- HARDWARE

- NOMINATIM
    - Installation steps
        - System configuration
        - Setting up Postgresql
        - Setting up the webserver
        - Installing nominatim
    - Database import
        - Flatnode files
        - Initial import of data
        - Updating the data
    - Data format of Nominatim responses

- GRAPHHOPPER
    - Deploy guide
        - Basics
        - Planet import
        - JVM tuning
        - Elevation data
    - Usage
        - Endpoint
        - Routing API
        - Pricing and credits

- SQL vs NOSQL

Hardware:

- **Hadoop cluster:**

4 Vms names: **hadoop-01 hadoop-02 hadoop-03 hadoop-04**
OS: Linux Ubuntu Mate 18.04 LT
User: dsproject
...

- **Nominatim server:**

1 Vm **nominatim-o1**
OS: Linux Ubuntu Mate 18.04 LT
User : dataproject
...

Nominatim

Nominatim is a tool to search OSM data by name and address and to generate synthetic addresses of OSM points (reverse geocoding).

Nominatim provides geocoding based on OpenStreetMap data. It uses a Postgresql database as a backend for storing the data.

There are three basic parts to Nominatim's architecture: the data import, the address computation and the search frontend.

The data import stage reads the raw OSM data and extracts all information that is useful for geocoding. This part is done by osm2pgsql, the same tool that can also be used to import a rendering database. It uses the special gazetteer output plugin in osm2pgsql/output-gazetter.[ch]pp. The result of the import can be found in the database table place.

The address computation or indexing stage takes the data from place and adds additional information needed for geocoding. It ranks the places by importance, links objects that belong together and computes addresses and the search index. Most of this work is done in Pl/pqSQL via database triggers and can be found in the file sql/functions.sql.

The search frontend implements the actual API. It takes queries for search and reverse geocoding queries from the user, looks up the data and returns the results in the requested format. This part is written in PHP and can be found in the lib/and website/ directories.

Installation steps

These instructions expect that you have a freshly installed Ubuntu 18.04.

Make sure all packages are are up-to-date by running:

```
sudo apt-get update -qq
```

Now you can install all packages needed for Nominatim:

```
sudo apt-get install -y build-essential cmake g++ libboost-dev libboost-system-dev \
                        libboost-filesystem-dev libexpat1-dev zlib1g-dev libxml2-dev\
                        libbz2-dev libpq-dev libproj-dev \
                        postgresql-server-dev-10 postgresql-10-postgis-2.4 \
                        postgresql-contrib-10 \
                        apache2 php php-pgsql libapache2-mod-php php-pear php-db \
                        php-intl git
```

If you want to run the test suite, you need to install the following additional packages:

```
sudo apt-get install -y python3-setuptools python3-dev python3-pip \
                        python3-psycopg2 python3-tidylib phpunit php-cgi

pip3 install --user behave nose
sudo pear install PHP_CodeSniffer
```

- **System Configuration**

The following steps are meant to configure a fresh Ubuntu installation for use with Nominatim. You may skip some of the steps if you have your OS already configured.

Creating Dedicated User Accounts

Nominatim will run as a global service on your machine. It is therefore best to install it under its own separate user account. In the following we assume this user is called nominatim and the installation will be in /srv/nominatim. To create the user and directory run:

```
sudo useradd -d /srv/nominatim -s /bin/bash -m nominatim
```

You may find a more suitable location if you wish.
To be able to copy and paste instructions from this manual, export user name and home directory now like this:

```
export USERNAME=nominatim
export USERHOME=/srv/nominatim
```

Never, ever run the installation as a root user.
Make sure that system servers can read from the home directory:

```
chmod a+x $USERHOME
```

- **Setting up PostgreSQL**

Tune the postgresql configuration, which is located in `/etc/postgresql/9.5/main/postgresql.conf`.

PostgreSQL tuning

You might want to tune your PostgreSQL installation so that the later steps make best use of your hardware. You should tune the following parameters in your postgresql.conf file.

```
shared_buffers (2GB)
maintenance_work_mem (10GB)
work_mem (50MB)
effective_cache_size (24GB)
synchronous_commit = off
checkpoint_segments = 100 # only for postgresql <= 9.4
checkpoint_timeout = 10min
checkpoint_completion_target = 0.9
```

The numbers in brackets behind some parameters seem to work fine for 32GB RAM machine. Adjust to your setup.
For the initial import, you should also set:

```
fsync = off
full_page_writes = off
```

Don't forget to reenable them after the initial import or you risk database corruption. Autovacuum must not be switched off because it ensures that the tables are frequently analysed.

- **Setting up the Webserver**

The website/ directory in the build directory contains the configured website. Include the directory into your webbrowser to serve php files from there.

Configure for use with Apache

Make sure your Apache configuration contains the required permissions for the directory and create an alias:

```
<Directory "/srv/nominatim/build/website">
  Options FollowSymLinks MultiViews
  AddType text/html   .php
  DirectoryIndex search.php
  Require all granted
</Directory>
Alias /nominatim /srv/nominatim/build/website
```

/srv/nominatim/build should be replaced with the location of your build directory.
After making changes in the apache config you need to restart apache. The website should now be available on http://localhost/nominatim.

