Grimm/Taxi HOWTO

Bryan Jurish
Berlin-Brandenburg Academy of Sciences · Jägerstraße 22/23 · 10117 Berlin · Germany
moocow@bbaw.de

May 7, 2009

Contents

1 Introduction **2**

2 Sources **4**
2.1 Directory Layout 5

3 The Build Subsystem **5**
3.1 From SGML to Raw XML 5
3.2 From Raw XML to Taxi XML 9
3.3 Additional Build Targets 13

4 The Indexing Subsystem **13**
4.1 Defining a New Index 14
4.2 Loading Corpus Data 18
4.3 Analyzing Corpus Data 19
4.4 Importing and Exporting Indices 21

5 The Runtime Subsystem **22**
5.1 Taxi Query Language 22
5.2 Direct Index Access 23
5.3 Server-Based Index Access 25

List of Figures

1 Grimm/Taxi System Architecture 3

2 Grimm/Taxi Server Architecture 25

List of Tables

1 Grimm/Taxi CVS directory layout 6

2 Common variables used by the build subsystem 7

3 Static DTD variables used by the build subsystem 7

4 Selected SGML→XML targets in `grimm/xml` 9

5 Selected Taxi-XML targets in `grimm/xml` 10

6 Phonetic analysis variables used by the build subsystem 12

7 Morphological analysis variables used by the build subsystem 12

8 Selected additional XML targets in `grimm/xml` 13

1 Introduction

This document attempts to provide a general overview of the Grimm/Taxi quotation evidence indexing system prototype as implemented and running at the Berlin-Brandenburg Academy of Sciences. The chief purpose of the Grimm/Taxi system is to a provide flexible and efficient application program interface (API) for fine-grained queries over a corpus of historical German text extracted from quotation evidence ocurring in the *Deutsches Wörterburch* (DWB) by Jacob and Wilhelm Grimm.[1]

The Grimm/Taxi quotation evidence indexing system is (as its name suggests) a system for the construction, analysis, and efficient retrieval from a corpus of quotation evidence drawn from a dictionary source. A dataflow diagram for the major components of the build and runtime system is given in Figure 1.

As shown in Figure 1, the Grimm/Taxi system is conceptually divided into three separate subsystems. The *build subsystem* is responsible for converting raw SGML DWB source documents to Taxi-XML format, the *indexing subsystem* is responsible for the administration of a MySQL relational database containing all relevant corpus information, and the *runtime subsystem* is responsible for parsing of user queries, and for the retrieval and

[1] Electronic sources for the *Deutsches Wörterbuch* in raw SGML format were kindly provided by the University of Trier.

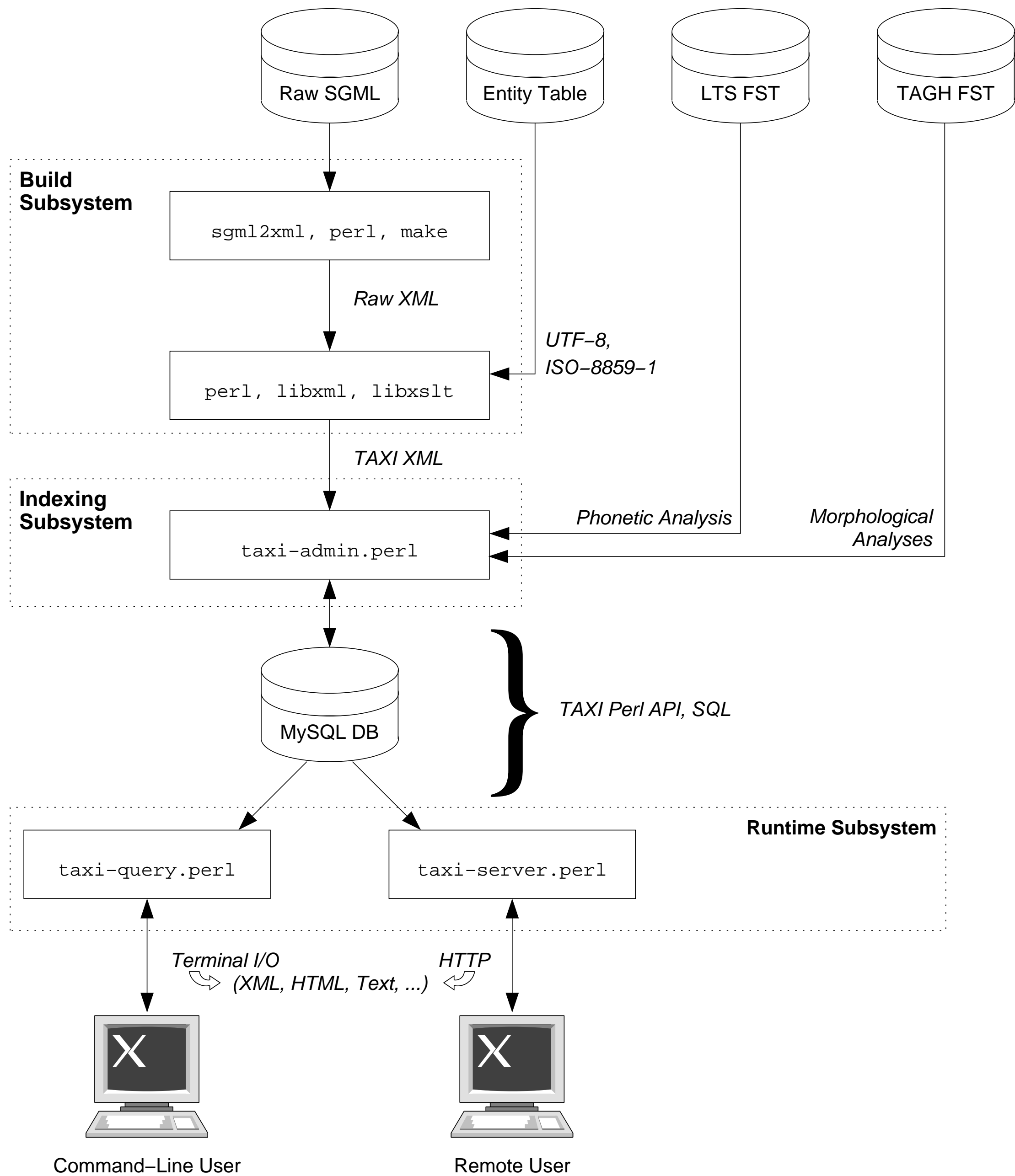

Figure 1: Grimm/Taxi System Architecture

formatting of corpus data in response to user queries. Both the indexing and runtime subsystems consist of an abstract layer (the Taxi perl API) which supports indexing and search of arbitrarily structured XML documents on the one hand, and on a DWB-specific set of subclasses (Taxi::Grimm) and configuration files on the other.

Each of the remaining sections of this document will focus on one specific subsystem. Section 2 describes how to acquire current versions of the system components, and gives a brief overview of the conventions used in and by the Taxi/Grimm CVS repository. Section 3 deals with corpus preparation with the build subsystem, section 4 contains a brief overview of document loading and database administration with the indexing subsystem, and section 5 deals with the runtime subsystem.

