BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

TOR NETWORK CONSENSUS DATA STORED IN OLAP DATABASE

OPTIMALIZACE ULOŽENÍ DAT O SÍTI TOR POMOCÍ OLAP

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR Bc. MICHAL CHOMO
AUTOR PRÁCE

SUPERVISOR Ing. LIBOR POLČÁK, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

21404

Student:	**Chomo Michal, Bc.**
Program:	Informační technologie Obor: Informační systémy
Název:	**Optimalizace zpracování dat o síti Tor pomocí OLAP**
	Tor Network Consensus Data Stored in OLAP Database
Kategorie:	Databáze

Zadání:

1. Seznamte se s *Tor directory protocol*, zveřejňovanými deskriptory popisující stav sítě Tor, volnou databází MaxMind a nástrojem consensus_parser vznikajícím v rámci projektu Tarzan.
2. Seznamte se s podporou pro OLAP ve volně dostupných databázích (např. Maria DB, PostgreSQL).
3. Dle pokynů vedoucího navrhněte zpracování dat vhodné pro data vytvářená nástrojem consensus_parser s využitím OLAP a vybranou databází. Zaměřte se na optimalizaci vyhledávání podle multidimenzionálních kritérií včetně rozsahů IP adres a temporálních dotazů.
4. Urychlete nástroj consensus_parser s využitím navrženého modelu a rozšiřte množinu podporovaných dotazů.
5. Otestujte výkonnost vytvořeného řešení na různých typech dotazů.
6. Práci vyhodnoťte a navrhněte možná zlepšení.

Literatura:

- LACKO, Luboslav. *Databáze: datové sklady, OLAP a dolování dat s příklady v Microsoft SQL Serveru a Oracle*. Brno: Computer Press, 2003, 486 s. 1 elektronický optický disk. ISBN 8072269690.
- GOLFARELLI, Matteo a RIZZI, Stefano. *Data warehouse design: modern principles and methodologies*. New York: McGraw-Hill, 2009, xxi, 458 s. : il. ISBN 978-0-07-161039-1.
- GALLINUCCI, Enrico, GOLFARELLI, Matteo, RIZZI Stefano, ABELLÓ Alberto a ROMERO Oscar. Interactive multidimensional modeling of linked data for exploratory OLAP. *Information Systems*, č. 77. Elsevier, 2018, str. 86-104. ISSN 0306-4379.
- The Tor Project. *Tor directory protocol, version 3*. Dostupné online https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce:	**Polčák Libor, Ing., Ph.D.**
Vedoucí ústavu:	Kolář Dušan, doc. Dr. Ing.
Datum zadání:	1. listopadu 2018
Datum odevzdání:	22. května 2019
Datum schválení:	29. října 2018

Abstract

Tor is a distributed network providing privacy and anonymity on the Internet. Information about Tor is publicly available in a form of consensus documents. Existing tools that are able to display this information do not provide both historical and detailed view of it. A tool named Consensus Parser that provides a detailed, historical view of this information and extends it with geolocation and DNS information, was created as a part of TARZAN research project at Brno University of Technology. It stores the information in regular files on disk and makes it accessible via REST API. This thesis extends Consensus Parser with MariaDB ColumnStore database with a schema designed to conform to OLAP needs. The searching capabilities of Consensus Parser were enhanced by adding 109 new endpoints to 12 existing ones and adding the ability to limit the retrieved information to certain fields only. Disk space needed for storing the information was reduced by a factor of five.

Abstrakt

Tor je distribuovaná sieť, ktorá poskytuje súkromie a anonymitu na internete. Informácie o nej sú verejne dostupné vo forme consensus dokumentov. Existujúce nástroje na zobrazovanie týchto informácií ich nedokážu zobraziť detailne a zároveň z ktoréhokoľvek bodu v čase. Nástroj Consensus Parser, vyvinutý v rámci výskumného projektu TARZAN na Vysokom učení technickom v Brne, to dokáže a navyše obohatí tieto informácie o geolokačné informácie a DNS záznamy. Consensus Parser tieto informácie ukladá do súborov na disku a umožňuje k nim prístup cez RESTové API. Táto diplomová práca rozširuje Consensus Parser o databázu MariaDB ColumnStore so schémou navrhnutou tak, aby spĺňala požiadavky na OLAP. Vyhľadávacie možnosti Consensus Parseru boli rozšírené pridaním 109 endpointov k 12 existujúcim a pridaním možnosti obmedziť hľadané informácie iba na špecifické polia. Miesto na disku potrebné na uloženie dát bolo redukované na pätinu.

Keywords

Tor, OLAP (Online Analytical Processing), MariaDB ColumnStore, REST API, GeoLite2

Kľúčové slová

Tor, OLAP (Online Analytical Processing), MariaDB ColumnStore, REST API, GeoLite2

Reference

CHOMO, Michal. *Tor Network Consensus Data Stored in OLAP Database.* Brno, 2019. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Libor Polčák, Ph.D.

