Project „Multilingual Maps“

Design Documentation

Copyright 2012 by Jochen Topf <jochen@remote.org>
and Tim Alder <kolossos@toolserver.org>

This document is available under a
Creative Commons Attribution ShareAlike License (CC-BY-SA).
http://creativecommons.org/licenses/by-sa/3.0/

Table of Contents

1 Introduction....2
2 Background and Design Choices....2
2.1 Scaling....2
2.2 Optimizing for the Common Case....2
2.3 Pre-rendered Tiles....2
2.4 Delivering Base and Overlay Tiles Together or Separately....3
2.5 Choosing a Language....3
2.6 Rendering Labels on Demand....4
2.7 Language Decision and Tile URLs....5
2.8 Flexible Styling....7
2.9 Evaluation of Existing Tileserver/Rendering-Software....7
2.9.1 mod_tile/renderd/Tirex....7
2.9.2 DevelopmentSeed Rendering Stack....7
2.9.3 MapQuest Render Stack....7
3 MLM Architecture....8
3.1 HTTP Reverse-Proxies/Caches....8
3.2 Web Server and Tile Handler....9
3.3 Queue (Tile Broker)....9
3.4 Tile Storage....9
3.4.1 Long-Term Storage Using Cassandra....9
3.4.2 Short-Term Storage Using memcached....10
3.4.3 Space requirements....10
3.5 Rendering....11
3.6 Rendering Database....11
4 Miscellaneous....12
4.1 Addressing Tiles....12
4.2 Tiles and Metatiles....12
4.3 Forced Re-Render....12
4.4 Queue Re-Ordering....13
5 Appendix....13
5.1 Relevant Tags in OSM....13
5.2 How often are tiles accessed?....13
5.3 Existing Rendering Configuration....14

1 Introduction

The goal of this project is to realize a flexible and fast tile rendering and storage system for multilingual map tiles. The system should deliver OpenStreetMap tiles in any language (depending on availability of OSM data in that language of course) to a map run on Wikipedia.

2 Background and Design Choices

2.1 Scaling

The system must work efficiently on a large server cluster. Especially tile rendering and tile storage must be distributable for speed and reliability. For use at the Wikipedia one server will not be enough. But there are also other organisations running large tile servers that might be interested in the software.

On the other hand the whole system must be installable on a single host. It should be reasonably easy to install and use. This is important so that many people can play with it and/or use the system productively in smaller setups. More users mean more people finding bugs, working on improvements etc, so thats needed for a healthy Open Source project.

2.2 Optimizing for the Common Case

Once the system has been running for a while, most accesses will be to existing tiles. So this is the most common case the system must be optimized for. Just a lookups and then the delivery of the tile as directly from disk to the client as possible.

If the tile is not available it must be rendered on-demand (if server load permits). Getting the data from the database and rendering the tile is expensive compared to everything else we need to do, so it doesn't matter if those parts of the system are somewhat slower.

2.3 Pre-rendered Tiles

In a typical tileserver as they have been used for OSM data for a while, incoming requests are either satisfied from a tile store if the tile is already available or the tile is rendered on the fly, stored to the tile store and then delivered to the client. This setup can be supplemented by a HTTP cache in front to speed up access to often requested tiles.

The problem with such as „classical“ tileserver in the context of multilingual maps is that there are far too many tiles to generate and store. The current tile store for tile.osm.org uses about 1.5 TByte of tile storage. If every tile is available in 300 languages, we would need to store 450 TByte of tiles. This is, of course, out of the question.

There are two ways to solve this: We know that most tiles will be the same in all languages, either because they don't contain any labels at all or because the labels are only in one language anyway. So we could store all tiles that look the same only once and add a level of indirection when accessing those tiles.

The second option is to split the tiles into background tiles and label overlays. The overlays are much smaller, so even if they are stored separately they take less space. And the overlays are much faster to render, because they contain less information, so we might render them on demand.

2.4 Delivering Base and Overlay Tiles Together or Separately

When using base map tiles plus language overlays there are two ways we can deliver them to the clients, either as separate images or together after merging them on the server.

Merging them on the server means more work for the server but only half the HTTP requests to handle. For tiles already available the latency (ie the time before they are shown in the browser) is better, because there is less data to transfer and, crucially, less HTTP requests. For tiles that need to be rendered, perceived performance might be worse, because the server might be able to deliver the base tile and the client can show it while the overlay is still rendering. For the client merged tiles is simpler to handle and only the rare use-case where somebody changes the language he wants will be faster if only the overlay has to be re-delivered.