2 Sources

All of the sources for the Taxi/Grimm system are available on the servers of the DWDS project. Most of the sources are available via CVS:

```
export CVS_RSH=ssh
export CVSROOT=:ext:USER@holodoc.woerterbuch-portal.de:/home/cvs
cvs co PROJECT
```

... where *USER* is replaced by a valid username on holodoc, and *PROJECT* is the name of a Taxi/Grimm related CVS project. Currently, these are:

Project Name	**Description**
`grimm`	Build subsystem utilities, including `Lingua::LTS`
`Taxi-Mysql`	Indexing and runtime perl sources
`DDC-Perl`	Drop-in DDC wrapper daemon (alternate runtime engine)

The DWB SGML sources are not included in any of the above projects. They can be found on the server `kirk.bbaw.de` in the directory:

```
/home/scratch/Austausch/fuer_moocow/grimm
```

The most recent version of the DWB SGML sources at the time of this writing was:

```
grimm-sgml-2006-11-22.tar.gz
```

2.1 Directory Layout

The most deeply nested project directory is the `grimm/` directory. In each major subdirectory of the `grimm/`, there is a file `README.moo.txt` which gives a brief overview of the directory's purpose, dependencies, contents, and usage. A minimal overview of the directory structure under `grimm/` is given in Table 2.1.

3 The Build Subsystem

The Grimm/Taxi build subsystem is responsible for converting raw SGML DWB source documents to Taxi-XML format. At the time of this writing, build subsystem is still under active development, and build conventions are subject to change as heuristics for extraction of prose quotation evidence are integrated into the procedure. Check the contents of the `README.moo.txt` file in the build directory for the most current information. This section describes the Grimm/Taxi build subsystem as implemented and running on 1st January, 2007. Section 3.1 deals with the conversion of the SGML DWB sources to raw XML, while section 3.2 is concerned with the conversion from raw XML to Grimm/Taxi-XML.

The Grimm/Taxi build subsystem is driven by a set of rules located in `grimm/xml/Makefile`, which require GNU `make` for correct interpretation. All command-line examples in the remainder of this section assume that the working directory is `grimm/xml/`. For further information, see the comments in `grimm/xml/Makefile` and the user variables in `grimm/config/common.mak`.

3.1 From SGML to Raw XML

The primary goal of the conversion of the DWB sources from SGML to raw XML is the imposition of an easily parseable format with minimal information loss. At the time of this writing, the following information in the original SGML sources is lost and/or modfied during conversion to raw XML:

- The DTD in the SGML <!DOCTYPE ...> declaration is set to `MHDWB_BBAW.dtd` in output XML. See sections 3.1.2 and 3.1.3 for details on XML DTDs.
- Typographical entities (*e.g.* `&kurisv;`, `&recte;`, `&super;`, `∩︀`, *etc.*) are converted to <hi>...</hi> elements, whereby the `&recte;` entity is translated as a close-tag "</hi>"; thus the SGML code:

  ```
  &kursiv;Seite&recte; 7&Super;b&super;
  ```

  is converted to the XML code:

  ```
  <hi rend="kursiv">Seite</hi> 7<hi rend="super">b</hi>
  ```

Directory	**Descrition**
`grimm`	Top-level directory for various tasks involving Grimm/DWB sources and (potential) analyses & manipulations thereof.
`grimm/bin`	Many useful (and some obsolete) scripts for hacking, munging, frobbing, tweaking, and/or twiddling Grimm/DWB sources.
`grimm/config`	Global configuration directory for Grimm/DWB stuff.
`grimm/doc`	High-level documentation sources for Grimm/DWB stuff.
`grimm/dtds`	Static and dynamically auto-generated XML DTDs for Grimm/DWB XML.
`grimm/dtds/entities`	Grimm entity resolution (UTF-8) and Latin-1 approximation tools live here.
`grimm/dtds/unicode`	UTF-8 related sandbox. Can probably be eliminated entirely. See ../entities for related stuff actually in use.
`grimm/lts`	Top-level directory for building LTS (Letter-To-Sound) analysis transducers.
`grimm/lts/Lingua-LTS`	Perl module sources for the Lingua::LTS module and support scripts.
`grimm/lts/bin`	Useful scripts for compiling LTS transducers.
`grimm/lts/grimm`	Make directory for building Grimm/DWB LTS (Letter-To-Sound) analysis transducer.
`grimm/lts/ims-german`	Make directory for building IMS German Festival LTS (Letter-To-Sound) analysis transducer.
`grimm/morph`	Directory for finite-state morphology transducer files. Empty in CVS:
`grimm/notes`	This directory contains various notes. It probably should not live in CVS, but it currently does.
`grimm/sgml`	Directory containing Grimm/DWB SGML sources.
`grimm/test`	This directory contains some test files and directories. It definitely should not live in CVS, but it currently does.
`grimm/tt`	Make directory for generating moot ”rare” and ”medium rare” format files from Grimm raw XML sources. Rules and code in this directory are quite old and are no longer used. It can safely be ignored (probably).
`grimm/xml`	This is a make directory for converting raw Grimm SGML sources into Taxi-XML, via Raw XML. Older rules also support conversion to DDC ”Free Index” XML.

Table 1: Grimm/Taxi CVS directory layout

- Output XML documents are well-formed: omitted close-tags for any open elements are inserted, at the latest on encountering a close-tag for an `entry` element "`</entry>`".

If all `make` variables are set correctly (see below), conversion to raw XML can be initiated by calling:

```
bash$ make xml
```

in the `grimm/xml` directory. Details on some of the available `make` variables and rules are presented in the following subsections.

3.1.1 SGML Sources

The rules in `grimm/xml/Makefile` expect all DWB SGML source files to be located in the in the directory `grimm/sgml/selected`. In particular, `grimm/xml/Makefile` uses the variables listed in Table 2, among others. Any or all of the build variables may be altered on the command-line or by setting an environment variable of the same name. See the `make(1)` manpage for details.

Variable	Default Value	Description
GRIMM_ROOT	$(HOME)/work/bbaw/grimm	build root directory
GRIMM_SGML_ROOT	$(GRIMM_ROOT)/sgml	DWB SGML root directory
GRIMM_SGML_DATA	$(GRIMM_SGML_ROOT)/selected	DWB SGML data directory
SGML_SOURCES	$(GRIMM_SGML_DATA)/*.sgm	selected SGML sources
XML_TARGETS	$(SGML_SOURCES:.sgm=.xml)	selected XML targets

Table 2: Common variables used by the build subsystem

3.1.2 Static XML DTDs

The build subsystem uses the directory variables listed in Table 3 to search for required static SGML and XML DTDs.

Variable	Default Value	Description
GRIMM_SGML_DTDS	$(GRIMM_SGML_ROOT)/dtds	DWB SGML DTDs
GRIMM_EXTRA_DTDS	$(GRIMM_ROOT)/dtds	Static XML DTDs
TEI_ROOT	$(HOME)/local/share/tei	TEI (p4) root
TEI_DTDS	$(TEI_ROOT)/xml/teip4/schema/dtd	TEI DTDs

Table 3: Static DTD variables used by the build subsystem

The static DTD referenced by raw XML documents is `MHDWB_BBAW.dtd`, which is located in the `$(GRIMM_EXTRA_DTDS)` directory. Currently, `MHDWB_BBAW.dtd` contains no structural consraints, and thus does not allow validation. It does however include a reference to a dynamically generated DTD for UTF-8 entity resolution. See the following section for details.

