify the target of the cross-ref as a kanji or reading text string. This effects sense→entry cross references, and, since there is no guarantee that a kanji or reading string will uniquely identify a single entry, may also be a one-to-many reference.

In the database both (and possibly other) kinds of cross references are modeled in table *xref*. As in JMdict XML, cross references occur within senses, which is represented in the *xref* table using columns *entr* and *sens* to identify the originating sense. Since the order of xrefs is significant, the column *xref* orders the xrefs in a sense and the columns *entr*,*sens*,*xref* constitute the table's primary key. Differing

from JMdict XML, the *xref* table gives the target of the cross-reference as a another entry-sense pair, *xentr* and *xsens* which results in each cross-reference pointing to a specific sense in one specific entry rather than an entire entry (or set of entries). If a cross-reference to multiple senses or multiple entries is wanted, multiple rows in the xref table are used. Table *xref* also has a column *typ* that gives the type of cross-ref, and whose values are defined in table *kwxref*.

It is often desirable to display a cross-reference using a reading and kanji other than the entry's first reading and kanji. The columns *rdng* and *kanj* point to the most appropriate values for display.

The cross reference table is colored purple in the schema diagram.

Unresolved cross-references

There is no guarantee that an an entry matching an XML <xref> or <ant> element exists in the XML file or database. This is often the case when loading XML data into the database when the entry referred to has not been read yet. Nor is there any guarantee that an <xref> or <ant> element uniquely identifies a single entry – it could match several.

The database schema therefore provides a table, *xresolv*, which is used for "unresolved" cross references into which cross reference items are inserted when initially loading XML data. After the data has been loaded, a second program, xresolv.py, is run which will attempt to identify a single entry that each row in *xresolv* refers to, create one or more rows in table xref pointing to it, and delete the row from table *xresolv*. Unresolvable (because there are no or multiple matching entries) xrefs are left in table *xresolv* after xresolv.py completes.

1.7. Corpora and Sequence Numbers

As mentioned, each entry (row in table *entr*) has a unique id number that is (usually) automatically assigned when an entry is created and uniquely identifies the entry within the database. Entries are also identified by their corpus (*entr.src*) and sequence number (*entr.seq*). This pair though may not be unique since there may be several such entries when edited copies of an entry are created. However, there will be at most one such entry that is "approved" (has a *entr.unap* value that is False) and normally only approved entries are exported into XML.

The corpora to which entries are assigned are defined in table *kwsrc*. Each corpus has some other attributes such as a short abbreviation (*.kw*), a long description (*.descr*) and a "type" (*.srct*) which is a reference to table *kwsrct* and associates corpora into groups that have certain shared requirements for things like display or editing.

Sequence numbers are automatically assigned when an *entr* row is created without an explicit seq number assigned. This is done by Postgresql trigger. Each corpus has its own sequence number generator that is implemented as a Postgresql sequence.[2] The

2 A Postgtresql sequence is an object that supplies sequential numbers when one is requested. It is

trigger will look in *kwsrc.seq* for the name of the Postgresql sequence to use to get the next available sequence number for the corpus the entry belongs to.

There is also a trigger on the *kwsrc* table that will automatically create the sequence for a corpus when a new corpus is added by inserting a new row in table *kwsrc*. The sequence will be created with minimum, maximum and increment values corresponding to the *.smin*, *.smax* and *.sincr* values in the new *kwsrc* row, Note that these values are only used when creating the sequence (which in turn happens only when a new row is inserted in *kwsrc*); changes made to these values after the sequence was created will have no effect. The sequence will be named "seq_" followed by the string given in *kwsrc.seq* something. When a *kwsrc* row is deleted the trigger will automatically delete the associated sequence.

1.8. XML Round-tripping

The module "fmtxml.py" will generate XML from an entry object read from the database. There may be differences between this regenerated XML and the original JMdict file XML that produced the database entry. The following are things I am aware of that may differ:

- Comments in the XML are not preserved so cannot be regenerated.[3]

- Element order may vary. Although elements are generated in the order defined by the JMdict DTD, when there are multiple successive elements of the same type, order may not be the same as in the original XML in some cases. The following elements will retain the original order: k_ele, r_ele, sense, gloss, k_inf, r_inf, pos, misc, field, dial, lsource, audit. The order of re_restr, stagr, stagk, ke_pri, and re_pri elements may differ. Entry elements in the current JMdict are in seq number order and can be regenerated in that order. Were that order to change though, it would not be preserved in the regenerated XML.

- When a sense without any "pos" (part-of-speech) tags follows a sense with pos tags, the pos tags on the preceding sense are assigned to the following sense. Thus regenerated XML may have pos tags in senses that didn't have them in the source XML.

- Regenerated XML takes advantage of default values when possible. If the source XML redundantly specifies a default value (xml:lang="eng" for example) it will not be reproduced in the regenerated XML.

---

not related to (other than being used to generate them) the "sequence" numbers used in JMdictDB database entries.

