# An alternative approach to OpenStreetMap

Stefan de Konink
stefan@konink.de

January 6, 2009

## 1 Abstract

We explore the performance implications of running OpenStreetMap on executable system code with the MonetDB5 database as database storage. Our preliminary results show the performance increase over typical interpreted language for processing are significant, query time for specific feature selection declined. The amount of dependencies for running a personal server were reduced.

## 2 Introduction

OpenStreetMap is a geographical database based on co-creation and the import of geographical data available in the public domain. The project in its current form (December 2008) is good for 6GB of compressed XML data, called the 'planet', this expands to 72GB of raw CSV data consisting of nodes, ways and relations. This topological form allows relations to have all the previous as members. Ways can take nodes as members. A node is a point on WGS84 projection separated in longitude and latitude. Each of the objects allow to be enhanced with tags, consisting of key- and value-pairs. With some trivial overhead for changes and users.

Within the community there are several challenges, the format is free, and so diverse are its implementations. Currently two implementations are broadly known, the actual 'server' MySQL implementation where the proprietary API runs from using the de facto schema, and a 'Mapnik' PostGIS implementation that runs on a table that exploits binary columns as storage for used tags in a rendering task.

We will add a new solution that is currently known as handler_osm. An implementation of a readable and writable OSM server implemented as a native extension to the Cherokee Webserver and using a connection pooled MAPI interface to MonetDB5. Schemawise we have improved the design in favour of normalisation and propose to drop 'ways' in favour of a more general 'relation', appropriately tagged. For compatibility reasons, the way sequence collides with the relation sequence, thus we test our work with this extra table.

While the current server implementation is able to keep track of the users changes, the lack of foreign key constraints and transactions makes its output under many circumstances inconsistent, the OSM community tries to overcome this with a more recent API version 0.6.

Our comparison will implement and explore what known is as the 'current' view. This will ignore history, but allows future comparisons with other solutions at also primary operation on recent data.

## 3 Implementation

We will discuss different implementations in order to come from data to the user and back. Thus we need implement an importing application, the API used by OSM editors and tools and an interface to a render machine so we are able to generate maps.

### 3.1 osmparser

In order to import pseudo-GIS data into a database some solutions are available from the OSM community. Popular ones are osm2pgsql and osmosis, the first is a perl script to convert OSM to PostGIS, the second a java implementation, as Swiss army knife for conversion. Both of them can do streaming operations, thus do not require the extra diskspace required for decompression.

With enough space available it seems that the overhead of parsing and decompression at the same time could be avoided. Our parser is build around a memory mapped planet file that is raw parsed using a tokenisation based state machine. We produce separate files having comma separated values for each table. This results in a set of commands to import these files into mclient as one transaction using copy into.

We have noticed that the enforcement of sequences, primary and foreign keys significantly influences the memory requirements for the import task. Secondary, the current planet has referential integrity problems, due to the chosen generation. Direct enforcement of foreign key constraints would fail the import. After the import the integrity must be checked in order to find violations and purge them so we can alter the table to add constraints and allow normal operation.

### 3.2 handler_osm

The Cherokee webserver is a high performance, high availability webserver platform in C build around a plugin based infrastructure. Next to common interface to CGI-like protocols it allows a developer to incorporate a module inside the Cherokee core hand have the webserver to dispatch a request to a specific plugin, or in slang, handler.

The general plugin is implemented having at least:

- configure method, upon creation configuration parameters and handler wide variables are initialised in this function.
- props_free method, is set by the configure function to clean up the variables after unloading.
- new method, is called upon using the handler for a new request.
- free method, is requested for cleanup after a request.
- init method, when a new request comes in it is processed within this function.
- add_headers method, based on the internal representation of the content in the content buffer a header should be set, common properties as content-type are set here.
- step method, the webserver copies a buffer from a specific length from the handler to the requester until the entire buffer is empty, or the connection is terminated.

We configure the webserver in read/write,for updates to the dataset, or read-only mode, as data query apparatus.

#### 3.2.1 Connection Pool

To reduce the overhead of the client access layer to database, we have implemented a connection pool. Per request it saves up to 1 second over normal per request started connections, thus increases the throughput in a webservice.