3.1.3 The Entity Table and Dynamic XML DTDs

UTF-8 approximations of most character entities in the DWB SGML sources are provided by a dynamically generated XML entity definition DTD file `MHDWB_BBAW.ent`, which is included by `MHDWB_BBAW.dtd`. The dynamic entity definition is specified by the `make` variable `GRIMM_MOOHACK_DTD`, and is automatically generated by the build subsystem from an "entity table" in native Perl syntax whose location is in turn specified by the `make` variable `GRIMM_MOOHACK_ETAB`. Complete documentation of the syntax of the entity table file and of the heuristics used to generate many of the UTF-8 approximations is beyond the scope of this document; for details, see the source code, which is available in the file `grimm/dtds/entities/EntityTable.pm`.

3.1.4 `make` Rules and Targets

The following is a partial list of `make` targets available in the `grimm/xml/` build directory. Targets listed with wildcards (one or more "`*`" characters) are applied to individual source files or intermediate targets, thus the target listed as "`*.xml`" can be called as `ga01.xml`, `ga02.xml`, ..., `gz25.xml`; assuming that the sources `ga01.sgm`, `ga02.sgm`, ..., `gz25.sgm` exist in the appropriate locations.

Target	Description
`xml`	batch target for conversion to raw XML
`all`	alias for `xml`
`grimm-catalog.xml`	auto-generated XML DTD catalog file: this only exists because the `libxml2` on SuSE 9.2 is hopelessly outdated
`../dtds/MHDWB_BBAW.ent`	auto-generated UTF-8 entity resolution file: generation `make` rules are here, guts are in `../dtds/entities`
`g*.xml`	top-level raw XML target (link to `g*.final.xml`)
`g*.errors`	top-level raw XML errors target (link to `g*.final.errors.xml`)
`*.dtd-hacked.sgml`	hacks SGML DTD name and hides entities in comment nodes
`*.sgml2xml.xml`	proto-XML generated by `sgml2xml` (from James Clark's `SP` package)

Target	**Description**
`*.sgml2xml.errors`	errors reported by `sgml2xml`
`*.doctype-hacked.xml`	hacks XML DTD name and restores entities hidden in comments
`*.xmllint.xml`	formatted raw XML with checked by xmllint (entities still hidden)
`*.xmllint.errors`	errors reported by `xmllint` (no entity-related errors)
`*.tei.xml`	formatted raw XML with element names replaced by `teiform` where applicable
`*.ents.xml`	formatted raw XML with entities un-hidden and un-resolved
`*.raw.xml`	final formatted raw XML (re-checked by xmllint for entity-safety)
`*.raw.errors`	more errors reported by xmllint (including entity-related errors)
`*.final.xml`	placeholder target (link to `*.raw.xml`) for final raw XML format
`*.final.errors`	collected errors for SGML→XML translation of corresponding file
`*.fmt.xml`	XML pretty-printing via `xmllint`
`*.nodtd.xml`	removes `DOCTYPE` declaration from `*.xml`
`clean`	removes most generated files
`realclean`	removes more generated files

Table 4: Selected SGML→XML targets in `grimm/xml`

3.2 From Raw XML to Taxi XML

Raw XML versions of the DWB sources can be extended with additional annotations in order to facilitate indexing with a `Taxi::Grimm` and/or `Taxi::Grimm2` XML index. For use as source documents for a Taxi index, the raw XML DWB sources must be tokenized, and ISO-8859-1 ("latin-1") approximations of each recognied token must be computed. Section 3.2.2 presents an overview of the tokenization procedures used by the build subsystem. Tokenized Taxi-XML documents may be optionally "pruned" of unneeded text nodes, as discussed in section 3.2.3. Additionally, each token in a Taxi-XML document may be optionally annotated with a phonetic form and/or with one or more morphological analyses, although by default such annotations will be re-computed during the Taxi indexing phase (see section 4). Methods for adding optional annotations during the build phase are discussed in sections 3.2.5 and 3.2.6. The `make` rules for generation of Taxi-XML are summarized in Table 5.

Target	Description
taxi	Batch rule for generation of all g*.taxi.xml files
g*.taxi.xml	Placeholder target (link to g*.taxi.tok.pruned.xl.xml)
*.noent.xml	Pre-tokenization: character entities replaced by their DTD expansions
*.taxi.tok.xml	Taxi-tokenized XML
.taxi..pruned.xml	Taxi-XML pruned of non-header text nodes
.taxi..xl.xml	Taxi-XML with Latin-1 token text approximations
.taxi..pho.xml	(obsolete) Taxi-XML with phonetic form annoations
.taxi..morph.xml	(obsolete) Taxi-XML with morphological annoations

Table 5: Selected Taxi-XML targets in `grimm/xml`

3.2.1 Entity Resolution

The first required step in the generation of Taxi-XML is the replacement of all character entities in the raw XML source document by their UTF-8 approximations using the `*.noent.xml` target, thus the raw XML:

```
kein g&oaboveu;tig wort
```

might be converted to:

```
kein gůtig wort
```

3.2.2 Tokenization

Taxi index source documents must be tokenized before indexing. The build subsystem supports tokenization of entity-free XML documents via the `*.taxi.tok.xml` target, which calls the script `$(GRIMM_BIN)/grimm-taxi-tokenize.perl` to perform the tokenization into `<w>` elements with UTF-8 text attributes "u"; thus the raw XML:

```
kein gůtig wort
```

might be tokenized as:

```
<w u="kein"/> <w u="gůtig"/> <w u="wort"/>
```

3.2.3 (Optional) Pruning

In the interest of keeping Taxi source files small, tokenized Taxi-XML documents may be "pruned" of unneccessary text nodes via the `*.taxi.tok.pruned.xml` rule. Pruning is applied by default.

3.2.4 Latin-1 Approximation

In order for analysis with the generated LTS transducer and/or the TAGH morphological analysis transducer, each token element must be associated with a Latin-1 (ISO-8859-1) approximation in its "l" (ell) attribute. Translation of UTF-8 approximations to Latin-1 approximations is performed by a finite state transducer which is generated from the entity table (see section 3.1.3), which is automatically generated and called by the `*.taxi.tok.pruned.xl.xml` rule. The example tokens:

```
<w u="kein"/> <w u="gůtig"/> <w u="wort"/>
```

might appear after Latin-1 approximation as:

```
<w u="kein" l="kein"/>
<w u="gůtig" l="gutig"/>
<w u="wort" l="wort"/>
```

3.2.5 (Optional) Phonetic Analysis

Each token in a Taxi-XML source document may be optionally annotated with a phonetic form in its "`p`" attribute. The build subsystem provides for phonetic analysis with a finite-state letter-to-sound (LTS) transducer by means of the `*.taxi.tok.pruned.xl.pho.xml` target. By default, phonetic analyses computed at build-time will be re-computed and overwritten during the Taxi indexing phase (see section 4.3.1). A summary of the `make` variables controlling build-time phonetic analysis is given in Table 6.