3 Prior to April 2008, comments were parsed and those that matched patterns for "merged entry" comments were turned into actual deleted entries in the database. However, recent "merged entry" comments identify the mergee by kanji rather than sequence number and will require a post-load pass to resolve, similar to the way xrefs are processed. The program for this has not been written yet (as of May 2008) and currently, all comments are effectively ignored.

- An xref in the database always refers to a specific target entry, and a single sense in that entry. An xref in the XML may or may not specify a target entry's sense. When the target entry has multiple senses, but the XML fails to give any senses, the XML importer will generate multiple xref's, one to each target sense. If the XML exporter sees a set of xrefs pointing to every sense of a target entry, it will assume the input XML did not specify any senses and generate an <xref> element with no sense numbers. However the source XML may have explicitly listed every sense number of the target entry. This is a particular problem when the number of target senses is one. (That is, xrefs of the form <xref>供・とも</xref> and <xref>供・とも・(1)</xref> will both result in <xref>供・とも</xref> when the XML is regenerated, assuming the target entry has only one sense.)

- Duplicate tags are filtered out during importing from XML and will not be produced in the regenerated XML. In particular, some JMdict entries have multiple <k_pri> or <re_pri> tags.

- When <ke_pri> or <re_pri> conflict, that is, there are two tags with the same scale (text value) but different numerical values (e.g. “nf09” and “nf14”, or “ichi1” and “ichi2”) that apply to the same reading and kanji pair, the one with the lower value is used in the database, the one with the higher value is discarded and thus won't appear in the regenerated XML. Note this only applies to freq values associated with the same reading, kanji pair: if R1 has “nf09”, “nf14”, K1 has “nf09”, and K2 has “nf14”, nothing is discarded because the two freq values are associated with different pairs (R1,K1, and R1,K2).

- The database can represent information that currently has no means of representation in XML with the standard JMdict DTD. Specifically:

  - Some tables provide for additional information such as *notes, status, parent* in the *entr* table, or *email, refs, notes* in the *hist* table (maps to the <audit> XML element) that JMdict does not support.
  - Cross-references refer to a specific entry even when there multiple references with the same reading or kanji text.
  - Cross-references may have more types that just the “see also” and “antonym” types available in the current JMdict XML.
  - Readings can have sound clips.
  - There may be keyword values in the kw* tables and used in the database that do not have equivalent entities defined in the JMdict DTD.

  If entries are created that make use of these capabilities, the additional information will not be representable in a regenerated JMdict XML file.

2. An Example Entry

This section looks at the actual table data used to store a JMdict entry, using data extracted from a live database.

Here is entry 1211370 as displayed by wwwjdic:

> 堪能(P); 勘能 【たんのう(堪能)(P); かんのう(ok)】 (adj-na) (1) proficient; skillful; (n,vs) (2) satisfaction; (n,vs) (3) {Buddh} fortitude; (P)

And here is its representation in the JMdictDB database:

Table *entr* row:

```
select * from entr where seq=1211370;
id    | src | stat | seq      | dfrm | unap | notes
------+-----+------+----------+------+------+-------
20894 | 1   | 2    |21211370 |      | F    |
```

id is an arbitrary number as assigned when the entry was created. Its purpose is to provide an anchor that related rows in other tables can refer to. Since it may change over time and may be different in different database instances, it should never be saved externally as a means of identifying an entry; use *seq* for that.

src says what collection of entries ("corpus") this entry belongs to and refers to table *kwsrc*.*id*. *id*=1 in *kwsrc* says this entry is part of JMdict. Currently in the EDRDG implementation of JMdictDB all entries are either JMdict or JMnedict entries but in other database implementations, one could find other corpora such as the Totoeba "examples" sentences, Kanjidic2 entries or custom site-specific corpora.

stat gives that status of this entry. It refers to table *kwstat* and *id*=2 in that table is "active". There are other *stat* values for new, deleted, and rejected entries.

seq is the same as the <ent_seq> element in JMdict XML file and identifies each JMdict entry across time and space.

dfrm and *unap* are used for managing edits and new submissions.

notes is a text field that can contain arbitrary information pertaining to an entry that is intended to be displayed to users.

Table *rdng* rows

```
select * from rdng where entr=20894;
entr  | rdng | txt
------+------+----------
20894 | 1    | たんのう
20894 | 2    | かんのう
```

entr identifies the entry the row belongs to. *rdng* disambiguates and orders multiple

readings belonging to the same entry. *entr* and *rdng* together constitute the primary key and any other tables that refers to specific readings (such as *restr* will contain a foreign key to these two columns. *txt* is the reading text. It is expected to have at least one kana character and should not contain any kanji characters.

Table *kanj* rows

```
select * from kanj where entr=20894;
entr  | kanj | txt
------+------+------
20894 | 1    | 堪能
20894 | 2    | 勘能
```

This is structured like table *rdng*. Kanji text is expected to contain at least one kanji character.

Table *sens* rows

```
select * from sens where entr=20894;
entr  | sens | notes
------+------+-------
20894 | 1    |
20894 | 2    |
20894 | 3    |
```

There is a *sens* row for each sense in the entry. *entr* says which entry the sense belongs to, *sens* disambiguates and orders the senses. *entr* and *sens* together constitute the primary key. *notes* contains an optional text providing some information unique to this sense that is intended to be displayed to users.