What is unclear is how this decision affects any HTTP caches between client and server. For n languages there are n large images to cache if images are merged on the server, and 1 large image plus n small images to cache if they are delivered separately. The latter should be better but doesn't take into account the storage overhead in the cache which might be large enough to dwarf the many small overlay images.

Delivering merged tiles is probably better, but we'll see how the operational experience is once the system is running.

2.5 Choosing a Language

One key issue of the multilingual map project is obviously how we choose which language labels to render into a map. There are two sides to consider:

1. What name tags do we have in the OSM data and
2. What is the preferred language (or the preferred languages) of the user

Side 1 seems to be simple. It is obvious that we can’t invent data thats not available. If there is no Mongolian name for an object in OSM we don’t have it. But it is a bit more complex, because not all name tags come with the information on which languages they are in. The “name” tag itself, as well as the “int_name”, the “alt_name” and some other tags don’t tell us which language they are using. Depending on the location of the object we might be able to tell which language a name tag is probably in. But sometimes people use name tags in the form “LanguageA (LanguageB)” or “LanguageA – LanguageB”, so this might be difficult.

And we have to take scripts into account. There might be a Serbian name:sr tag using Cyrillic letters, which might not be useful to a user who can’t read Cyrillic, but it can be transliterated into Latin script.

So we probably need some kind of “language detection” and “normalizing” step first, that finds all name-related tags and decides what languages and scripts are involved. This is never going to be perfect in practice, but most cases should be reasonably straightforward to decide. And we always have a good “fallback”: If in some case our algorithm doesn’t properly detect a language, we tell OSM users to add a “name:language” tag which will always overwrite whats in the generic name tags.

We do not want to burden the OSM database with unnecessary data. This „normalizing“ step can help because names don't have to be in the database in all the different dialects and scripts, but can be transliterated or otherwise transformed where this is possible to do automatically.

On the other side we have the “map owner” or the end user deciding which languages they would like. In some cases the person creating a map can decide which languages should be used. Maybe he knows the target audience well enough, maybe he wants to create a printed map for a travel guide

and only wants to print it in one language. But it our case we want the flexibility given to the user. The user should be able to choose the languages he or she wants to see. This is not only one language but a list of languages. The user might prefer his native French, but can also read Spanish or English. I'll not go into the detail how a users tells us technically which language to use, this might be by HTTP header or cookies or some other means. Lets postpone this question and just assume that the user gives us an ordered list of the languages he wants.

We know get into a few problems:

First, if we pre-render map tiles based on only one preferred language, we can't always give the users the choice he wants. Say we created the a French language map and decide to render French labels and if they are not available fall back to English and Spanish in that order. But the user might rather see the Spanish labels than the English ones. We have just given the user the wrong labels.

This problem becomes more pronounced when we consider different scripts. Most people probably only know one script. I only understand Latin letters and am completely lost when I see, say, Chinese characters. I'd rather see the Latin transliteration of a Mongolian language name of a Chinese city if that happens to be available than the original Hanzi characters. I don't speak the Mongolian language but the chances are still better that I recognize something this way.

Another problem is the question what to do if none of the languages preferred by the user is available. We could always fall back to a default language, or better, have a list of default languages to fall back to. But again, those defaults might be better if we take the languages the user did choose into account. If a user choose Spanish as one of her languages, chances are a Portugese or Italian name might be better to understand for her than, say, a German label, because how closely related those languages are.

Now to be sure, there are a lot of special cases here, but for most people a few rules are probably enough. But it is not easy to define those few rules without a lot of knowledge about the languages written around the world and the people using them. Getting those rules wrong can seriously discriminate against some people. And it could lead to even worse edit wars in OSM than we already have. So we really have to keep the system as flexible as possible.

Note that there are still cases where the map should always show the local names (basically what we have now in OSM), for instance if you need to orient youself on the ground and want to compare street signs with the map. And some OSM mappers might want to see specific tags rendered so that they can make sure that they have entered the data properly.

It is difficult enough to pre-render maps in several hundred different languages, but it is obviously not possible to pre-render maps in all the different language combinations a user might want. One obvious solution would be not to pre-render the maps but create them on-the-fly based on the users language choices. With the current technology available to us we can not render the whole map on the fly, because it would be too slow, but we can render the labels on the fly and put them on a pre-rendered base map.