The example tokens:

```
<w u="kein" l="kein"/>
<w u="gůtig" l="gutig"/>
<w u="wort" l="wort"/>
```

might appear after phonetic analysis as:

```
<w u="kein" l="kein" p="k[aI]n"/>
<w u="gůtig" l="gutig" p="gutIC"/>
<w u="wort" l="wort" p="vO6t"/>
```

Variable	Default Value	Description
GRIMM_LTS_ROOT	$(GRIMM_ROOT)/lts	base dir
GRIMM_LTS_BIN	$(GRIMM_LTS_ROOT)/bin	script dir
GRIMM_LTS_LANG	$(GRIMM_LTS_ROOT)/grimm	"language" dir
GRIMM_LTS_DICT	$(GRIMM_LTS_LANG)/lts-grimm.dict	dictionary
GRIMM_LTS_FST	$(GRIMM_LTS_LANG)/lts-grimm.gfst	transducer
GRIMM_LTS_LAB	$(GRIMM_LTS_LANG)/lts-grimm-latin1.lab	labels
GRIMM_LTS_LABENC	ISO-8859-1	label encoding

Table 6: Phonetic analysis variables used by the build subsystem

3.2.6 (Optional) Morphological Analysis

Similar to the case for LTS analysis, the Grimm/Taxi system can use a variant of the TAGH finite-state morphology for morphological analysis of input tokens. Each token in a Taxi-XML source document may be optionally annotated with zero or more morphological analyses as child "`ma`" elements. The build subsystem provides for morphological analysis with a TAGH-style finite-state transducer by means of the `*.taxi.tok.pruned.xl.pho.morph.xml` target. By default, morphological analyses computed at build-time will be re-computed and overwritten during the Taxi indexing phase (see section 4.3.2). A summary of the `make` variables controlling build-time morphological analysis is given in Table 7.

Variable	Default Value	Description
GRIMM_MORPH_EOW	%	EOW marker
GRIMM_MORPH_FST	$(GRIMM_ROOT)/morph/mootm-stts-*.gfst	transducer
GRIMM_MORPH_LAB	$(GRIMM_ROOT)/morph/mootm-stts.lab	labels

Table 7: Morphological analysis variables used by the build subsystem

The exmple tokens:

```
<w u="kein" l="kein" p="k[aI]n"/>
<w u="gůtig" l="gutig" p="gutIC"/>
<w u="wort" l="wort" p="vO6t"/>
```

might thus appear after morphological analysis as:

```
<w u="kein" l="kein" p="k[aI]n">
  <ma lemma="keine" pos="PIS" feat="[nom][sg][neut]"/>
  <ma lemma="keine" pos="PIS" feat="[acc][sg][neut]"/>
  <ma lemma="keine" pos="PISNEG" feat="[nom][sg][neut]"/>
```

```
    <ma lemma="keine" pos="PISNEG" feat="[acc][sg][neut]"/>
    <!-- etc.  -->
  </w>
  <w u="gůtig" l="gutig" p="gutIC"/>
  <w u="wort" l="wort" p="vO6t">
    <ma lemma="Wort" pos="NN" feat="[neut][sg][nom_acc_dat]"/>
  </w>
```

3.3 Additional Build Targets

Several other output formats are supported by the `make` rules in `grimm/xml/Makefile`. In particular, support is included for DDC XML with optional phonetic form annotations. The `make` rules for generation of alternative targets are summarized in Table 8.

Target	Description
`ddc`	Batch rule for basic DDC XML in `*.ddc.xml`
`ddc.pho`	Batch rule for extnded DDC XML in `*.ddc.pho.xml`
`ddc-split`	(experimental) batch rule for DDC XML document splitting into directory `out/`
`g*.ddc.xml`	DDC-compatible XML format (basic)
`g*.ddc.pho.xml`	DDC-compatible XML format with phonetic field

Table 8: Selected additional XML targets in `grimm/xml`

The `ddc` targets are designed to produce input documents acceptable to the DDC corpus indexing system by Alexey Sokirko. A DDC wrapper daemon with support for “sounds-like” queries based on an FST-derived phonetic field “pho” such as produced by the `ddc.pho` target can be found in the CVS module `DDC-perl`. Note that unlike the Taxi system, the DDC wrapper daemon requires runtime access to the phonetic analysis transducer. See the `ddc-lts-wrapper.perl` manpage for details.

4 The Indexing Subsystem

The Grimm/Taxi indexing subsystem is responsible for creation and administration of a MySQL database index of a tokenized corpus. The `Taxi::MySQL` Perl distribution provides an abstract interface to MySQL database corpus indices, while the `Taxi::MySQL::Grimm` and `Taxi::MySQL::Grimm2` modules provide concrete implementations for corpora in the Grimm/Taxi format. The sources for the indexing and runtime subsystems can be found in the `Taxi-Mysql` CVS module. Most administration tasks described in this section can

be accomplished by calls to the utility script `taxi-admin.perl`, which is included in the `Taxi::Mysql` distribution.

4.1 Defining a New Index

Each Taxi index is defined by a configuration file in native perl syntax. The configuration file must assign an index object as a value to the (local) variable `$obj`, usually by calling:

```
my $obj = Taxi::Mysql->new(%keyword_arguments);
```

Common arguments to `Taxi::Mysql::new` include:

- `prefix => $prefix`
  Causes `$prefix` to be prefixed to all table names maintained by this Taxi index. Default=`'_taxi_'`.

- `handleArgs => \%handleArgs`
  Arguments for the underlying database handle, especially `dsn => $dbi_dsn_string`.

  Particularly useful is the ability to specify MySQL client configuration files and client tags in the DBI DSN string. For instance, if you specify:

  ```
  handleArgs => {
    dsn=>("DBI:mysql:"
          ."mysql_read_default_file=~/.my.cnf;"
          ."mysql_read_default_group=grimmTaxi;")
  }
  ```

  and your `/.my.cnf` contains a `grimmTaxi` section such as:

  ```
  [grimmTaxi]
  host     = localhost
  database = myschema
  user     = myuser
  password = mypassword
  ```

  ... then the underlying database is assumed to be on the local machone `localhost`, in the schema `myschema`, and will be accessed as the user `myuser` with password `mypassword`.

  See `Taxi::Mysql::Handle(3pm)` for details.

- `dbEncoding => $encoding`
  Assumed encoding of underlying database strings (perl side). `$encoding` should be some encoding known to the perl `Encode` module. The actual encoding conventions must match those of the MySQL `$charset` (see below). Default=`'UTF-8'`

- `dbCharset => $charset`
  Encoding of underlying database strings (mysql side). Default=`'utf8'`

  Beware: Bad things will happen if the value of `$charset` does not match the default character set of the underlying MySQL schema. You can change the MySQL schema character set "by hand" by executing the SQL query (e.g. in the `mysql` client program):

  `alter database` *SCHEMA* `charset` *CHARSET*;

  If you get a lot of funny looking characters where you expect diacritics, this may be what's biting you.

- `tables => %tableName2tableSpec`
  Specifies source document and index structure. This is where all of the interesting configuration takes place. See the `Taxi::Mysql::Table` manpage for details. For examples, see the default table specifications in the `table` keyword arguments in the `new()` methods for the `Taxi::Mysql::Grimm` and/or `Taxi::Mysql::Grimm2` classes in the `Taxi-Mysql` CVS module.

- `queryArgs => %queryArguments`
  Specify some default arguments to pass when creating new Query objects. Useul keywords in `%queryArguments` include:

```
pagesize      => $n,       ##-- defalt number of hits per page
default_table => $tabName, ##-- default 'token' table name
default_hit   => $refName, ##-- default hit container name
```

- `hitTables => hitTables`
  List of tables to join into the dataset returned for each hit. Default joins on all tables defined by the index.

- `extraHitColumns => [ [$tab,$col], ... ]`
  Specify additional columns to include in the dataset returned for each hit. Default is all 'attr'-type columns in any hit table.