Table *gloss* rows

```
select * from gloss where entr=20894;
entr  | sens | gloss | lang | txt
------+------+-------+------+--------------
20894 | 1    | 1     | 1    | proficient
20894 | 1    | 2     | 1    | skillful
20894 | 2    | 1     | 1    | satisfaction
20894 | 3    | 1     | 1    | fortitude
```

In this table we see the first two rows are glosses for the first sense, the second row is a gloss for second sense, and the third row a gloss for the third sense. The *lang* values are all 1, indicating english (from table *kwlang*). Because this database instance was loaded from a JMdict_e file, there are no non-english glosses in it. Had it been loaded from the full JMdict file, that would not be the case.

Table *pos* rows

```
select * from pos where entr=20894;
entr  | sens | ord | kw
------+------+-----+----
20894 | 1    | 1   | 2
20894 | 2    | 2   | 17
20894 | 2    | 3   | 46
20894 | 3    | 4   | 17
20894 | 3    | 5   | 46
```

The *pos* table contains a list of part-of-speech keywords for each sense of an entry. Again, *entr* identifies the entry and *sens* the sense. *kw* refers to the *id* column of table *kwpos*. Looking in that table we see the 2 is "adj-na", 17 is "n", and 46 is "vs":

```
select * from kwpos where id in (2,17,46);
id |   kw   |                      descr
---+--------+-----------------------------------------------
 2 | adj-na | adjectival nouns or quasi-adjectives (keiyodoshi)
17 | n      | noun (common) (futsuumeishi)
46 | vs     | noun or participle which takes the aux. verb suru
```

Table *fld* rows

```
select * from fld where entr=20894;
entr  | sens | ord | kw
------+------+-----+----
20894 | 3    | 1   | 1
```

fld is a keyword list table like *pos*: *entr* identifies the entry and *sens* the sense. *kw* refers to the *id* column of table *kwfld*. Looking in that table we see the 1 is "Buddh".

Table *rinf* rows

```
select * from rinf where entr=20894;
entr  | rdng | ord | kw
------+------+-----+----
20894 | 2    | 1   | 3
```

The *rinf* table contains a list of <re_inf> keywords for each reading of an entry. *entr* identifies the entry and *rdng* the reading. *kw* refers to the *id* column of table *kwrinf*. In this case, *kw*=3 indicates the rinf keyword, "ok" ("out-dated or obsolete kana usage"). It applies only to reading 2 (かんのう).

The entry we are examining here has no data in the *kinf* table, but that table is the same except its second column is *kanj* rather than *rdng* and the kw column references table *kwkinf* rather than *kwrinf*.

Table *restr* rows

```
select * from restr where entr=20894;
entr  | rdng | kanj
------+------+------
20894 | 1    | 2
```

Contains <re_restr> element info used when not all combinations of reading and kanji

are valid. *entr* identifies the entry, *rdng* and *kanj* identify a reading and kanji pair that are *not valid* together. Note that this is the opposite of the JMdict XML where re_restr elements identify *valid* pairs. Since only the pair たんのう / 勘能 is listed as invalid, the combinations たんのう / 堪能, かんのう / 堪能, and かんのう / 勘能 are valid.

There is no special indicator for the "re_nokanji" tag that exists in JMdict XML -- simply having *restr* rows for every kanji in an entry paired with one of the readings in sufficient to indicate that reading is "nokanji".

It is the application 's responsibility to generate “valid” restr pairs or the “nokanji” flag from the existing data if that is how the information is to be presented.

Table *freq* rows

```
select * from freq where entr=20894;
entr  | rdng | kanj | kw | value
------+------+------+----+-------
20894 | 1    | 1    | 5  | 16
20894 | 1    | 1    | 7  | 1
```

Table freq contains frequency-of-use information found in JMdict in the re_pri and ke_pri elements. As mentioned earlier, a frequency of use datum usually applies to a reading-kanji pair, but if not, one of the columns *rdng* or *kanj* may be NULL. This will be the case for entries without kanji text, and of frequency metric derived from purely lexicographic means such as from Internet search engine page counts. The *kw* column refers to the table *kwfreq:*

```
jmdict=# select * from kwfreq where id in (5,7);
 id |  kw  | descr
----+------+-------
  5 | nf   |
  7 | news |
```

So the data shown tells us that the kanji-1/reading-1 pair (堪能 / たんのう) has the frequency-of-use tags, “nf16” (5,16) and “news1” (7,1).

This entry has no dialect, source language, misc info, kinf info, etc, so a queries like the following will return 0 rows:

```
select * from dial where entr=20894;
entr | sens | ord | kw
-----+------+-----+----
(0 rows)
```

3. Using SQL

This section describes how the table structures and relations previously discussed affect the SQL you write to find and extract data.

3.1. Retrieving an Entry's Data

If you are writing SQL as part of an application, extraction of data is easy in most cases. That is because there is a library API that you can use that will extract all the data needed and construct an object representation of the entry in application code. In this case the library API will take care of the SQL for you, if you know the entries you want. In the JMdictDB code, the Python library module jdb.py contains function entrList() which does this.