Tor Network Consensus Data Stored in OLAP Database

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Ing. Libor Polčák, PhD. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Michal Chomo
May 21, 2019

Acknowledgements

I want to thank my supervisor Ing. Libor Polčák, PhD. for his advice and time.

Contents

Glossary

API Application Programming Interface. 3, 11–14, 16, 32, 37–40, 42

CIDR Classless Inter-Domain Routing. 14, 33, 35
CSV Comma-separated values. 13, 14, 34, 35, 42–44, 46, 47

DBMS Database Management System. 17, 20, 21
DDL Data Definition Language. 22, 23
DML Data Modification Language. 22, 23
DNS Domain Name System. 8, 12, 13, 16, 36, 39

HTML Hypertext Markup Language. 14, 37
HTTP Hypertext Transfer Protocol. 8, 10, 11

JSON JavaScript Object Notation. 11, 14, 37

OLAP Online Analytical Processing. 3, 17, 18, 21, 23, 24, 49
OLTP Online Transaction Processing. 3, 17, 18, 23, 26

REST Representational state transfer. 11, 12, 14, 16, 32, 37, 40

SHA-1 Secure Hash Algorithm 1. 9, 10, 29
SQL Structured Query Language. 17, 22, 32, 34–37, 47

TCP Transmission Control Protocol. 7, 22
TLS Transport Layer Security. 6–8

URL Uniform Resource Locator. 12, 14, 39, 40, 46

YAML YAML Ain't Markup Language. 32, 34

Chapter 1

Introduction

Privacy and security on the Internet are becoming more and more important now that governments, organizations and companies track, collect and analyse data about the Internet users. Being anonymous on the Internet is a very difficult task for regular people without in-depth knowledge of computers and computer networks. By using Tor, a service that provides anonymity on the Internet, users can protect their privacy, circumvent censorship or freely express their opinion without fear. But even Tor can be compromised by a party with enough resources. This may be prevented by having means to analyse data about the Tor network itself, gaining better visibility and more complete view of the network.

Statistical information about the Tor network is publicly available and it is continuously updated. There are some existing tools that are able to present this information, such as ExoneraTor[1] or Onionite[2], but they provide either brief historical information or detailed information about the current state. This thesis extends a tool called Consensus Parser that offers detailed historical information. Consensus Parser is developed as a part of TARZAN research project at Faculty of Information Technology at Brno University of Technology. However, it uses regular files to store data, not a database, which is a disadvantage. Because the information about the Tor network is continuously updated, it has multiple versions in time and its volume is large and will grow.

A traditional relational database is not a good choice for storing large amounts of historical data, but there is a technology called Online Analytical Processing (OLAP) that is well suited to store such data and provide a multidimensional view of it. The aim of this thesis is to modify Consensus Parser to use an OLAP database to store the information about the Tor network and extend its searching and filtering capabilities.

In Chapter 2, principles of onion routing and overview of Tor service are introduced along with the description of the Tor directory protocol and the existing tools for retrieving information about the Tor network. Chapter 3 presents OLAP and compares it to Online Transaction Processing (OLTP). Then, it describes multidimensional data model and column-store database principles. Next, it compares PostgreSQL and MariaDB support for OLAP and it closes with MariaDB ColumnStore architecture description. In Chapter 4, design of the data schema for storing data about the Tor network is introduced, and the fact table and the dimension tables are presented. Chapter 5 describes the integration of MariaDB with Consensus Parser and extensions to the API. In Chapter 6, results of

[1]https://metrics.torproject.org/exonerator.html
[2]https://onionite.now.sh

the experiments comparing the original version of Consensus Parser to the new one are presented.

Chapter 2

Tor

This chapter begins with the overview of Tor followed by the description of Tor's design. Next, the Tor directory protocol is presented followed by the existing tools to query information about Tor. The chapter closes with the description of Consensus Parser. The information provided in this chapter is not meant to be a detailed description of Tor, it is more like a summary of the principles and parts of the service. This chapter should help the reader understand what is the context of the data that is stored in the system described by this thesis.

2.1 Overview

Tor is a service providing anonymity on the Internet by bouncing the client's communications around a distributed overlay network of relays that are provided voluntarily by people and organizations around the world [26]. This network is called the Tor network and the relays are also called onion routers. Tor was originally designed and implemented by the U.S. Naval Research Laboratory[1] to protect government communications [29]. Today it is maintained and developed by a non-profit organization The Tor Project[2] [26]. Users of Tor are regular people, journalists, activists, law enforcement officers, the military, IT professionals and many others. They use Tor to ensure privacy, prevent identity theft, anonymously report abuse, express opinions without fear of losing their job, bypass the Internet censorship, and more [29]. However, it is not always used for legitimate reasons as criminals also make use of it to prevent being caught. The Tor Project argues that criminals have many other means of staying anonymous while normal users do not [24]. Tor is most commonly used with the Tor Browser[3] that is based on Mozilla Firefox[4] and allows users to browse the web anonymously without installing anything else. „Tor“ may also refer to the client program or „onion proxy“ running on the user's device. Both onion routers and onion proxies are also referred to as „nodes“.