- `fmtClass => $defaultFormatClass`
  Specify default class to use for formatting hit results. `$defaultFormatClass` should be a descendant of (or at least conform to the API specified by) the Perl class `Taxi::Mysql::Format::Base`.

- `fmtArgs => %formatArgs`
  Specify additional options to pass to (any) hit formatter's `new()` method.

  Formatter arguments can be used for instance to set output encoding, change default XML element names, apply XSLT fragments, select bibliographic data for display, etc.

See `Taxi::Mysql::Format::Base(3pm)` and subclasses for details on known arguments.

For details on index options and subclasses, see the `Taxi::Mysql`, `Taxi::Mysql::Grimm`, and/or `Taxi::Mysql::Grimm2` manpages, and/or the configuration files `zzz-grimm.PL` and `zzz-grimm2.PL` in the `Taxi-Mysql` CVS module.

The remainder of this section will assume an index configuration file `index.PL`. Assuming such a file exists in the current directory, the underlying MySQL database index can be created by:

```
bash$ taxi-admin.perl -i index.PL create
```

If an underlying MySQL database is no longer needed, it can be removed with:

```
bash$ taxi-admin.perl -i index.PL drop
```

4.1.1 Taxi::Mysql::Grimm2 Extensions

The `Taxi::Mysql::Grimm2` module accepts the following additional keyword arguments to its `new()` method:

- `lemmaEditCostMatch => $cost`
  Cost of a literal character match for the edit-distance lemmainstantiation heuristic.

  Default=0.

- `lemmaEditCostInsert => $cost`
  Cost of a single character insertion for the edit-distance lemmainstantiation heuristic.

  Default=1

- `lemmaEditCostSubst => $cost`
  Cost of a single character substitution for the edit-distance lemmainstantiation heuristic.

  Default=1.2

- `lemmaEditMaxDistSql => $sqlFragment`
  MySQL fragment to compute maximum acceptable edit-distance cost given a lemma in the MySQL variable `lemma` and an orthographic word type in the MySQL variable `text` for the edit-distance lemma instantiation heuristic. May be `undef` indicating no maximum.

  Default=`'LEAST(LENGTH(lemma),LENGTH(text))-1'`

- `lemmaEditRestrict => $sqlFragment`
  MySQL fragment to compute restrictions on valid lemma instantiations. May use the following variables, for a lemma type ℓ and an instance-type i:

Variable	Formula	Range	Description
`li.sim`	$\mathrm{sim}(\ell, i)$	$[0, 1]$	unit-normalized edit-distance similarity
`li.freq`	$\mathrm{f}(\ell, i)$	$\mathbb{N}$	raw joint frequency
`li.mi_bits`	$\mathrm{I}(\ell, i)$	$\mathbb{R}$	pointwise mutual information (bits)
`li.mi_by_l`	$\mathrm{I}(i\|\ell) = \frac{\mathrm{I}(\ell,i)-\min \mathrm{I}(\ell,*)}{\max \mathrm{I}(\ell,*)-\min \mathrm{I}(\ell,*)}$	$[0, 1]$	MI unit-normalized by lemma-type
`li.mi_by_i`	$\mathrm{I}(\ell\|i) = \frac{\mathrm{I}(\ell,i)-\min \mathrm{I}(*,i)}{\max \mathrm{I}(*,i)-\min \mathrm{I}(*,i)}$	$[0, 1]$	MI unit-normalized by instance-type
`li.score_bits`	$\mathrm{sim}(\ell, i) \times \mathrm{I}(\ell, i)$	$\mathbb{R}$	raw score (pseudo-bits)
`li.score_by_l`	$\mathrm{sim}(\ell, i) \times \mathrm{I}(i\|\ell)$	$[0, 1]$	score unit-normalized by lemma-type
`li.score_by_i`	$\mathrm{sim}(\ell, i) \times \mathrm{I}(\ell\|i)$	$[0, 1]$	score unit-normalized by instance-type
`li.score_avg`	$\frac{\mathrm{sim}(\ell,i)\times(\mathrm{I}(i\|\ell)+\mathrm{I}(\ell\|i))}{2}$	$[0, 1]$	symmetric average unit-normalized score

  See `Taxi-Mysql/Grimm2.pm` for current default value.

- `lemmaEditOrderBy => $sqlFragment`
  MySQL `ORDER BY` fragment which sorts (lemma, instance) pairs (ℓ, i) in descending order of the estimated likelihood that the instance component i is in fact an instance of the lemma component ℓ.

  Default=`'li.score_avg DESC'`.

- `ltsFstFiles => %ltsFstFiles`
  Hash-reference specifying locations of the letter-to-sound tranducer files. Use of an LTS FST requires `libgfsm`, the Perl `Gfsm` module, as well as the Perl `Lingua::LTS` module (included in the `grimm` CVS project under `grimm/lts/Lingua-LTS`).

  `%ltsFstFiles` accepts the following keys:

  - `fst => $fstFile`
    Filename of the LTS transducer, in binary GFSM format.

    Default=`'lts-grimm.gfst'`
  - `lab => $labFile`
    Filename of the LTS transducer alphabet.

    Default=`'lts-grimm.lab'`

  - `dict => $dictFile`
    Filename of the LTS exception dictionary. May be `undef` to use FST results exclusively.
    Default=`'lts-grimm.dict'`

- `ltsFstArgs => %ltsFstArgs`
  Additional arguments to `Lingua::LTS::FST::new()` for the LTS transducer. The defaults should be sensible.

- `morphFstFiles => %morphFstFiles`
  Hash-reference specifying locations of the morphological analysis tranducer files. Use of a morphology FST requires `libgfsm`, the Perl `Gfsm` module, as well as the Perl `Lingua::LTS` module (included in the `grimm` CVS project under `grimm/lts/Lingua-LTS`).

  `%morphFstFiles` accepts the following keys:

  - `fst => $fstFile`
    Filename of the morphology transducer, in binary GFSM format.
    Default=`'morph-grimm.gfst'`
  - `lab => $labFile`
    Filename of the morphlogy transducer alphabet.
    Default=`'morph-grimm.lab'`
  - `dict => $dictFile`
    Filename of the morphological analysis exception dictionary. May be `undef` to use FST results exclusively.
    Default=`'morph-grimm.dict'`

4.2 Loading Corpus Data

Corpus data may be loaded into an existing index by means of the `load` command to `taxi-admin.perl`. Assuming a valid index file `index.PL` and corpus files `c1.xml` and `c2.xml`, the command:

```
bash$ taxi-admin.perl -i index.PL load c1.xml c2.xml
```

will cause the corpus files `c1.xml` and `c2.xml` to be parsed and uploaded to the MySQL database specified by `index.PL`.

Warning: the `taxi-admin.perl` "`load`" command is **not** incremental: loading new documents into an existing index currently clears any and all data previously stored in that index!

Note that the `load` command to `taxi-admin.perl` calls `LOAD DATA INFILE` on the backend MySQL server, which requires that the executing MySQL user have the server-global `FILE` privilege. If you get permission errors, this might be why.

A number of options are available for the `load` command. See the `taxi-admin.perl` manpage, as well as the documentation of the `Taxi::Mysql::loadData()` method and of the dedicated loader class `Taxi::Mysql::Loader` for more information.