The pool is implemented as thread-safe linked list, and has a fixed percentage of overhead connections, that is dynamically maintained to reduce the chance of a client to wait for the connection setup. Upon free, the current amount of free connections is evaluated, if at that moment in time it exceeds the overhead, that connection is freed.

#### 3.2.2 Operations

In order to do a comparison between the current production API in Ruby on Rails with

MySQL and handler_osm as Cherokee extension in C using MonetDB5, API0.5 had to be implemented. We have ignored the user management, GPX (pointclouds) and history part. We did implemented the additional XAPI that allows users too query the database in a pseudo XPath-attribute selection style.

Commonly used queries;

- GET map by boundingbox. The core business of OSM returns vector data, currently all data present in the database filtered within a rectangle. Since OSM data is randomly added, the performance on this retrieval code, solely depends on clever indexing. This might be the only query within OSM that could justify spatial extensions, since it is operating on the fundamental level of the data model that is mapping points to nodes.

- GET object (full). Each object can be fetched using the REST based API based on their ID. While the current production API only partly enforces referential integrity, relations in database sense, foreign keys are heavily used to map nodes to ways, and join different objects to one big relation. The full modifier will return all objects that are referenced within one output.

- PUT object by ID. The update and creation mechanism lives by lives over three tables with each having their own sequence.

### 3.3 Mapserver

Mapserver is a GIS render platform that makes extensive use of GDAL/OGR to interface with different GIS formats. We have extended the OGR virtual tables to allow linestrings to be generated from the OSM tables without specific geodatatypes, next to the already supported points. The geodatatypes are on the fly created using OGR. We have interfaced with OGR using the MonetDB ODBC connection.

## 4 Results

Our benchmarks were done with the CVS version of MonetDB5 (Dec 2008) using the new default Algebra code having debug and asserts. The system was an AMDX2 5200+, 8GB RAM running Linux 2.6.27. The dataset used that represented The Netherlands, and used around 1.8GB of memory for memory mapping the tables. The database server for the full dataset located in London is equipped with an identical CPU/Memory configuration, but balances its MyISAM and InnoDB tables over different harddrives. Opposed to our test that is limited to a single harddrive and could only be increased in throughput using a RAID configuration or logical mapping of diskspace. Since The Netherlands dataset is smaller than the total amount of memory available, the task we compare is only limited to the CPU. This measures the performance of the algorithms opposed to a complete dataset that could pose other bottlenecks such as storage disk speed.

### 4.1 Import

The conversion of 1.4GB worth of XML data to CSV using one harddrive for the read operations and one harddriver for the final files took:

| | |
|---|---|
| real | 48.800s |
| user | 40.731s |
| sys | 3.252s |

The import task was done on a tableset that did not enforce foreign or primary keys, this reduces the amount overhead due to constraint tracking and secondly will give better chances to successfully loaded. Left the server resource usage right the mclient usage. In total 27456818 rows over 10 tables where copied into the database.

| | | | |
|---|---|---|---|
| real | 1m48.891s | real | 1m44.425s |
| user | 2m11.940s | user | 0m0.008s |
| sys | 0m4.700s | sys | 0m0.000s |

### 4.2 Integrity

The current planet export files in XML lack the referential integrity or the strict enforcement of tag key uniqueness we try to achieve. This forces us to import our database constraintless and find the violators manually. For this task we have compared:

```
SELECT DISTINCT relation
FROM relation_members_way
WHERE to_way NOT IN
        (SELECT id FROM ways);

SELECT DISTINCT
        relation_members_way.relation
FROM relation_members_way
LEFT JOIN ways ON
        relation_members_way.to_way = ways.id
WHERE ways.id IS NULL;
```

The execution times of both queries after several runs results in:

| | | |
|---|---|---|
| best (nc) | 172.693ms | 160.851ms |
| best (c) | 182.898ms | 18.816ms |

The query has no output.