Restart the postgresql service after updating this config file.

```
sudo systemctl restart postgresql
```

Finally, we need to add two postgres users: one for the user that does the import and another for the webserver which should access the database for reading only:

```
sudo -u postgres createuser -s $USERNAME
sudo -u postgres createuser www-data
```

Setting up the Apache Webserver

You need to create an alias to the website directory in your apache configuration. Add a separate nominatim configuration to your webserver:

```
sudo tee /etc/apache2/conf-available/nominatim.conf << EOFAPACHECONF
<Directory "$USERHOME/Nominatim/build/website">
  Options FollowSymLinks MultiViews
  AddType text/html   .php
  DirectoryIndex search.php
  Require all granted
```

```
</Directory>

Alias /nominatim $USERHOME/Nominatim/build/website
EOFAPACHECONF
```

Then enable the configuration and restart apache

```
sudo a2enconf nominatim
sudo systemctl restart apache2
```

- **Installing Nominatim**

Building and Configuration

Get the source code for the release and change into the source directory

```
cd $USERHOME
wget https://nominatim.org/release/Nominatim-3.2.0.tar.bz2
tar xf Nominatim-3.2.0.tar.bz2
cd Nominatim-3.2.0
```

The code must be built in a separate directory. Create this directory, then configure and build Nominatim in there:

```
mkdir build
cd build
cmake ..
make
```

You need to create a minimal configuration file that tells nominatim where it is located on the webserver:

```
tee settings/local.php << EOF
<?php
 @define('CONST_Website_BaseURL', '/nominatim/');
EOF
```

Nominatim is now ready to use.

DATABASE import

The following instructions explain how to create a Nominatim database from an OSM planet file and how to keep the database up to date. It is assumed that you have already successfully installed the Nominatim software itself

- **Flatnode files**

If you plan to import a large dataset (e.g. Europe, North America, planet), you should also enable flatnode storage of node locations. With this setting enabled, node coordinates are stored in a simple file instead of the database. This will save you import time and disk storage. Add to your settings/local.php:

```
@define('CONST_Osm2pgsql_Flatnode_File', '/path/to/flatnode.file');
```

Replace the second part with a suitable path on your system and make sure the directory exists. There should be at least 40GB of free space.

- **Initial import of the data**

Important: first try the import with a small excerpt, for example from **Geofabrik.**
Download the data to import and load the data with the following command:

```
./utils/setup.php --osm-file <data file> --all [--osm2pgsql-cache 28000] 2>&1 | tee setup.log
```

The --osm2pgsql-cache parameter is optional but strongly recommended for planet imports. It sets the node cache size for the osm2pgsql import part (see -C parameter in osm2pgsql help). As a rule of thumb, this should be about the same size as the file you are importing but never more than 2/3 of RAM available. If your machine starts swapping reduce the size.
Computing word frequency for search terms can improve the performance of forward geocoding in particular under high load as it helps Postgres' query planner to make the right decisions. To recompute word counts run:

```
./utils/update.php --recompute-word-counts
```

This will take a couple of hours for a full planet installation. You can also defer that step to a later point in time when you realise that performance becomes an issue. Just make sure that updates are stopped before running this function.

- **Updating the data**

There are many different possibilities to update your Nominatim database. The following section describes how to keep it up-to-date with Pyosmium. For a list of other methods see the output of ./utils/update.php --help.

- **Installing the newest version of Pyosmium**

It is recommended to install Pyosmium via pip. Run (as the same user who will later run the updates):

```
pip install --user osmium
```

Nominatim needs a tool called pyosmium-get-updates, which comes with Pyosmium. You need to tell Nominatim where to find it. Add the following line to your settings/local.php:

```
@define('CONST_Pyosmium_Binary', '/home/user/.local/bin/pyosmium-get-changes');
```

The path above is fine if you used the --user parameter with pip. Replace user with your user name.

- **Setting up the update process**

Next the update needs to be initialised. By default Nominatim is configured to update using the global minutely diffs.
If you want a different update source you will need to add some settings to settings/local.php. For example, to use the daily country extracts diffs for Ireland from geofabrik add the following:

```
// base URL of the replication service
@define('CONST_Replication_Url', 'https://download.geofabrik.de/europe/ireland-and-northern-ireland-updates');
// How often upstream publishes diffs
@define('CONST_Replication_Update_Interval', '86400');
// How long to sleep if no update found yet
@define('CONST_Replication_Recheck_Interval', '900');
```

To set up the update process now run the following command:

```
./utils/update.php --init-updates
```

It outputs the date where updates will start. Recheck that this date is what you expect.
The --init-updates command needs to be rerun whenever the replication service is changed.

- **Updating Nominatim**

The following command will keep your database constantly up to date:

```
./utils/update.php --import-osmosis-all
```

(Note that even though the old name "import-osmosis-all" has been kept for compatibility reasons, Osmosis is not required to run this - it uses pyosmium behind the scenes.)