2.6 Rendering Labels on Demand

Our tile rendering process is too slow to render the whole tile on demand. Tiles sometimes take several hundreds of seconds to render. It would be interesting to optimize the whole process so the whole tile can be rendered on demand, but thats a bit out of the scope for the multilingual maps project and too difficult to achieve for the moment. But it might just be possible to pre-render base map tiles and then overlay labels on top whenever a tile is accessed.

Why might this problem be solvable but not the pre-rendering of the whole tile? The difference is in the amount of data to be taken into account. Most OSM objects do not have a name. It takes a while,

for instance, to get all building polygons out of the database and render them, but buildings usually don't have names, so I don't have to take them into account for the labelling. There are only about 35 million names in the OSM database[1]. Thats not much compared to the over 1.6 billion objects in the database. On the other hand rendering labels is relatively expensive because they are not allowed to overlap.

I see two different ways this could be handled: We could either hold all the information needed for the labels in a separate database and whenever a tile is accessed get this information out and render it on top of the tile. Or we could keep the label information together with the pre-rendered base tile. So the pre-rendering process decides which labels might be needed for which tile and store this information as vector data together with the tile.

The database for the first approach would have to fit into memory, otherwise it would never be fast enough. We need to deliver thousands of tiles per second, so this process has to be really fast. How would this database be structured? The simplest way would be to use the same structure as the main rendering database, but leave out everything that doesn't have any names. This way it is easy to use the existing map styles, they just have to be split in two, one for the pre-render, one for the label overlay. The database would be updated together with the main rendering database whenever the OSM data changes. This still could lead to some problems, because the data in the pre-rendered tile might not fit with the labels rendered later.

In the second approach the job of getting the data out of the database and deciding where to render a label can already be done in the pre-render step. But instead of doing the actual rendering, the data is stored in some format together with the base map image. Maybe the Mapnik Metawriter could be used for this, but this approach might need some large changes in Mapnik. An advantage is that we always have data that fits together, because it is read from the rendering database at the same time. And because the tile has to be read from disk anway it is not much more expensive to read a bit more data. No extra database is needed.

There is one problem with both those approaches. Normally when rendering tiles, POI icons, labels, and shields should never overlap. If icons and shields are rendered into a bitmap and we add the labels later we can not make sure that there will be no overlap. One possible solution would be to somehow keep the information where there are icons and make sure we do not render into them. But this could lead to an important label not being rendered because all space is taken up by unimportant icons. Another option would be to also render icons and shields on-demand together with the labels.

Another complicating factor is the use of metatiles. Usually on tile servers the 256×256 pixel tiles are not rendered alone but in groups of (typically) 8×8 tiles. Such a group is called a metatile. There are several reasons for doing this: It is faster to render one large image than many small ones and it is more efficient in storage, too. But in this context the most important reason is the label placement. If the area rendered in one go is larger, it is easier to place labels in good positions and the maps will look better. So we somehow have to do this on-demand-rendering on the metatile and not the tile. This means we need some kind of short-term caching of the metatiles with the rendered labels and make sure all the requests coming in shortly after another for nearby tiles (and therefore the same metatile) end up finding this metatile.

2.7 Language Decision and Tile URLs

Whatever way we'll develop for rendering the multilingual tiles, we have to get the information about which language(s) the user wants from the user to the tileserver. Web browsers typically support the selection of an ordered list of preferred languages. This list is sent through the Accept-

1 With name tag, int_name, alt_name, and the many name:language tags. See Appendix for some statistics.

Language header to the web server. We could use this setting to determine the language for the labels. But in some situations people might not know about this setting or can't change it. Maybe they are sitting in an Internet cafe in a foreign country. In that case it might be easier for them if they can just change the language setting on the web page. This is especially interesting if the web site knows which languages are available and only shows those options. The browser setting doesn't know anything about actually available languages, it just has one large master list.

So we probably want the default to be the browser setting and the user can override this on the web page itself. This is how it is done in many web pages. If the user sets the language, we could either store this information in a cookie or we can put it in the tile URLs. If it is stored in a cookie the server would have to read the Accept-Language header and the cookie and determine the desired language from those settings. Instead we could aggregate the information from the Accept-Language header and the user setting inside the browser into one setting and put that into the URL.

Having the language choice in the URL has several advantages: It is easier to use the tiles for everybody not using a normal browser. Say, you request tiles from the command line with wget, or maybe from some specialized program. It is much easier to just use the URL instead of setting an Accept-Language header and a cookie. There is also the question of caching. Do all caches which might be between the server and the browser and the cache in the browser know about the Accept-Language header and the cookie and take them into account? If all information is in the URL, we do not have to think about that.