4.3 Analyzing Corpus Data

During the course of corpus parsing and upload, a number of temporary files are created. Taxi index subclasses may implement and/or override special analysis subroutines to alter these files prior to upload. Additionally, Taxi index subclasses may provide methods for post-upload analysis of an already populated MySQL database. In general, pre-upload analysis routines are implicitly invoked by the `load` command, but can also be called separately by:

```
bash$ taxi-admin.perl -i index.PL analyze
```

Post-upload routines should never be implicitly invoked by the `load` command, but must be invoked separately after uploading, by:

```
bash$ taxi-admin.perl -i index.PL dbanalyze
```

One common idiom is to immediately call post-upload analysis routines after loading, by:

```
bash$ taxi-admin.perl -i index.PL load corpus/*.xml , dbanalyze
```

The `Taxi::Mysql::Grimm` and `Taxi::Mysql::Grimm2` index subclasses offer both pre- and post-upload analysis routines, which are described in the following subsections.

4.3.1 Letter-to-Sound (LTS) Analysis

Letter-to-Sound analysis assigns a unique phonetic form to each orthographic word type in the corpus to be uploaded. LTS analysis is currently implemented as a pre-upload analysis hook in the `Taxi::Mysql::Grimm2::analyzeLTS()` method, which uses the LTS analysis transducer configuration specified by the `ltsFst*` keyword arguments to the constructor `Taxi::Mysql::Grimm2::new()`.

4.3.2 Morphological Analysis

Morphological analysis assigns zero or more morphological analyses to each orthographic word type in the corpus to be uploaded. Morphological analysis is currently implemented

as a pre-upload analysis hook in the `Taxi::Mysql::Grimm2::analyzeMorph()` method, which uses the morphological analysis transducer configuration specified by the `morphFst*` keyword arguments to `Taxi::Mysql::Grimm2::new()`.

4.3.3 Type-Wise Analysis

Various properties of orthographic word types in the `type` table are computed by the post-upload analysis hook `Taxi::Mysql::Grimm2::analyzeTypes()`. Such properties include:

- `freq`: observed frequency
- `isalpha` (boolean): true if the type's orthographic form contains only alphabetic characters (*i.e.* if the type looks like something the morphology "ought" to know about).
- `haspmorph` (boolean): true if any phonetic variant of the type has at least one morphological analysis.

4.3.4 Edit-Distance Lemma Instantiation Heuristics

Edit-distance lemma instantiation heuristics are an experimental feature introduced in `Taxi::Mysql::Grimm2` for estimating which (phonetic) types ocurring within an `<entry>` element for a given lemma might be considered instances of that lemma. The heuristic classifies that phonetic type i in each individual block a of quotation evidence as an instance of the lemma ℓ with which that block is associated for which the likelihood $L(\ell, i)$ of i instantiating ℓ is maximal among all types occurring in a, and for which $L(\ell, i)$ does not exceed user-specified bounds parameterized by the lemma and the argument type.

Lemma instantiation heuristics populate the `lemmatype` field of the `add` table with that instance type i occurring the `add` element a which best instantiates the lemma ℓ of the `entry` e containing the a, for $\mathcal{A}$ the set of all word types and $\mathcal{L}$ the set of all entry lemmata:[2]

$$\text{lemmatype}(a) \quad = \quad \arg \max_{w \in \mathcal{A} : a \lhd^* w} L(\text{pho}(w), \text{pho}(\text{lemma}(\text{entry}(a))))$$

where:

$$\text{L}(i, \ell) \quad = \quad \frac{\text{sim}(\ell, i) \times (\text{I}(\ell|i) + \text{I}(i|\ell))}{2}$$

[2]The formulae given above are incomplete in many respects. Note in particular that the user-specified upper bound `lemmaEditMaxDistSql` on "acceptable" edit distances, as well as the `lemmaEditRestrict` parameter on "acceptable" likelihoods are not included in these formulae, and that these parameters may cause an `add` element to have no "best" lemma-instantiating type at all. In such cases, lemmatype(a) is set to 0 (zero).

$$
\begin{aligned}
\mathrm{sim}(\ell,i) &= \frac{\texttt{lemmaEditMaxDist}(\ell,i) - \mathrm{EditDistance}(\ell,i)}{\texttt{lemmaEditMaxDist}(\ell,i)} \\
\texttt{lemmaEditMaxDist}(\ell,i) &= \min\{|\ell|,|i|\} - 1 \\
\mathrm{I}(i|\ell) &= \frac{\mathrm{I}(\ell,i) - \min \mathrm{I}(\ell,\mathcal{A})}{\max \mathrm{I}(\ell,\mathcal{A}) - \min \mathrm{I}(\ell,\mathcal{A})} \\
\mathrm{I}(\ell|i) &= \frac{\mathrm{I}(\ell,i) - \min \mathrm{I}(\mathcal{L},i)}{\max \mathrm{I}(\mathcal{L},i) - \min \mathrm{I}(\mathcal{L},i)} \\
\mathrm{I}(\ell,i) &= \log_2 \frac{\mathrm{P}(\ell,i)}{\mathrm{P}(\ell)\mathrm{P}(i)} \\
\mathrm{P}(\ell,i) &= \frac{|\{x \in Tok : x \cong i \;\&\; \mathrm{entry}(\ell) \lhd^* x\}|}{|Tok|} \\
\mathrm{P}(\ell) &= \sum_i \mathrm{P}(\ell,i) \\
\mathrm{P}(i) &= \sum_\ell \mathrm{P}(\ell,i)
\end{aligned}
$$

The `lemmavariant` table is populated from the `lemmatype` table by inserting a pair (ℓ, i) for every pair of *orthographic* types which are conflated by these heuristics.

Edit distance lemma instantiation heuristics are currently implemented as a post-upload analysis hook in the `Taxi::Mysql::Grimm2::dbAnalyzeTypes()` method, which makes use of the `lemmaEdit*` keyword arguments to `Taxi::Mysql::Grimm2::new()`. Use of edit-distance lemma instantiation heuristics requires the `PDL::EditDistance` module, available from http://www.ling.uni-potsdam.de/~moocow/projects/perl.

4.3.5 Morphological Coverage Summary

A number of informative statistics on breadth of the morphological coverage are gathered automatically by the post-upload analysis hook `Taxi::Mysql::Grimm::dbAnalyzeCoverage()`. Currently, this method just populates the database's `coverage` table. See the comments associated with the `coverage` table in the `Taxi-Mysql/Mysql/Grimm2.pm` sources.

4.4 Importing and Exporting Indices

For archiving purposes, as well as to facilitate use of identical Taxi indices on multiple hosts (*i.e.* for load distribution), any existing Taxi index may be exported to the local filesystem by means of the `export` command to `taxi-admin.perl`:

```
bash$ taxi-admin.perl -i index.PL -data-dir=./exported export
```

The `export` command creates one TAB-separated text file (`.txt`) for each table in the index's inventory, as well as an SQL script "`import.sql`" in the data export directory specified by the `-data-dir` option.

Exported index data may be (re-)loaded into the backend database server by the `import` command to `taxi-admin.perl` on the same index specification file used for export. Using the above example:

```
bash$ taxi-admin.perl -i index.PL -data-dir=./exported import
```

Alternately, the exported `import.sql` script can be used without the need for calling `taxi-admin.perl` to re-create and upload the exported data:

```
bash$ mysql -B SCHEMA < ./exported/import.sql
```

. . . of course, any subsequent Taxi queries require a valid `Taxi::Mysql` index, so hopefully you saved the `index.PL` with which your data was created when you exported the data too. . .

