1 Hashed files

In a (basically free content) project we had to deal with tens of thousands of files. Most are in XML format, but there are also thousands of PNG, JPG and SVGimages. In large project like this, which covers a large part of Dutch school math, images can be shared. All the content is available for schools as HTML but can also be turned into printable form and because schools want to have stable content over specified periods one has to make a regular snapshot of this corpus. Also, distributing a few gigabytes if data is not much fun.

So, in order to bring the amount down a dedicated mechanism for handling files has bene introduced. After playing with a SQLITE database we finally settled on just LUA, simply because it was faster and it also makes the solution independent.

The process comes down to creating a file database once in a while, loading a relatively small hash mapping at runtime and accessing files from a large data file on demand. Optionally files can be compressed, which makes sense for the textual files.

A database is created with one of the CONTEXT extras, for instance:

```
context --extra=hashed --database=m4 --pattern=m4all/**.xml --compress
context --extra=hashed --database=m4 --pattern=m4all/**.svg --compress
context --extra=hashed --database=m4 --pattern=m4all/**.jpg
context --extra=hashed --database=m4 --pattern=m4all/**.png
```

The database uses two files: a small `m4.lua` file (some 11 megabytes) and a large `m4.dat` (about 820 megabytes, coming from 1850 megabytes originals). Alternatively you can use a specification, say `m4all.lua`:

```
return {
    { pattern  = "m4all/**.xml$", compress = true  },
    { pattern  = "m4all/**.svg$", compress = true  },
    { pattern  = "m4all/**.jpg$", compress = false },
    { pattern  = "m4all/**.png$", compress = false },
}

context --extra=hashed --database=m4 --patterns=m4all.lua
```

You should see something like on the console:

```
hashed > database 'hasheddata', 1627 paths, 46141 names,
    36935 unique blobs, 29674 compressed blobs
```

So here we share some ten thousand files (all images). In case you wonder why we keep the duplicates: they have unique names (copies) so that when a section is updated there is no interference with other sections. The tree structure is mostly six deep (sometimes there is an additional level).

Accessing files is the same as with files on the system, but one has to register a database first:

```
\registerhashedfiles[m4]
```

A fully qualified specifier looks like this (not to different from other specifiers):

```
\externalfigure
  [hashed:///m4all/books/chapters/h3/h3-if1/images/casino.jpg]
\externalfigure
  [hashed:///m4all/books/chapters/ha/ha-c4/images/ha-c44-ex2-s1.png]
```

but nicer would be :

```
\externalfigure
  [m4all/books/chapters/h3/h3-if1/images/casino.jpg]
\externalfigure
  [m4all/books/chapters/ha/ha-c4/images/ha-c44-ex2-s1.png]
```

This is possible when we also specify:

```
\registerfilescheme[hashed]
```

This makes the given scheme based resolver kick in first, while the normal file lookup is used as last resort.

This mechanism is written on top of the infrastructure that has been part of CONTEXT MKIV right from the start but this particular feature is only available in LMTX (backporting is likely a waste of time).

Just for the record: this mechanism is kept simple, so the database has no update and replace features. One can just generate a new one. You can test for a valid database and act upon the outcome:

```
\doifelsevalidhashedfiles {m4} {
    \writestatus{hashed}{using hashed data}
    \registerhashedfiles[m4]
    \registerfilescheme[hashed]
} {
    \writestatus{hashed}{no hashed data}
}
```

Future version might introduce filename normalization (lowercase, cleanup) so consider this is first step. First we need test it for a while.