So how is the URL going to look? Lets look at a "traditional" tile URL first. Typically there is a prefix, the map style, the zoom level, x and y coordinates and the file type in the URL. Something like this:

```
/tiles/osm/3/2/1.png
```

The prefix (`/tiles/` in this case) basically tells the server that there is a tile here, it can be empty in some cases, but might be needed in others if the same server is used for other web content, too. The map style parameter (here `osm`) is configured in the server and will lead to the right Mapnik style being loaded. The other parameters are pretty much self-explanatory.

We could put the language choice in the style slot, but then we can not have different styles for the labels. And we probably need a fixed list of languages, because existing tile server software needs this parameter to be pre-configured. We could also add it the end after a question mark: `/tiles/osm/3/2/1.png?lang=en,de`. That might be problematic for caching again, as some caches might think that URLs which such parameters can not be cached.

But we can always just add another path element somewhere in the URL: `/tiles/osm/en,de/3/2/1.png`. Unfortunately we don't have a good way of asking for the default language, because leaving this element empty leaves two forward slashes after another which might be confusing.

We can always extend this scheme to allow other parameters, too. Say for choosing the tile size: `/tiles/osm/en,de/256/3/2/1.png`

This might not be the prettiest URL ever, but it'll do. We could argue about the order of the elements, maybe putting the language after the coordinates, but that doesn't really matter. A variant might make it more clear that those parameters are basically parameters to the style: `/tiles/osm;en,de;256/3/2/1.png` or something like it. But it is not all all clear whether that makes things easier or more difficult.

We should probably keep the image format in the `.png` form, because this allows software that doesn't have access the the Content-Type header to infer what format the image might be in.

One variant would be to name the attributes, something like `.../lang=de,en/style=osm/...` but this is longer and invites re-ordering of those

elements which would not work well with caching so it shouldn’t be allowed.

2.8 Flexible Styling

Multilingual maps will mostly look the same but have different labeling. This concept can possibly be extended to other cases where multiple maps should be rendered that mostly look the same, but have some differences, such as different color scheme for some features or different points of interest showing.

To make the system more flexible subsystems should, where possible, be designed to allow an arbitrary number and arbitrary types of style choices.

2.9 Evaluation of Existing Tileserver/Rendering-Software

2.9.1 mod_tile/renderd/Tirex

Most current OSM tile servers with dynamic tile rendering use the Apache mod_tile module together with either renderd or Tirex. Both solutions have been around for several years and work well.

But there are a few restrictions which make this route unsuitable going forward:

- The mod_tile Apache module depends on the setup using Apache. Compared to other web servers, Apache is relatively heavy weight and not exactly known for its high performance. Of course the dependence on Apache could be removed if a replacement for mod_tile is written.
- The whole mod_tile/renderd/Tirex setup is designed to run on a single host only. There are no provisions for distribution the tile rendering or tile delivery onto several hosts.
- Tiles are stored on disk in the file system. File system layout and the use of 8x8 metatiles is pretty much fixed. Experience on large tile servers show that storing the tiles in the file system makes expiry expensive because you need to walk the whole directory tree to find tiles to expire.

2.9.2 DevelopmentSeed Rendering Stack

The rendering stack developed by DevelopmentSeed for their MapBox product seems to be based on the assumption that all tiles can be pre-rendered. This is very different from the setup we are persuing here.

2.9.3 MapQuest Render Stack

MapQuest developed their own render stack and later released it as Open Source[2]. The software is written in C++ and based on ZeroMQ and Mongrel2. The software is designed from the ground up to be distributable without any single point of failure.

Based on discussions with the main developer Matt Amos and a look at the mostly clean source code this software seems to be a good basis for this project. There are numerous changes needed, but they fit well in the design.