5 The Runtime Subsystem

5.1 Taxi Query Language

The Taxi query language is a high-level PROLOG-like language for expressing queries over a Taxi-indexed corpus, specifically designed for querying sequential text corpora whose basic elements are *tokens*. Taxi queries are expanded into SQL in two phases:

- **Hit acquisition phase**: The initial Taxi query is expanded into an SQL query which is assumed to return one row for each hit, together with a perl-parseable comma-separated row of independent variable (token) identifiers (primary keys) indicating which rows of the independent data rows (which tokens) triggered the match (used for highlighting).
- **Hit population phase**: For each hit returned by the initial query, an additional query may be constructed & sent to the backend database in order to collect more specific information on the hit in question.

Taxi queries are implemented as perl objects inheriting from the class `Taxi::Mysql::Query::Base`. Queries in the native syntax are parsed using the lexer/parser pair in the classes `Taxi::Mysql::YYLexer` and `Taxi::Mysql::YYParser`, respectively. Query interpretation constraints and other high-level routines are located in `Taxi::Mysql::Query::Parser`, which simultaneously serves to define a high-level API for support of alternative query languages. Each "hit" retrieved is implemented as a `Taxi::Mysql::Hit` object, which may be collected into a "page" of hits represented as a `Taxi::Mysql::HitList`.

For details on the implementation and use of the Taxi::Mysql query API, see the relevant manpages. For details on Taxi's native query syntax, see the file `doc/queryhelp.html` in the Taxi-Mysql distribution.[3].

5.2 Direct Index Access

A Taxi index may be queried directly with the command-line program `taxi-query.perl`, included in the Taxi distribution. The general form of an incantation is:

`taxi-query.perl -index=`*INDEX_FILE OPTIONS QUERY*

Assuming an index configuration file `index.PL` as described in Section 4.1, the incantaton is:

`taxi-query.perl -index=index.PL` *OPTIONS QUERY*

where *OPTIONS* are zero or more options, and *QUERY* is a query in the native Taxi query syntax (cf. Section 5.1).

Some of the more useful options to `taxi-query.perl` are:

-help
: Display a brief help message.

-index=*INDEX_FILE*
: Query the index stored in *INDEX_FILE*. This option is required.

-pagesize=*SIZE*
: Specify the maximum number of hits per page. May be set to 0 (zero) to retrieve all hits.

-pagenum=*NUM*
: Specify the first page number to display, counting from 0 (zero). Default=0.

-format-class=*CLASS_NAME_OR_SUFFIX*
: Format hits using the formatter class *CLASS_NAME_OR_SUFFIX*. Predefined formatter classes include the following (short names appear in parentheses on the far right):

- `Taxi::Mysql::Format::Text` (`Text`)
  Raw text formatter. Very informative, but very ugly.

[3] You may have to generate it first with `xsltproc`, but there's a Makefile which ought to take care of that for you in `doc/Makefile`.

- `Taxi::Mysql::Format::Text1` (`Text1`)
  One-hit-per-line text formatter. Still pretty ugly.
- `Taxi::Mysql::Format::TextBibl` (`TextBibl`)
  Human-readable text formatter (default).
- `Taxi::Mysql::Format::XML` (`XML`)
  Flat XML record list formatter.
- `Taxi::Mysql::Format::XMLBibl` (`XMLBibl`)
  XML formatter with bibliographic metadata headers for each hit.
- `Taxi::Mysql::Format::HTML` (`HTML`)
  Default HTML formatter.

-format-option=*OPTION=PERL_CODE*
Override default formatter options for a predefined formatter class. See formatter class manpages for details.

-format-file=*FORMAT_FILE*
Format with a user-defined formatter object read from *FORMAT_FILE*. Useful when the predefined formatter classes aren't enough. In conjunction with the abstract `Taxi::Mysql::Format::XSLT` and `Taxi::Mysql::Format::XSLTBibl` classes, may be used to apply an arbitrary user-defined XSL stylesheet to one of the `XML` or `XMLBibl` formats, especially dynamically generated stylesheets or stylesheets with bound perl functions. See the `Taxi::Mysql::Format::HTML` class for an example of these techniques.

-output=*OUTPUT_FILE*
Send formatted hit output to *OUTPUT_FILE*, rather than standard output.

-tracefile=*TRACE_FILE*
Record all SQL queries sent to backend MySQL server in *TRACE_FILE*. Useful for debugging.

5.2.1 Examples

- Query index from `index.PL`, retrieving the first 10 occurrences of the orthographic type “rabe”, formatting as human-readable text to standard output:

```
taxi-query.perl -i=index.PL -ps=10 -pn=0 -fc=TextBibl "rabe"
```

- Query `index.PL`, retrieving all occurrences of ”rabe” as 1-hit-per-line text:

```
taxi-query.perl -i=index.PL -ps=0 -fc=Text1 "rabe"
```

- Query `index.PL`, retrieving all occurrences of ”rabe” as bibliographically annotated XML to the file ”rabe.xml”:

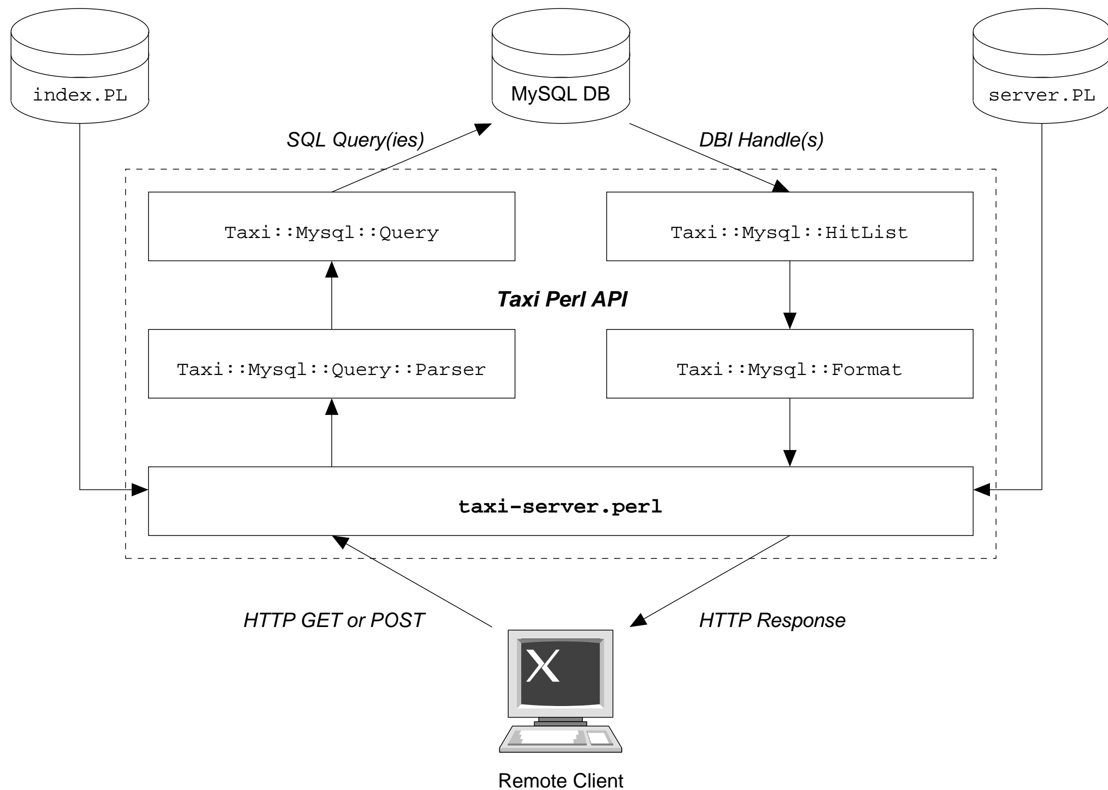

Figure 2: Grimm/Taxi Server Architecture

```
taxi-query.perl -i=index.PL -ps=0 -fc=XMLBibl -o=rabe.xml "rabe"
```