Even if you need to write the SQL yourself (perhaps because you are creating such an API function) it is quite simple. Because all the tables have a column *entr* you need to execute a set of SQL statements, one for each table, with a WHERE clause that specifies the the *entr* values you are interested in. For example (in Perl) assuming that the variables $id1 and $id2 contain the entry id numbers of two entries you are interested in:

```
$dbh   = DBI->connect("dbi:Pg:dbname=”jmdict”, PrintWarn=>0, RaiseError=>1)
$dbh->{pg_enable_utf8} = 1;
$sth   = $dbh->execute (“SELECT * FROM entr WHERE id IN (?,?)”, [$id1,$id2]);
$entr  = $sth->fetchall ();
$sth   = $dbh->execute (“SELECT * FROM kanj WHERE entr IN (?,?)”, [$id1,$id2]);
$kanj  = $sth->fetchall ();
$sth   = $dbh->execute (“SELECT * FROM kinf WHERE entr IN (?,?)”, [$id1,$id2]);
$kinf  = $sth->fetchall ();
$sth   = $dbh->execute (“SELECT * FROM kfreq WHERE entr IN (?,?)”, [$id1,$id2]);
$kfreq = $sth->fetchall ();
$sth   = $dbh->execute (“SELECT * FROM rdng WHERE entr IN (?,?)”, [$id1,$id2]);
$rdng  = $sth->fetchall ();
$sth   = $dbh->execute (“SELECT * FROM sens WHERE entr IN (?,?)”, [$id1,$id2]);
$sens  = $sth->fetchall ();
[...]
```

or in Python (using the Psycopg2 Postgresql adapter):

```
dbh    = conn = psycopg2.connect (database=”jmdict”)
psycopg2.extensions.register_type(psycopg2.extensions.UNICODE)
cursor = dbh.cursor()
cursor.execute (“SELECT * FROM entr WHERE id IN (?,?)”, [id1,id2])
entr   = cursor.fetchall ()
cursor.execute (“SELECT * FROM kanj WHERE entr IN (?,?)”, [id1,id2])
kanj  = cursor.fetchall ()
cursor.execute (“SELECT * FROM kinf WHERE entr IN (?,?)”, [id1,id2])
kinf  = cursor.fetchall ()
cursor.execute (“SELECT * FROM kfreq WHERE entr IN (?,?)”, [id1,id2])
kfreq = cursor.fetchall ()
cursor.execute (“SELECT * FROM rdng WHERE entr IN (?,?)”, [id1,id2])
$dng  = cursor.fetchall ()
cursor.execute (“SELECT * FROM sens WHERE entr IN (?,?)”, [id1,id2])
sens  = cursor.fetchall ()
[...]
```

Having gotten all the needed rows for the entries $id1 and $id2 in variables $entr, $kanj, etc., you can now use that data to build the objects representing the two entries.

3.2. Finding Entries

When you need to find information, the SQL becomes more complex. The basic rule is that you need to join together those tables that have columns that are part of your search criteria. Fortunately, because the relationship hierarchy is simple and fixed, this too is usually quite straight forward. Since you will usually want to know which entries meet the criteria, you will be interested in getting the *entr.id* values. In many cases you may not even need a join.

For example, to find the entries that have a reading, "つける", the following is sufficient:

```
SELECT entr FROM rdng WHERE txt='つける';
```

Of course, if the criteria include information that is in different tables, a join or some other combination of the tables is inevitable. To find the entries that contain both "つけ" and "漬", and have a sense with a PoS that is a noun:

```
SELECT DISTINCT r.entr
  FROM rdng r
  JOIN kanj k ON k.entr=r.entr
  JOIN pos p ON p.entr=k.entr
  WHERE r.txt LIKE '%つけ%'
    AND k.txt LIKE '%漬%'
    AND p.kw = 17;
```

"17" is the *id* number of the "noun" keyword in table *kwpos*.

If you need more than just the *entr.id*, say the entry's sequence number, then the *entr* table will also need to be included:

```
SELECT DISTINCT e.seq
  FROM entr e
  JOIN rdng r ON r.entr=e.id
  JOIN kanj k ON k.entr=r.entr
  JOIN pos p ON p.entr=k.entr
  WHERE r.txt LIKE '%つけ%'
    AND k.txt LIKE '%漬%'
    AND p.kw != 17;
```

It is noteworthy that the *sens* table was not needed in the above two queries, even though it is lies between *entr* and *pos* in the table hierarchy.

3.3. Views

The scripts that build that database include mkviews.sql which creates a number of useful views and functions, including ones to provide a (somewhat simplified) edict-ish textual summary of an entry, list valid and invalid reading-kanji combinations based on table *restr*, present meta-information about entries such as the number of kanji, readings and senses each has, and others. They are currently still in too much

flux to document yet, but will be included in a future version of this paper.
In the meantime, the comments in pg/mkviews.sql may be useful.