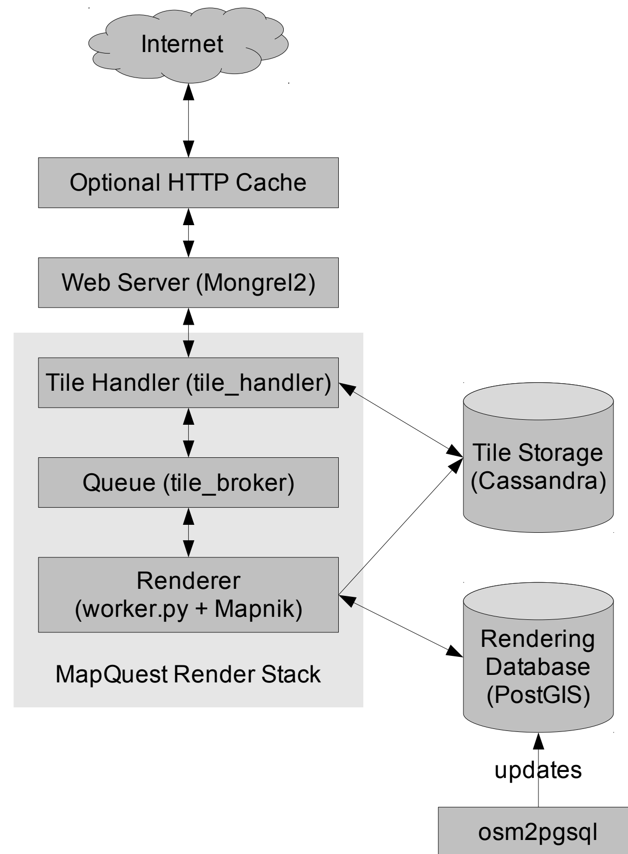
There is some documentation inside the source code but unfortunately nearly no high level documentation[3], so it is quite difficult to use the software at the moment. Part of this project must be

2 https://github.com/MapQuest/MapQuest-Render-Stack
3 https://github.com/MapQuest/MapQuest-Render-Stack/wiki/Documentation

to document the software and make it easier to use.

3 MLM Architecture

Based on the background and design and architecture choices described in the preceding chapters, we plan to implement the architecture described in this chapter.



Every component in this graph (except the osm2pgsql update of the database) can be independently scaled, ie there can be multiple daemons running on a single host or on multiple hosts. An HTTP redirector might be needed if the HTTP cache or web server is distributed on several hosts.

Note that the „Language Rendering Hint Database" envisioned in the original plan for this project is not needed in this architecture. So that part has become considerably simpler.

3.1 HTTP Reverse-Proxies/Caches

To allow maximum speed and low latencies reverse HTTP proxies are used in front of the rest of the system. This is an optional component, but it probably makes sense to optimize the most common case (see chapter 2.2). HTTP caches such as Squid[4], Apache TrafficServer[5] and Varnish[6] are optimized for quick delivery of lots of data. If needed those HTTP caches can be distributed to several locations around the world to reduce latency when accessing often used tiles.

Note that those caches need only to hold the most often accessed tiles, possibly even in memory. They can be configured to expire tiles quite aggressively (say every 10 minutes or so) making sure

4 http://www.squid-cache.org/
5 http://trafficserver.apache.org/
6 https://www.varnish-cache.org/

that tiles that are expired in the backend will not linger in the caches for a long time. This will still reduce the load on the backend considerably. The job of the caches is only to lower the peaks in the access curve, they will not hold anwhere near the whole set of tiles.

Some of those proxies support traffic- or access-rate-limiting which might also be useful.

To give an idea what these caches can store: Varnish can be configured to use RAM cache only. It has about 1kB of overhead per cached object.[7] With an average tile size of 4kB in lower zoom levels, this means we can store about 12 million tiles in 64 GB RAM. There are about 1 million tiles up to and including zoom level 10. So we can store 12 language versions of all these tiles and deliver them at wire speed from the RAM.

3.2 Web Server and Tile Handler

The Mapquest Render Stack uses the Mongrel2 web server. It implements a handler for Mongrel2 that handles tile requests. This handler has to be extended to work with the new tile URLs which contain the language choice as described in chapter 2.7. The language choice has to be added to the request thats going to later parts of the system.

The handler needs also the changes to allow different tile rendering priorities described in the next chapter.

3.3 Queue (Tile Broker)

The Mapquest Render Stack and has a built-in render queue manager called tile_broker which holds one queue in RAM. We will need some way of prioritizing requests for the labels layer. Otherwise long-running render jobs for the background layer can starve out the on-demand-rendering of the labels. To implement this we need configurable priorities for the queue (depending on the map parameter in the URL) and not the few fixed priorities currently used.

It will also need a way for rendering backends to only request jobs for a particular map style from the queue, this can be implemented by using several queues or by keeping the one queue but allowing some kind of filtered access. Details have to be hashed out with the Mapquest Render Stack developers.