- ... retrieve only up to 42 hits, and format as a flat list of XML records:

```
taxi-query.perl -i=index.PL -ps=42 -fc=XML -o=rabe.xml "rabe"
```

- ... format as HTML and store in ”rabe.html”:

```
taxi-query.perl -i=index.PL -ps=42 -fc=HTML -o=rabe.html "rabe"
```

5.3 Server-Based Index Access

In addition to command-line based direct index access using `taxi-admin.perl` and the Taxi Perl API itself, Taxi indices may be queried using a standalone HTTP server `taxi-server.perl`, which itself wraps the perl class `Taxi::Mysql::Server`. Client queries are sent to the Taxi server using HTTP `GET` and/or `POST` methods. The Taxi server itself then parses the query and fetches a hit-list from the backend MySQL server. Hit lists are formatted and returned to the remote client via HTTP. See Figure 2 for a graphical portrayal.

5.3.1 Server Configuration

In addition to an index configuration file `index.PL` as described in Section 4.1, a Taxi server may also make use of a server configuration file. A *server configuration file* is a just a Perl source file which assigns a `Taxi::Mysql::Server` object as a value to the (local) variable `$obj`, usually by calling:

```
my $obj = Taxi::Mysql::Server->new(%keyword_arguments);
```

Common arguments to `Taxi::Mysql::Server::new` include:

- `index => $INDEX`
  Underlying Taxi::Mysql index object. You might just want to load `index.PL` here, e.g. with:

  ```
  my $ix = Taxi::Mysql->loadFile("index.PL")
             or die("load failed for index.PL: $!");
  my $obj = Taxi::Mysql::Server->new(index => $ix, %et_cetera);
  ```

- `uris => \%PATH_TO_CONFIG`
  Maps local URI paths to configurations which determine how the server will respond to matching URIs. See the `Taxi::Mysql::Server` documentation and sources for details.

  A server should define at least one URI of type `query` (expanding to an object of type `Taxi::Mysql::Server::URI::query`) in order to provide query-level access to an underlying `Taxi::Mysql` index.

- `daemonArgs => \%ARGS`
  Arguments to `HTTP::Daemon->new()`, which may include:

  - `LocalAddr => $LOCAL_IP_OR_HOSTNAME`
    IP or hostname of the local interface on which the server should listen for incoming queries.
  - `LocalPort => $LOCAL_PORT_OR_SERVICE`
    Port number or service name of the local port on which the server should listen for incoming queries.
  - `ReuseAddr => $BOOL`
    You probably want to set this to a true value. Really.

    See the `IO::Socket::INET(3pm)` manpage and/or the `socket.h(7)` manpage and/or the documentation of your system's C library for more details.

- `daemonMode => $mode`
  One of "fork" or "serial". Default is "serial".

- `allow => \@allow_ip_regexes`
  Always allow queries from these clients (default: empty).

  See "`deny`" for allow/deny semantics.

- `deny => \@deny_ip_regexes`
  Deny queries from these clients, unless they are explicitly allowed (default: empty).

  Queries from a client are allowed if and only if:

  $$\mathrm{ip}(client) \in \mathrm{Allowed}\ \mathbf{or}\ \mathrm{ip}(client) \notin \mathrm{Denied}$$

  ... that is, if the client's IP matches any regular expression in `@allow_ip_regexes`, then a query from the client is explicitly allowed. Otherwise, if the client's IP does matches any regular expression in `@deny_ip_regexes`, then a query from the client is explicitly disallowed. Otherwise, a query from the client is implicitly allowed.

- `fmtClass => $FORMAT_CLASS`
  Set default hit formatter class.

- `fmtArgs => \%FORMAT_ARGS`
  Set default hit formatter arguments.

5.3.2 Running the Server

The program `taxi-server.perl` is provided as a standalone Taxi index server. A typical server startup call looks like:

> `taxi-server.perl -config=`*SERVER_CONFIG OPTIONS*

This, if your server configuration file is located in `server.PL`, the call becomes:

> `taxi-server.perl -config=server.PL` *OPTIONS*

Some useful options to `taxi-server.perl` include:

-help
Display a brief help message.

-config=*SERVER_CONFIG_FILE*
Load server configuration from *SERVER_CONFIG_FILE*.

-index=*INDEX_CONFIG_FILE*
Load index configuration from *INDEX_CONFIG_FILE*. May be used to override a value set in *SERVER_CONFIG_FILE*, if any.

-fork, -nofork
Set or override default daemon mode (forking or serial).

-daemon-host=*HOST*
Override the local interface on which to listen (default: all interfaces).

-daemon-port=*PORT*
Set or override the local port on which to listen (default: 8080).

-daemon-option=*OPTION=VALUE*
Set or override other `HTTP::Daemon` options.

-logfile=*LOG_FILE*
Name of file to which server activity will be logged, in place of STDERR.

5.3.3 Querying the Server

To send a query to a running Taxi server, a client need only send a specially formatted HTTP request (using either the `GET` or the `POST` method) to the host and port on which the server is listening. The client should request a URI configured in the server as a `query` URI, and the request should include a `query` variable whose value is the Taxi native query string. Supported HTTP request variables include:

- **query=*QUERY***
  Client query in native Taxi syntax. *QUERY* may use XML-style literal character entities (decimal or hexidecimal escapes) anywhere in the query.
- **pagenum=*NUM***
  Request page number *NUM* of hits.
- **pagesize=*SIZE***
  Request pages of up to *SIZE* hits per page. *SIZE* may be set to zero to indicate no maximum.
- **formatClass=*CLASS***
  Specify formatter class. *CLASS* must be the name of a pre-loaded hit formatter class which respects the `Taxi::Mysql::Format::Base` API conventions.

Here is an example script to query a `Taxi::Mysql::Server` running on `localhost` port `8080` with a `query` URI at path '/qpath':

```
#!/usr/bin/perl -w

use LWP::UserAgent;
```

```
use HTTP::Request::Common;

$server = "http://localhost:8080/qpath";         # base server URL
$fmt    = 'TextBibl';                            # formatter class
$query  = "token.type.text='rabe'";              # Taxi query string

$url = ($server                                  # full query URL
        . '?query='       . uri_escape($query)
        . '&formatClass=' . uri_escape($fmt)
       );
$ua  = LWP::UserAgent->new();                    # user agent
$response = $ua->request(GET($url));             # ... The Answer

print $response->as_string, "\n";                # ... as a string
```

5.3.4 Grimm HTML GUI

The `Taxi::Mysql::Grimm` and `Taxi::Mysql::Grimm2` subclasses introduce a serverl additional URI subclasses, notably the `Taxi::Mysql::Server::URI::Grimm2::WordInfo` subclass for detailed information on single (orthographic) word types, as well as the `Taxi::Mysql::Server::URI::Grimm2::Status` subclass for dynamically generated summaries of database content and basic coverage statistics. See the source code for details.