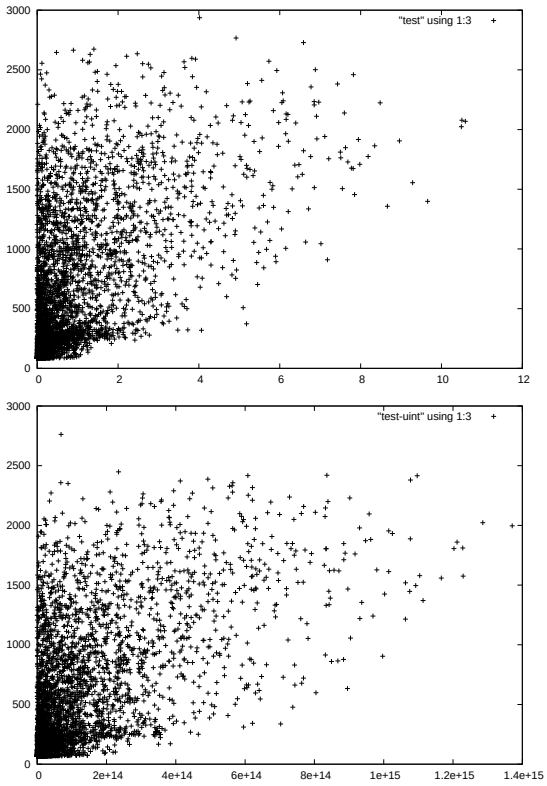
### 4.3 Bounding Box retrieval

The most used queries within OpenStreetMap fetch a specific area from the data set. We compare different SQL queries and datatypes for this task. In principle it can be denoted as:

```
lat BETWEEN y1 AND y2 AND
long BETWEEN x1 AND x2
```

We compare the area it is spanning plus the result count and relate this to the time it takes. We make a comparison between integer and doubles. GIS point tests resulted in full table scans, and took far more time for each SELECT than any other format. Our test took generated random bounding boxes withing the bounding box of The Netherlands, and summed the results using count(*). This resulted in the following scatter plots with on the Y-axis the time in milliseconds and on the X-axis the surface.

The output of doubles performs worse over larger areas; because we didn't show the amount of results found in each query it is not directly comparable but does show the reader it might be worth investigating unsigned integers further, while it does not outperform doubles by significant numbers.




### 4.4 Lookup

The lookup of tags (related to objects), ways within a bounding box (related to nodes) is compared for performance differences in recycling subqueries using IN by comparing (EQUI)JOIN, LEFT JOIN and IN.

Our test will simulate an XAPI request for a specific tag pair to look up a street name.

```
SELECT count(*) FROM ways, way_tags
WHERE
        ways.id = way_tags.way AND
        k = 'name' AND
        v = 'Oude Trambaan';

SELECT count(*) FROM ways
LEFT JOIN way_tags ON
        ways.id = way_tags.way
WHERE
        k = 'name' AND
        v = 'Oude Trambaan';
```

```
SELECT count(*) FROM ways
WHERE id IN (
SELECT way FROM way_tags
WHERE
        k = 'name' AND
        v = 'Oude Trambaan' );
```

| | | | |
|---|---|---|---|
| best (nc) | 1.660ms | 7999.047ms | 1.766ms |
| best (c) | 2.392ms | ERROR | 1.735ms |

This specific results show that the performance for IN with a relatively low amount of tuples (42) is still comparable to JOIN. We describe the error in the non-constrained version in the discussion of this essay.

The $\{highway, secondary\}$ key/value-pair represent 108992 tuples which results in the following latency:

| | | | |
|---|---|---|---|
| best (nc) | 240.106ms | 7567.258ms | 319.194ms |
| best (c) | 113.410ms | ERROR | 329.635ms |

A decrease in IN performance over a normal JOIN is observed. Which could give us a hint about the amount of tuples it is still worth to cache results. Finally we will compare both keys together.

```
SELECT count(*) FROM ways,
        way_tags AS w1,
        way_tags AS w2
WHERE
        ways.id = w1.way AND
        ways.id = w2.way AND
        w1.k = 'highway' AND
        w1.v = 'secondary' AND
        w2.k = 'name' AND
        w2.v = 'Oude Trambaan';

SELECT count(*) FROM (
        SELECT id FROM ways
        LEFT JOIN way_tags ON
                ways.id = way_tags.way
        WHERE
                k = 'highway' AND
                v = 'secondary'
) AS q1
LEFT JOIN way_tags ON
        q1.id = way_tags.way
WHERE
        k = 'name' AND
        v = 'Oude Trambaan';
```

```
SELECT count(*) FROM ways
WHERE id IN (
        SELECT way FROM way_tags
                WHERE k = 'highway' AND
                v = 'secondary' AND
                way IN (
        SELECT way FROM way_tags
        WHERE k = 'name' AND
        v = 'Oude Trambaan'
                )
);
```

| | | | |
|---|---|---|---|
| best (nc) | 242.235ms | 8521.733ms | 101.336ms |
| best (c) | 135.907ms | ERROR | 108.973ms |