3.4 Tile Storage

The **Tile Storage** stores rendered tiles and, possibly, meta information for those tiles. The Maquest Render Stack supports pluggable storage backends. For small setups on a single host tiles can be stored in the filesystem. For larger setups other storage backends are possible.

All tile storage uses 8x8 metatiles. This is the way this is currently implemented in the Mapquest Render Stack. It is unclear how easy it would be to change this, and anyway, it is a sensible choice.

The interface to the storage backend has to be changed to allow storing tiles with different map parameters (here: languages).

We will deliver two storage backend plugins for the Mapquest Render Stack, one for long-term storage using Cassandra and one for short-term storage using Memcached.

3.4.1 Long-Term Storage Using Cassandra

A long-term storage of tiles is to be developed based on the Apache Cassandra[8] database. Cassandra

7 https://www.varnish-software.com/static/book/Tuning.html#storage-backends

8 http://cassandra.apache.org/

is a mature project with high performance for reads and writes on smallish data, which metatiles are.

Cassandra is a cluster storage without a single point of failure. It handles the distributing of tiles to several servers making a configurable amount of copies. It is possible for instance, to define that low zoom level tiles will be distributed to many servers to make read access faster and to make sure the tiles will survive multiple disk or server failures. High level tiles that can be re-rendered faster and are not accessed as often can be stored without replication or with a smaller replication factor.

Cassandra supports setting a TTL (time to live) on the objects it stores which obviates the need for a tile expiry mechanism.

There are other options for such a distributed tile store. The OpenStack Storage[9] component seems to be more geared towards larger blobs of data that don't change that often. Redis[10] seems to be optimized for more complex data models and queries and it uses master/slave replication instead of the peer-to-peer of Cassandra, Hypertable[11] is based on a distributed filesystem which must be managed. Riak[12] looks like a strong contender, but seems a bit more complex with several configurable storage backends etc. If we find a problem with the Cassandra database after implementation we could try Riak.

3.4.2 Short-Term Storage Using memcached

For languages or language combinations not accessed very often, metatiles can be stored in a storage backend based on memcached[13] that is to be developed. Tiles are only stored for a short time, there is no need for explicit expiry as memcached has builtin LRU expiry.

When a tile is accessed the metatile it belongs to is rendered and all other tiles belonging to the same metatile can be accessed from the memcached.

3.4.3 Space requirements

Tile storage on tile.osm.org is about 1.5 TB for all tiles (600 GB for z0-z16, 1200 GB for z17-z18). Tiles differ a lot in size, an empty tile is only about 100 Bytes, a tile with lots of detail can easily be 20 kBytes or more, the average is somewhere around 2 to 4 kBytes, a metatile contains 64 of those, it needs on average about 200 kBytes. So we need to store on the order of 8 million metatiles. This number is, of course, growing with the success of OpenStreetMap, but it is bound by the size of the earth and the density of data.

The language overlays are mostly empty and should be re-created on the fly for all but the most often requested languages. Compared to the base tiles they will need much less space.

Low-level tiles (up to zoom level 16) must be stored securily in a RAID 10 or, better, on several hosts, because those tiles are more expensive to render and/or more often accessed. For higher zoom levels this is not as critical and tiles can be re-rendered if they are lost. (On tile.osm.org RAID10 is used for lower zoom levels and RAID0 for z17-z18.)

Setting up a cluster where three copies are held for z0-z16 base tiles and two copies for z17-18 base tiles will need about 600 * 3 + 1200 * 2 = 4800 GByte plus Overhead. Doing this on three machines for load balancing and security against failure each machine would only need say two 1.5TByte disks.

9 http://www.openstack.org/software/openstack-storage/
10 http://redis.io/
11 http://hypertable.org/
12 http://riak.basho.com/
13 http://memcached.org/

3.5 Rendering

In the Mapquest Render Stack the worker.py Python program retrieves jobs to be rendered from the Queue and renders them using Mapnik. Mapnik in turn uses the usual PostgresSQL/PostGIS database filled with Osm2Pgsql using the hstore extension to store all language variants of all names. (See chapter 3.6 below for details on this and how it can be extended.)

We have to extend worker.py to read the language choice from the job and decide which OSM tags to use when rendering labels. To get this running quickly we will simply use the right „name:LANGUAGE" tag and fall back to „name" if that is not available. This can be done with the usual database. In the future we can work with the different language communities to create more complex mapping rules, for instance using transliteration when only a Chinese name is available but English was requested.

The code for mapping from language requested to database query used will be written in Python as part of the worker.py and so it is reasonably easy to change and extend.