4. Annotated Schema

The Postgresql DDL statements below will create the tables described above. They are based on revision 1.29 (2007/09/09) of mktables.sql. As commonly happens, this documentation may lag behind the code and so should not be taken as definitive.

```
CREATE TABLE entr (
    id SERIAL NOT NULL PRIMARY KEY,
      -- Arbitrary and unique integer to identify entries within
      -- the database.  We don't want to use seq as the PK because
      -- the database may include entries from non-JMdict sources
      -- (such as JMnedict) in the future and those source may not
      -- have seq numbers.
      -- All rows in other tables related to an entry will have a
      -- foreign key pointing to the entry's id value.
    src SMALLINT NOT NULL,
      -- Indicates where this entry came from (1=jmdict).
      -- References kwsrc.
    stat SMALLINT NOT NULL,
      -- References table kwstat and indicates the status of this
      -- entry.  Typical values are "new", "active", "deleted",
      -- "rejected", etc.
    seq INT NOT NULL,
      -- Sequence number.  This number maps to the <ent_seq>
      -- element of the JMdict XML file, and is intended
      -- to be a stable identifier of a particular word.  (The
      -- id column is *not* suitable for uses spanning either
      -- time or space.)
    dfrm INT,
      -- References another entry that was the edit source for
      -- this entry.
    unap BOOLEAN NOT NULL,
      -- If TRUE, this entry has not been approved by an editor.
    srcnote VARCHAR(255),
      -- Additional ad-hoc information about the source of this
      -- entry.
    notes TEXT
      -- Arbitrary textual information about this entry.  Will be
      -- displayed by applications.
     );

CREATE SEQUENCE seq
      -- This is a sequence for getting new entr.seq values.  It is
      -- not entr.seq's default value and applications are expected
      -- to get new seq values and apply them when needed.
      -- MAXVALUE is <9000000 in order not to intrude of the space
      -- reserved for the jis212 entries.
    INCREMENT 10 MINVALUE 1000000 MAXVALUE 8999999
      NO CYCLE OWNED BY entr.seq;

CREATE TABLE rdng (
      -- Defines the reading (or other kana representation)
      -- of an entry.  All entries are expected to have at
      -- least one reading row.  Maps to the JMdict XML element
      -- <re_ele>.
    entr INT NOT NULL,
      -- FK to entr.id to identify the entry that this reading
      -- belongs to.
    rdng SMALLINT NOT NULL,
      -- Disambiguates and orders multiple readings in an entry.
    txt VARCHAR(2048) NOT NULL,
```

```
      -- The reading text.  Should contain at least one kana
      -- character and should not contain any kanji characters.
      -- Maps to the JMdict XML element <reb>.
    PRIMARY KEY(entr,rdng));

CREATE TABLE kanj (
      -- Defines a kanji text associated with an entry.  An entries
      -- may not have any kanji rows.  Maps to the JMdict XML element
      -- <ke_ele>.
    entr INT NOT NULL,
      -- FK to entr.id to identify the entry that this kanji
      -- string belongs to.
    kanj SMALLINT NOT NULL,
      -- Disambiguates and orders multiple kanji in an entry.
    txt VARCHAR(2048) NOT NULL,
      -- The kanji text.  Should contain at least one kani
      -- character.  Maps to the JMdict XML element <keb>.
    PRIMARY KEY(entr,kanj));


CREATE TABLE sens (
      -- Represents an individual sense in an entry.  Maps to the
      -- the JMdict XML element <sense>.
    entr INT NOT NULL,
      -- FK to entr.id to identify the entry that this sense
      -- belongs to.
    sens SMALLINT NOT NULL,
      -- Disambiguates and orders multiple senses in an entry.
    notes TEXT,
      -- Arbitrary additional textual information pertaining to
      -- this sense.  Maps to the JMdict XML <s_inf> element.
    PRIMARY KEY(entr,sens));

CREATE TABLE gloss (
      -- Represents a gloss in a sense.  Maps to the JMdict XML
      -- <gloss> element.
    entr INT NOT NULL,
      -- FK to entr.id to identify the entry this gloss belongs to.
    sens SMALLINT NOT NULL,
      -- FK to sens.sens to identify the sense this gloss belongs to.
    gloss SMALLINT NOT NULL,
      -- Disambiguates and orders multiple glosses in a sense.
    lang SMALLINT NOT NULL,
      -- FK to kwlang.id.  Language of this gloss.
    ginf SMALLINT NOT NULL,
      -- FK to kwginf.id.  Type of gloss.
    txt VARCHAR(2048) NOT NULL,
      -- The gloss text.
    PRIMARY KEY(entr,sens,gloss));

CREATE TABLE xref (
      -- Models various types of cross-references between entries (or
      -- more accurately, between senses of entries.)  Maps approximately
