Introduction

This documents the design of the DisOrder databases.

Requirements

Each track has two types of metadata associated with it, track data and prefs data. Track data can be reconstructed automatically (but slowly), prefs data is unrecoverable if lost. The current design separates these into `tracks.db` and `prefs.db`, and starts track data keys with "_" to distinguish them from prefs keys.

Aliases: some of the prefs start `trackname_display_` and create alternative names for tracks. Aliases should be visible in the virtual filesystem under the same root. Where an alias and the real track are in the same directory the alias name should win.

The database doesn't have to care about sort order within directories, the upper layers sort that out on the basis of the original track name and `trackname_sort_` prefs.

Searching: it should be possible to efficiently get from a word (or a set of words) to the set of tracks that contain that word in anything that might reasonably be thought to be a name for them. It is acceptable for the database to give a small superset of that set of tracks and filter it down but the excess should never be large enough to noticably affect performance or disk usage.

The set of stopwords may affect the meaning of the search database. It is acceptable to require a manual process to deal with changes to this list since they will be rare.

Tags: tags are nonempty UTF-8 strings that do not contain control characters, spaces, commas or semicolons, but are otherwise arbitrary. Each track's tags are stored space-separated in the `tags` pref. It should be possible to efficiently get from a tag to the tracks that have it, so that tags can be used (for instance) to restrict random play, and possible to efficiently enumerate all tags. (2006-05-03)

Recovery: Any database except the prefs database might get manually removed (but not while the server is running); if this happens it must be automatically reconstructed.

Updates

The databases are empty to start with.

At startup, at user request and periodically, a rescan of an individual collection is performed. (A collection is a separately configured chunk of filesystem with a single root which contains files. Collections never overlap.)

This rescan will produce a list of all the files in the collection along with the associated octet-string filenames (for passing to players; these octet-strings are basically the only non-UTF-8-compatible strings in DisOrder.)

Since this list may be huge the current implementation then does a second sweep over the known tracks and removes any not found in the filesystem. (It does length calculations as part of the same process.) Removing obsolete tracks at rescan time is a requirement but doing it this way in particular is not.

It is more important that the full set of tracks is quickly available than that their lengths are all computed. Length calculation should not delay notification of tracks.

Compatibility

The existing `prefs.db` is sacred, but other databases can be changed around as need be.

Old context-free `trackname_` prefs can be disregarded.

Rescans may be required to apply to all collections at once but this is dis-preferred.

Other Notes

It is acceptable to use a separate process if necessary.

Threads are strongly dispreferred.

Databases

tracks.db

Keys are tracks. Values are url-encoded strings of track data name=value pairs (all names starting with "_"). Known names are:

Name	Meaning
_path	path name in local encoding
_length	track length in seconds
_alias_for	real track that this is an alias for

prefs.db

Keys are tracks. Values are url-encoded strings of name=value prefs. If there are no prefs then there need not be a key for that track.

search.db

Keys are words. Values are track names. Duplicate keys are allowed.

tags.db

Keys are tags. Values are track names. Duplicate keys are allowed.

Consistency

For each existing track the following should be true:

- `tracks.db` has an entry with `_path` and (eventually) `_length` set
- `search.db` has an entry for each word in the name, pointing to the original track
- If there are any `trackname_display_` prefs, `tracks.db` should have an entry for the alias with (just) `_alias_for` set
- Aliases never have the same name as any existing track
- Two different tracks never have the same alias (but it is not defined which wins)
- `tags.db` has an entry for each tag, pointing to the original track

There should only be `tracks.db` entries for tracks that exist. `search.db` and `tags.db` entries should not list excess tracks or tracks that do not exist.

Operations

Notice Track

A track noticed in rescan may be new or it may not; if it already exists then the path might have changed. The length might or might not be known. The track may be already reflected in the prefs database in any case. The search database might already know about it or not in any case.

We start by opening a transaction and getting any existing `tracks.db` entry for the track.

Alias Clash: If it has an `_alias_for` entry then this this real track clashes with the alias for some existing track. We remove the `_alias_for` entry.

Path: If it has a `_path` entry then we update it if wrong, if it does not we add it.

Alias: We compute the desired alias, if any, as described below. If an alias is desired then we add a `tracks.db` record for it, with `_alias_for` pointing to the real name. (No `_path`.) This can't clash with an existing track as the alias computation below checks for that already.