Compared to the usual way of rendering tiles we have two changes to the rendering with Mapnik itself:

1. There are two style files now, one for the base map and one for the language overlay. It is very simple to split up an existing style file as they are already organized in layers and all that needs to be changed is that some layers go into one style file and some in the other.
2. The language overlay style has to be changed for every request that comes in to reflect the right database lookup. This can already be done with the current Mapnik. We might need some small changes to Mapnik to make changing the datasource parameters more efficient or so as discussed here. Details need to be hashed out, but there is no big problem here and the solution should be generic enough that it would end up in the main Mapnik source.

This setup is not much different from the existing configuration (see Chapter 5.3), but there are two differences: To reduce latency for clients base tiles and language overlay tiles are joined on the server. And changing of the Mapnik configuration happens on-the-fly and this way it can support any number of language combinations.

3.6 Rendering Database

The **Rendering Database** contains the actual vector data needed for rendering. For this project we just use the usual PostgreSQL database with PostGIS extension that is populated and kept up to date with Osm2pgsql. No new software is needed.

Disk space needed is currently about 500GB. For performance, this database should be on an SSD.

For performance reasons it will probably be necessary to work with something other than the normal Osm2Pgsql tables to access the names. Once the system is running, we can evaluate and tune its performance. Possible actions to improve performance ordered by the effort needed/possible performance gains:

1. Using a special index for the names.
2. Using a separate table with a copy of only the names.
3. Putting this table in a different tablespace so it can be put on a different spindle.
4. Setting up a separate database server and rendering name overlays on that server.

Depending on the actual performance needs we'll implement the cheapest measure.

As small change would be to add a special table for labels, this table would contain for critical zoom level 2-13 all places, peaks and bodies of water. So it would be faster to render the transparent language overlays. A test gives a speed-up factor of around 8. These objects make up only 3% of the database.

It would be also possible to split all name:* from normal tags-hstore to an aditional hstore-column where the languages are the keys.

To know which languages an object has, it would be enough to save an array with all languages: „{{aa,af,al,....}}“

4 Miscellaneous

4.1 Addressing Tiles

Tiles are addressed by the zoom level (z), their coordinates (x, y) and a configurable number of additional attributes. Attributes names and values are strings out of a list of strings defined in the configuration.

Example for multilingual map: z=10, x=78, y=89, lang=en

Example for different map styles: z=8, x=13, y=14, style=dark

Allowed attribute values must be changeable without invalidating any data structures such as the tile storage. So, in the MLM case, languages can be added if need be. Attribute names might invalidate the setup. (Although it would of course be better if they don't.)

For each attribute a default value can be configured if the attribute is missing in the request.

4.2 Tiles and Metatiles

The usual tile servers deliver 256x256 pixel tiles, but render and store tiles using so-called **metatiles** that typically contain 64 tiles in a 8x8 configuration. This allows more efficient rendering and storage. And because tiles are typically accessed together with their neighbors it makes sense to store neighboring tiles close to each other.

Because adjacent tiles need to fit together, icons and labels should not be split by a tile boundary. When metatiles are used, they can be split by a tile boundary but not by a metatile boundary, allowing better placement of icons and labels. So for rendering we will use metatiles.

Storage in typical tile servers is one file per metatile. This means there are a lot less files then if each tile is stored separately. Filesystems/directories with many files are sometimes slow to access, so this is more efficient. It also reduces storage space overhead because filesystems store data in blocks and half-empty blocks waste storage space. Fewer and bigger files mean less wasted space.

4.3 Forced Re-Render

Do we need the option to force re-rendering of a cached tile from outside the system (for instance thru a special URL)? Problem: Easy way for a DOS attack.

(On Wikipedia this is probably not needed, but tile.osm.org always had this feature and it was useful on occasion.)

The Mapquest Render Stack supports this feature.

4.4 Queue Re-Ordering

Tiles are generally queued and rendered on a first-in-first-out basis, but it should be possible to re-arrange the queue to achieve faster rendering. Experiments have shown that it is faster to render level n+1 tiles under a level n tile when that tile has just been rendered (because of caching in the database and/or system).

Tiles with high rendering priorities probably have to be rendered in-order, but tiles with low priorities (ie. when no user is waiting for them) can be re-ordered. Defining such a policy is not part of this project, but the system should be flexible enough to allow it later.