      -- to JMdict XML elements <xref> and <ant>.
    entr INT NOT NULL,
      -- FK to entr.id.  Identifies the entry this cross-reference
      -- belongs to.
    sens SMALLINT NOT NULL,
      -- FK to sens.id.  Identifies the sense this cross-reference
      -- belongs to.
    xref SMALLINT NOT NULL,
      -- Disambiguates and orders multiple xrefs in a sense.
    typ SMALLINT NOT NULL,
      -- FK to kwxref.id.  Identifies the type of cross reference.
```

```
    xentr INT NOT NULL,
      -- FK to entr.id.  Identifies the entry this cross-reference
      -- points to.
    xsens SMALLINT NOT NULL,
      -- FK to sens.id.  Identifies the sense this cross-reference
      -- points to.
    rdng SMALLINT,
      -- FK to rdng.rdng.  Identifies the reading to use when displaying
         this cross-reference.
    kanj SMALLINT,
      -- FK to kanj.kanj.  Identifies the kanji to use when displaying
         this cross-reference.
    notes TEXT,
      -- Additional textual information pertaining to this cross-
      -- reference for display to user.
    PRIMARY KEY (entr,sens,xref));


CREATE TABLE hist (
      -- This table maintains a series of history records for each
      -- entry that documents significant changes to the entry.
      -- Maps roughly to JMdict element <audit>.
    entr INT NOT NULL,
      -- FK to entr.id.  Identifies the entry this history record
      -- belongs to.
    hist SMALLINT NOT NULL,
      -- Disambiguates and orders multiple hist records with an entry.
    stat SMALLINT NOT NULL,
      -- FK to kwstat.id.  Status of entry after the change this
      -- record documents was made.
    unap BOOLEAN NOT NULL DEFAULT TRUE,
      -- Approval status of entry after this change was made.
    dt TIMESTAMP NOT NULL DEFAULT NOW(),
      -- Date and time changes were made.
    userid VARCHAR(20),
      -- If user making change was a logged in editor, this is that
      -- editor's authenticated userid.
    name VARCHAR(250),
      -- Unauthenticated name of user who made the changes.
    email VARCHAR(250),
      -- Unauthenticated email address of user who made the changes.
    diff TEXT,
      -- Text string that documents the difference between the original
      -- and changed entries.
    Refs TEXT,
      -- Verifiable public references supporting this change.
    notes TEXT
      -- Additional textual information pertaining to this change.
      );

CREATE TABLE xresolv (
      -- This table is used only during loading the database from
      -- a JMdict XML file.  It holds xref strings prior to resoling
      -- them to actual entry/senses.
    entr INT NOT NULL,
    sens SMALLINT NOT NULL,
    ord SMALLINT NOT NULL,
    typ SMALLINT NOT NULL,
    rtxt VARCHAR(250),
    ktxt VARCHAR(250),
    tsens SMALLINT,
    notes VARCHAR(250),
    prio BOOLEAN DEFAULT FALSE,

CREATE TABLE freq (
```

```
      -- Contains frequency-of-use items from the JMdict XML <ke_pri>
      -- <re_pri> elements, and other sources.
    entr INT NOT NULL,
      -- FK to entr.id.  Identifies the entry this datum is associated
      -- with.
    rdng SMALLINT NULL,
      -- FK to rdng.rdng.  Identifies the reading item that this datum
      -- applies to.  May be NULL if it applies to only a kanji element.
    kanj SMALLINT NULL,
      -- FK to kanj.kanj.  Identifies the kanji item that this datum
      -- applies to.  May be NULL if it applies to only a reading element.
    kw SMALLINT NOT NULL,
      -- FK to kwfreq.id.  Identifies the FoU scale in table kwfreq.
    value INT,
      -- The value of the datum on the scale identified by kw.
    UNIQUE (entr,rdng,kanj,kw),
    CHECK (rdng NOTNULL OR kanj NOTNULL));

CREATE TABLE dial (
      -- Provides lists of dialects for senses.  This information
      -- corresponds to the JMdict XML <dial> element.
    entr INT NOT NULL,
      -- FK to entr.id.  Identifies entry that dialect applies to.
    sens SMALLINT NOT NULL,
      -- FK to sens.sens.  Identifies the sense that field applies to.
    ord SMALLINT NOT NULL,
      -- Maintains order of keywords.
    kw SMALLINT NOT NULL,
      -- FK to kwdial.id.  Identifies the dialect keyword.
    PRIMARY KEY (entr,sens,kw));

CREATE TABLE fld (
      -- Provides lists of fields for entries.  This information
      -- corresponds to the JMdict XML <field> element.
    entr INT NOT NULL,
      -- FK to entr.id.  Identifies entry that field applies to.
    sens SMALLINT NOT NULL,
      -- FK to sens.sens.  Identifies the sense that field applies to.
    ord SMALLINT NOT NULL,
      -- Maintains order of keywords.
    kw SMALLINT NOT NULL,
      -- FK to kwfld.id.  Identifies the field keyword.
    PRIMARY KEY (entr,sens,kw));

CREATE TABLE kinf (