In the more complex scenario something interesting going on using the nested IN; when the order is chosen right heuristically the query is able to output perform a standard JOIN. For this query worst case scenario where both are flipped 181.766ms is achieved.

## 5 Discussion

In preliminary tests we have experimented with MonetDB4 and XQuery on the dataset. The usability of the system, even at the XQuery commandline was bad. In order for the system to work a 200% overhead of diskspace was needed, and search times measured where far higher than the current 'official' OSM implementation. It seems to be clear that structured data can be stored in many ways, and it is up to the database system to choose the appropriate form. MonetDB4, at the time of writing, does not exploit DTDs or XSDs to enhance the disk format or increase the lookup performance. The idea of just serving subtrees of a document using an XQuery interface seems, from a esthetic point of view, a good idea, sadly its implementation seems to be the main bottleneck.

OpenStreetMap development is doing a fairly good job in inventing solutions that will break the fundamental SQL92 principles. The visible attribute on the 'current' table might be the best example of overengineering at the sacrifice SQL features. The scope of the attribute is to be able

to mark an object as 'deleted'. Since this does not enforce foreign key violations, opposed to a real DELETE, thus undermines the methods that where put in SQL92 to overcome integrity issues. The solution that is solved there has to do with the lack of another feature in SQL, transaction management over time. To allow to browse the history, a perfect DBMS for OSM should implement look ups based on time. It would implement a journal, like in many copy-on-write filesystems is present, to go back to an older write state.

The importing mechanism described as osm-parser was used on many systems, it became clear that when MonetDB5 was used the amount of memory required to successfully import the full Planet had to be extended in all cases to virtual memory. The biggest system, an 8 way Xeon with 32GB of memory, also used in the MonetDB4 task, also ran into out-of-memory elimination. It is questionable if this in an optimal scenario is required, or better if a reduction of consumed memory could not lead to a better throughput. Nevertheless should MonetDB facilitate its own needs by memory mapping opposed to memory allocation.

Other improvements that would benefit the OSM dataset are the reuse of query results within a query, or even within a transaction. In the actual implementation of the API we have implemented a query series that stores and reuses results (using the earlier described IN method) opposed to queries that execute a deeper query again. For smaller results the IN method is beneficial as seen from the results. Even heuristically speaking IN outperforms a normal JOIN when the sets are small enough. A database application must be able to trust the database engine that optimal strategies are chosen. This is currently not the case. Therefore our application is unable to optimise to MonetDB from a standpoint that all sizes perform linear.

After adding our constraints the LEFT JOIN operation broke; when trying to verify the actual results the foreign key index broke, and another bug was discovered.

# 6 Further Work

We have not done a direct comparison between two large datasets. Or done any end to end comparisons or benchmarks under load over our API, opposed to the Ruby on Rails API. This is an area to be explored in a later essay. An open issue is the current area creation in a virtual table. We have created code for supporting area from resultsets, but the actual rendering did not succeed.

Cherokee implements its own version of DB-Slayer, currently only MySQL is supported. It must be possible to add a native MAPI interface to this handler to allow loadbalancing using the Cherokee balancer infrastructure.

# 7 Conclusion

MonetDB is clearly capable in handling large amounts of data if sufficient hardware is thrown to the task. MonetDB is experimental, and has areas that we would like to use such as 'geom' but are not ready for large usage due to memory leaks. Since that is rather a technical malfunction opposed to a design flaw the future might show that storage of geotables could actually be an attractive and painless operation.

# 8 References


http://monetdb.org/
http://wiki.openstreetmap.org/wiki/WMS_Server
http://wiki.openstreetmap.org/wiki/Cherokee/MonetDB_Handler_OSM
http://code.nytimes.com/projects/dbslayer
http://cherokee-project.com/
http://gdal.org/