5 Appendix

5.1 Relevant Tags in OSM

In mid 2012 there are about 31 million `name` tags in OSM (about 20 million of these are on highways) and about another 2 million related tags. The most common are in the following table:

31,000,000	name
700,000	name:en
306,000	alt_name
216,000	name:ja
213,000	name:ru
192,000	name:fr
172,000	int_name
164,000	name:de
157,000	name:ar
55,000	name:be
33,175,000	Sum

Note that 33 million name tags would easily fit into memory even if they are on average, say 30 bytes long. So there aren't that many tags around that need to be taken into account for the overlay label rendering.

There are also some other tags such as `name_1` that might be relevant for labelling.

Of the 600,000 `ref` tags about 80% are highway refs and therefore are relevant for shield labelling.

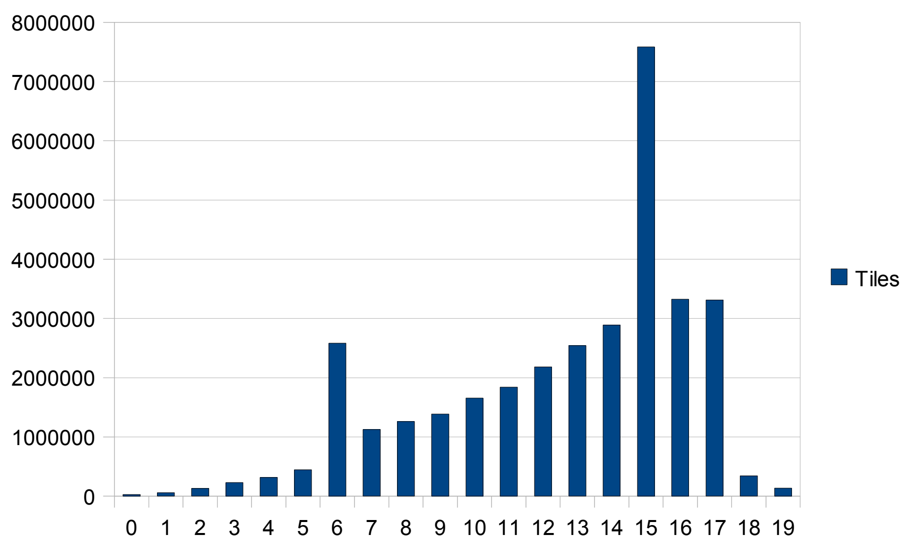
(Source: taginfo.openstreetmap.org)

5.2 How often are tiles accessed?

Stats from mercator.openstreetmap.de for one month (2012-02-17 to 2012-03-16). This was tile.openstreetmap.de at this time. Looking at the „german map style“. These statistics are not really that representative because access to map tiles in Germany is probably much more common than other areas of the planet. But hopefully it gives some insight. Similar statistics could be gathered from other tile servers.

There are about 33 million accesses to 4.7 million unique tiles. Thats about 13 tiles delivered per second on average. There are about 38 million tiles in the tile store, so most of them were never accessed in this month.

Accesses for the different zoom levels are seen here:



Zoom level 6 is the default zoom level for http://www.openstreetmap.de/karte.html, so the reason for the many accesses in this level is probably people accessing the map and then not doing any further action. It is unclear where the spike at zoom level 15 comes from.

2,613,453 of the 4,721,464 unique tiles have only been accessed once, thats about 55%.

The 2 million tiles accessed more than once only need 3 GByte of storage (at 1.5 kByte per tile). This would easily fit into a RAM cache.

5.3 Existing Rendering Configuration

We have an existing rendering configuration for multilingual maps on toolserver:

http://toolserver.org/~osm/locale/ (Thanks to mazder and avar.)

The system has two parts:

*Base style render into tiles (osm-no-labels)

The style definition is so that the base style (osm-no-labels) ist derivated from standard mapnik style. All labels are commented out.

*Additionally transparent overlays for each lang (osm-labels-{lang}). The transparent layer needs only to render labels so it's much faster.

The overlays contain some shilds (against collision) and all labels.

There is one central style definition: https://svn.toolserver.org/svnroot/p_osm/styles/osm-locale/osm-locale.xml

The SQL- queries look like this: `select way,COALESCE(tags->'name:&locale;', name) AS name from &planet;_polygon where boundary='national_park')`

The `COALESCE(tags->'name:&locale;', name)` does the fallback to standard name tag if no name:lang exist. Language is specify in variable &locale; which is the only thing different in the style files for each language.

So the existing mapnik style seem good to maintain.

Both tiles comes together in client-software.