      -- Provides lists of kanji info tags for kanji items.  This
      -- information corresponds to the JMdict XML <ke_inf> element.
    entr INT NOT NULL,
      -- FK to entr.id.  Identifies entry that the kinf tag applies to.
    kanj SMALLINT NOT NULL,
      -- FK to kanj.kanj.  Identifies the kanji item that kinf tag
      -- applies to.
    ord SMALLINT NOT NULL,
      -- Maintains order of keywords.
    kw SMALLINT NOT NULL,
      -- FK to kwkinf.id.  Identifies the kanji info keyword.
    PRIMARY KEY (entr,kanj,kw));

CREATE TABLE lsrc (
      -- Provides the foreign language word and language for each
      -- sense for imported words.  This information corresponds to
      -- the JMdict XML <lsource> element.
```

```
    entr INT NOT NULL,
      -- FK to entr.id.  Identifies entry that field applies to.
    sens SMALLINT NOT NULL,
      -- FK to enrt.sens.  Identifies the sens item that lsrc tag
      -- applies to.
    ord SMALLINT NOT NULL,
      -- Maintains order of keywords.
    lang SMALLINT NOT NULL,
      -- FK to kwlang.id.  Identifies the language keyword.
    txt VARCHAR(250)
      -- The word in language 'lang' from which this word sense
      -- was derived.
    part BOOLEAN,
      -- When FALSE, indicates the entire Japanese word/sense was
      -- derived from this source word.  When TRUE, only a part of
      -- the Japanese word was derived from the source word.
    wasei BOOLEAN,
      -- When FALSE, this source word or phrase is a valid word
      -- or phrase with currency in its language.  When TRUE, this
      -- source word does not have currency in its language.
    PRIMARY KEY (entr,sens,kw,txt));

CREATE TABLE misc (
      -- Provides lists of misc tags for senses.  This information
      -- corresponds to the JMdict XML <misc> element.
    entr INT NOT NULL,
      -- FK to entr.id.  Identifies entry that misc tag applies to.
    sens SMALLINT NOT NULL,
      -- FK to sens.sens.  Identifies the sense that misc tag applies to.
    ord SMALLINT NOT NULL,
      -- Maintains order of keywords.
    kw SMALLINT NOT NULL,
     -- FK to kwmisc.id.  Identifies the misc keyword.
    PRIMARY KEY (entr,sens,kw));

CREATE TABLE pos (
      -- Provides lists of pos (part-of-speech) tags for senses.
      -- This information corresponds to the JMdict XML <pos> element.
    entr INT NOT NULL,
      -- FK to entr.id.  Identifies entry that pos tag applies to.
    sens SMALLINT NOT NULL,
      -- FK to sens.sens.  Identifies the sense that pos tag applies to.
    ord SMALLINT NOT NULL,
      -- Maintains order of keywords.
    kw SMALLINT  NOT NULL,
     -- FK to kwpos.id.  Identifies the pos keyword.
    PRIMARY KEY (entr,sens,kw));

CREATE TABLE rinf (
      -- Provides lists of reading info tags for reading items.  This
      -- information corresponds to the JMdict XML <re_inf> element.
    entr INT NOT NULL,
      -- FK to entr.id.  Identifies entry that the rinf tag applies to.
    rdng SMALLINT NOT NULL,
      -- FK to rdng.rdng.  Identifies the reading item that rinf tag
      -- applies to.
    ord SMALLINT NOT NULL,
      -- Maintains order of keywords.
    kw SMALLINT NOT NULL,
      -- FK to kwrinf.id.  Identifies the reading info keyword.
    PRIMARY KEY (entr,rdng,kw));
```

```
CREATE TABLE restr (
      -- Lists invalid pairs of reading and kanji elements.
      -- Note that this is the inverse of the representation in JMdict
      -- XML file where <restr> elements list the _valid_ pairs.
    entr INT NOT NULL,
      -- FK to entr.id.  Identifies entry that the restriction applies to.
    rdng SMALLINT NOT NULL,
      -- FK to rdng.rdng.  Identifies the reading within the entry that
      -- the restriction applies to.
    kanj SMALLINT NOT NULL,
      -- FK to kanj.kanj.  Identifies the kanji within the entry that the
      -- restriction applies to.
    PRIMARY KEY (entr,rdng,kanj));

CREATE TABLE stagr (
      -- Lists invalid pairs of sense and reading elements.
      -- Note that this is the inverse of the representation in JMdict
      -- XML file where <stagr> elements list the _valid_ pairs.
    entr INT NOT NULL,
      -- FK to entr.id.  Identifies entry that the restriction applies to.
    sens SMALLINT NOT NULL,
      -- FK to sens.sens.  Identifies the sense within the entry that
      -- the restriction applies to.
   rdng SMALLINT NOT NULL,
      -- FK to rdng.rdng.  Identifies the reading within the entry that
      -- the restriction applies to.
    PRIMARY KEY (entr,sens,rdng));

CREATE TABLE stagk (
      -- Lists invalid pairs of sense and kanji elements.
      -- Note that this is the inverse of the representation in JMdict