Searching. We compute the set of search words for the track as described below. For each word, we check that the `search.db` entry for this word/track pair exists and if not create it.

Tags. We check that all the `tags.db` entries for each tag/track pair exists and if not create it.

Finally we store any records we changed that haven't been stored yet and commit the transaction.

Obsolete Track

We start by opening a transaction and getting any existing `tracks.db` entry for the track. If there is no such entry, or if it has an `_alias_for` entry, then we close the transaction having made no change, and stop. Otherwise...

Alias. We compute the desired alias as below. If there is one then we delete the corresponding `tracks.db` entry. We don't delete some other track's alias that beat us to it because the alias computation checks for that.

Searching. We compute the set of search words for the track as described below. For each word we delete the

`search.db` entry for this word/track pair.

Tags. For each tag we delete the `tags.db` entry for this tag/track pair.

Finally we delete the `tracks.db` entry for this track and commit the transaction.

Set/remove pref

We open a transaction and get the `prefs.db` entries for the track. If the required changeis actually no change we just commit and stop.

Aliases: If the change is to a `trackname_display_` pref then we compute the alias in terms of the new and old values. The possibilities are:

- no alias desired, no alias present. Do nothing.
- alias desired, same alias present. Do nothing.
- no alias desired, old alias present.
    - delete the `tracks.db` record for the alias
- alias desired, no alias present
    - add a `tracks.db` record for the alias, with `_alias_for` pointing to the real name (and nothing else)
- alias desired, differing alias present
    - remove the old alias's `tracks.db` record
    - add a `tracks.db` record for the alias as above

Tags: If the change is to `tags` then we compare the old and new tag lists and determine which tags have added and which removed. We add new `tags.db` tag/track pairs, or delete existing ones, as appropriate.

Searching: We compute the old and new search word list. We determine which words have been added which removed. Those removed have their track/word entries in `search.db` removed and those added have new entries made.

We then write back the `prefs.db` entry and commit.

Get pref

This is simple enough, but should actively refuse to get pref values given an alias name, as this indicates a bug elsewhere.

Resolve Alias

Again pretty simple. Aliases always point to the original track so if the result is an alias then that indicates a bug somewhere.

Garbage Collection and Length Computation

These operations go well together.

The current mechanism is to attach a callback to the main event loop which iterates over the entire database. For each track found it checks whether it is present in the filesystem and if not obsoletes it. If the track is present but is missing a `_length` entry in `tracks.db` then it computes the length - potentially an expensive process, as in some cases it implies iterating over a whole track - and stores it.

This has a couple of problems. The main one is that it can tie up the event loop with heavy CPU jobs. The other is

that the event-loop integration is horrendously. (The basic series of operations is fine, however.)

The solution to this is to run the whole rescan and recheck sequence in a niced subprocess. This is `disorder-rescan` or `server/rescan.c`. Only one instance of this program runs at once.

An implication of this is that we must enable locking in the db environment, and run an instance of `db_deadlock` (carefuly chosen to match the db version on Debian, or wrapped in our own executable). This is `disorder-deadlock` or `server/deadlock.c`.

Care must be taken to shut down the deadlock manager *after* the rescanner (and any other database-accessing subprocesses we may create).

Alias Computation

Aliases are computed from `trackname_display_PART` prefs and a pattern language. Characters other than `\` and `{` stand for themselves. `\\` and `\{` stand for `\` and `{`. Two forms of expansion are permitted, `{PART}` which just expands to the value of that part (using `prefs.db` and the `namepart` regexps) and `{/PART}` which is the same but if the result is not an empty string prefixes a `/`.

If the result does not use the database, or if it produces the same answer as the original track name, then the result is no alias.

Furthermore the resulting alias is looked up in `tracks.db`. If it is found and has a `_path` entry, or has an `_alias_for` entry pointing to a different track, then the result is no alias. Thus aliases always lose clashes with existing tracks and differing tracks with equal aliases have the first to get there win. The various callers of this routine rely heavily on this!

Database lookups for alias computation happen in the transaction specified by the caller.

Search Word Computation

The search word list for a track is the union of all the words found in the underlying track name, with the collection root stripped off, and any `trackname_display_` preferences; all with any stopwords removed.

Links

db4.3 documentation

-- RichardKettlewell - 08 May 2005