Data formats of Nominatim responses

JSON
JSONv2
GeoJSON
Geocode JSON
XML

Graphhopper

GraphHopper is an open-source routing library and server written in Java and provides a web interface called GraphHopper Maps. As well as a routing API over HTTP.

Deploy guide

- **Basics**

For simplicity you could just start jetty from maven and schedule it as background job: ./graphhopper.sh -a web -i europe_germany_berlin.pbf -d --port 11111. Then the service will be accessible on port 11111.
For production usage you have a web service included. Use -c config.yml in the script to point to it. Increase the -Xmx/-Xms values of your server server e.g. for world wide coverage with a hierarchical graph do the following before calling graphhopper.sh:

```
export JAVA_OPTS="-server -Xconcurrentio -Xmx17000m -Xms17000m"
```

- **Import a planet file**

GraphHopper is able to handle coverage for the whole OpenStreetMap road network. It needs approximately 22GB RAM for the import (CAR only) and ~1 hour (plus ~5h for contraction). If you can accept slower import times this can be reduced to 14GB RAM - you'll need to set datareader.dataaccess=MMAP
Then 'only' 15GB are necessary. Without contraction hierarchy this would be about 9GB.
With CH the service is able to handle about 180 queries per second (from localhost to localhost this was 300qps). Measured for CAR routing, real world requests, at least 100km long, on a

linux machine with 8 cores and 32GB, java 1.7.0_25, jetty 8.1.10 via the QueryTorture class (10 worker threads).

- **System and JVM tuning**

Especially for large heaps you should use -XX:+UseG1GC. Optionally add -XX:MetaspaceSize=100M.
Avoid swapping e.g. on linux via vm.swappiness=0 in /etc/sysctl.conf.

- **Elevation Data**

If you want to use elevation data you need to increase the allowed number of open files. Under linux this works as follows:

- sudo vi /etc/security/limits.conf
- add: * - nofile 100000 which means set hard and soft limit of "number of open files" for all users to 100K
- sudo vi /etc/sysctl.conf
- add: fs.file-max = 90000
- reboot now (or sudo sysctl -p; and re-login)
- afterwards ulimit -Hn and ulimit -Sn should give you 100000

Usage example:

- **Endpoint**

Our endpoint is https://graphhopper.com/api/[version]/vrp
Graphopper can provide up to 6 API: Routing API, Route Optimization API, Isochrone API, Map Matching API, Matrix API ZGeocoding API.

- **Routing API**

Get distance between two points:
curl "https://graphhopper.com/api/1/route?point=51.131,12.414&point=48.224,3.867&vehicle=car&locale=de&key=[YOUR_KEY]"
Example in our case:
https://localhost:11111/1/route
?point=51.131,12.414&point=48.224,3.867&vehicle=car&locale=de&key=3160e710-58ed-45da-bb39-09d383b1c5b2

- **Pricing and credits**

The GraphHopper routing engine is Open Source under the permissive Apache License and is therefor free to use for anything. You could even integrate it in your products, modify GraphHopper and sell this, without noticing or contributing back. Although it is encouraged to contribute back so that your feature gets maintained for free by graphhopper service. Also you can host GraphHopper on your own servers for 'free' and do whatever you want with it.
The GraphHopper Directions API that we host falls under usage terms and always requires an API key. It was decided to make it free for development purposes and Open Source projects, both with a limit of currently 500 queries per day. So, the free usage of the API in a company internally would not be allowed. But there are custom packages possible.

One Routing API request costs 1 credit. Every 10 via-points cost 1 more credit. E.g. 11 via-points cost 2 credits, 21 via-points costs 3 credits and so on. And if you specify optimize=true the credits will be multiplied by 10 i.e. one requests costs 10 credits for 1 to 10 locations, 20 credits for 11 to 20 locations and so on. Changing the parameter algorithm costs additionally 2 credits, i.e. calculating the alternative route between two points costs 1+2=3 credits.
For more info about the other APIs:
https://graphhopper.com/api/1/docs/FAQ/

SQL vs NO_Sql

SQL: Structured Query Language (SQL) happens to be the more structured, rigid way of storing data, like a phone book. For a relational database to be effective, you'll have to store your data in a very organized fashion. SQL databases remain popular because they fit naturally into many venerable software stacks, including LAMP and Ruby-based stacks. These databases are widely supported and well understood, which could be a major plus point if you run into problems.

The main problem with SQL is scaling it as your database grows. You see, even though scalability is usually tested in production environments, it's often lower than NoSQL databases.

NoSQL: If your are dealing with massive amounts of unstructured data and your data requirements aren't clear at the outset, you probably don't have the luxury of developing a relational database with a clearly defined schema. You get much more flexibility than its traditional counterparts, with non-relational databases. Picture non-relational databases as file folders, assembling related information of all types.

The needs and the volume and variety of data consumption will dictate the choice between SQL and NoSQL.