      -- XML file where <stagk> elements list the _valid_ pairs.
    entr INT NOT NULL,
      -- FK to entr.id.  Identifies entry that the restriction applies to.
    sens SMALLINT NOT NULL,
      -- FK to sens.sens.  Identifies the sense within the entry that
      -- the restriction applies to.
    kanj SMALLINT NOT NULL,
      -- FK to kanj.kanj.  Identifies the kanji within the entry that
      -- the restriction applies to.
    PRIMARY KEY (entr,sens,kanj));



-- All keyword tables names' start with "kw" and have the
-- same structure.  The primary key is a small integer
-- and that is what is used in other table to refer to a
-- keyword.  he other two columns, 'kw' and 'descr' provide
-- a short text abbreviation, and a longer textual description
-- for use in displays to users.

CREATE TABLE kwdial (
    id SMALLINT PRIMARY KEY,
    kw VARCHAR(20) NOT NULL UNIQUE,
    descr VARCHAR(255));

CREATE TABLE kwfreq (
    id SMALLINT PRIMARY KEY,
    kw VARCHAR(20) NOT NULL UNIQUE,
    descr VARCHAR(255));
```

```
CREATE TABLE kwfld (
    id SMALLINT PRIMARY KEY,
    kw VARCHAR(20) NOT NULL UNIQUE,
    descr VARCHAR(255));

CREATE TABLE kwginf (
    id SMALLINT PRIMARY KEY,
    kw VARCHAR(20) NOT NULL UNIQUE,
    descr VARCHAR(255));

CREATE TABLE kwkinf (
    id SMALLINT PRIMARY KEY,
    kw VARCHAR(20) NOT NULL,
      -- 'kw' not declared UNIQUE because there are some
      -- rows where kw differs only in case, and this (at
      -- in some of the locales used during testing) violated
      -- the unique constraint.
    descr VARCHAR(255));

CREATE TABLE kwlang (
    id SMALLINT PRIMARY KEY,
    kw VARCHAR(20) NOT NULL UNIQUE,
    descr VARCHAR(255));

CREATE TABLE kwmisc (
    id SMALLINT PRIMARY KEY,
    kw VARCHAR(20) NOT NULL,
      -- 'kw' not declared UNIQUE because there are some
      -- rows where kw differs only in case, and this (at
      -- in some of the locales used during testing) violated
      -- the unique constraint.
    descr VARCHAR(255));

CREATE TABLE kwpos (
    id SMALLINT PRIMARY KEY,
    kw VARCHAR(20) NOT NULL UNIQUE,
    descr VARCHAR(255));

CREATE TABLE kwrinf (
    id SMALLINT PRIMARY KEY,
    kw VARCHAR(20) NOT NULL UNIQUE,
    descr VARCHAR(255));

CREATE TABLE kwsrc (                  -- Entry corpora
    id SMALLINT PRIMARY KEY,
    kw VARCHAR(20) NOT NULL UNIQUE,
    descr VARCHAR(255),
    dt DATE,
      -- Date associated with this corpus.
    notes VARCHAR(255),

    -- The following four attributes are used read by a trigger on
    -- on the kwsrc table and used to automatically create a Postgresql
    -- sequence when a new row (corpus) is inserted in kwsrc.  That
    -- sequence will be used to create corpus-specific entr.seq numbers
    -- when entries of this corpus are inserted in table entr.
    -- The values are only used when a row in inserted in kwsrc;
    -- changing them later will have no effect on the (already created)
    -- sequence.
    seq VARCHAR(20) NOT NULL,
      -- Name (sans leading "seq_") of the Postgresql sequence.
    sinc SMALINT,
      -- Increment value of the sequence.
    smin BIGINT,
      -- Starting value of the sequence.
    smax BIGINT,
      -- Maximum value of the sequence.
```

```
    srct SMALLINT NOT NULL);
      -- FK to kwsrct.id.  Identifies this corpus being of the
      -- given source type where all entries of the same source
      -- share certain common requirements for display and editing.

CREATE TABLE kwsrct (
    id SMALLINT PRIMARY KEY,
    kw VARCHAR(20) NOT NULL UNIQUE,
    descr VARCHAR(255));

CREATE TABLE kwstat (
    id SMALLINT PRIMARY KEY,
    kw VARCHAR(20) NOT NULL UNIQUE,
    descr VARCHAR(255));

CREATE TABLE kwxref (
    id SMALLINT PRIMARY KEY,
    kw VARCHAR(20) NOT NULL UNIQUE,
    descr VARCHAR(255));
```

[1] http://www.csse.monash.edu.au/~jwb/edict_doc.html

[2] http://www.edrdg.org/~smg/

[3] The JMdict file is available in two forms: an English-only gloss version (ftp://ftp.cc.monash.edu.au/pub/nihongo/JMdict_e.gz), and a multi-lingual gloss version (ftp://ftp.cc.monash.edu.au/pub/nihongo/JMdict.gz) that is a superset of the English-only version. The JMdict database schema and tools support both versions. However, the non-English glosses in the multi-lingual version were merged in from other non-JMdict project files and the integration leaves something to be desired so most of the JMdictDB development work uses the English-only version.

[4] http://www.edrdg.org/~smg/. The JMdictDB source code licensed under the GPL and is available for download at this URL.

[5] http://www.postgresql.org/

[6] The schema.png file is distributed with the JMdictDB source code, or at http://www.edrdg.org/~smg/jmdict/schema.png