[1] https://www.nrl.navy.mil

[2] https://www.torproject.org

[3] https://www.torproject.org/projects/torbrowser.html.en

[4] https://www.mozilla.org/en-US/firefox/

2.2 Design and functionality

The basic principle of Tor is encrypting user's data with three different encryption keys and then relaying the encrypted data via three relays in the Tor network. Encryption layering can be seen in Figure 2.1. User's client knows all of the keys while each relay knows only one of the keys. This ensures that no relay can tell the user is connecting to the destination. Although the first relay is connected directly to the user's client, it only knows that the user is using Tor. The second relay knows nothing about the user or the destination, it only knows that the first relay is relaying traffic to the third relay via itself. The third relay knows the destination, and possibly the payload data, if it is not encrypted. If the user does not want the third relay to see the data, he has to use additional end-to-end encryption such as TLS[5]. It is also necessary to not use any identification credentials if the user wants to be anonymous to the destination.

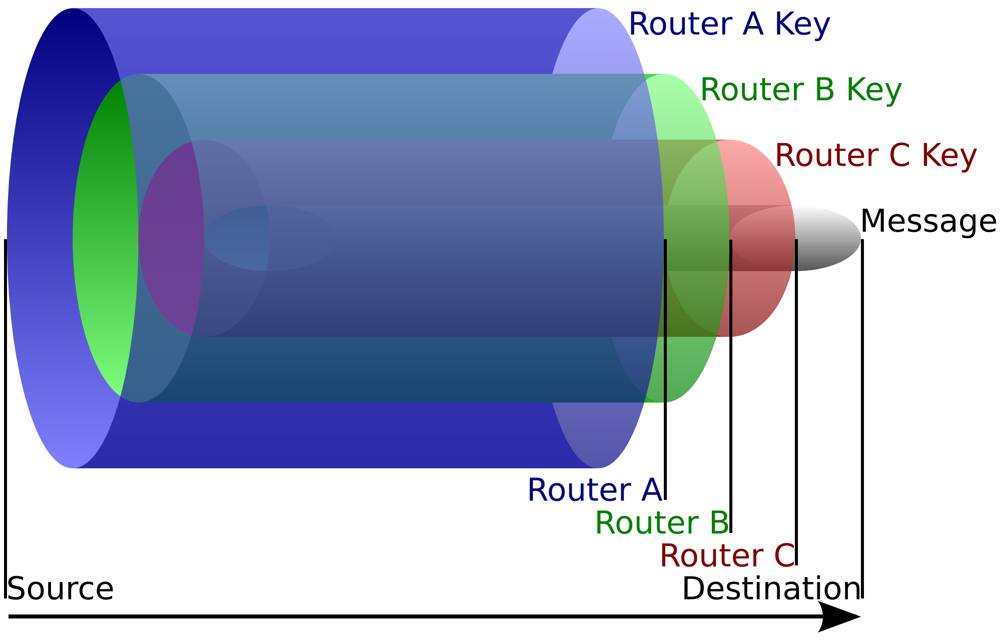


Figure 2.1: Principle of encryption in onion routing. The onion proxy exchanges a key with each onion router in the circuit using Diffie-Hellman method. Each onion router can then decrypt one layer and relay the message forward. Reproduced from [17].

Cell

Cell is a fixed-size unit of communication which onion routers and onion proxies use to communicate with each other. The size of the cell is 512 bytes and each cell contains a circuit identifier, a command, and a payload [6]. Cells are divided to control and relay cells. Control cells are interpreted by the receiving node while relay cells are relayed to the next node. Relay cells have additional relay header at the beginning of the payload. All

[5]TLS – Transport Layer Security, more at https://tools.ietf.org/html/rfc8446

cells are passed via TLS connections with ephemeral keys, thus providing perfect forward secrecy.

Circuit

Circuit is a path (see Figure 2.2) in the Tor network from the onion proxy to the destination server via three relays where all the keys between the onion proxy and the relays are established. It is identified by a circuit identifier contained in a header of each cell. This identifier is different on each connection between two nodes. For example, in the connection between the onion proxy and the first relay, the circuit identifier is *c1* and in the connection between the first and second relay it is *c2*. Both *c1* and *c2* refer to the same circuit, but the onion proxy does not know *c2* and the second relay does not know *c1*. Multiple TCP streams can be attached to one circuit and multiple circuits can be multiplexed over one TLS connection [6].

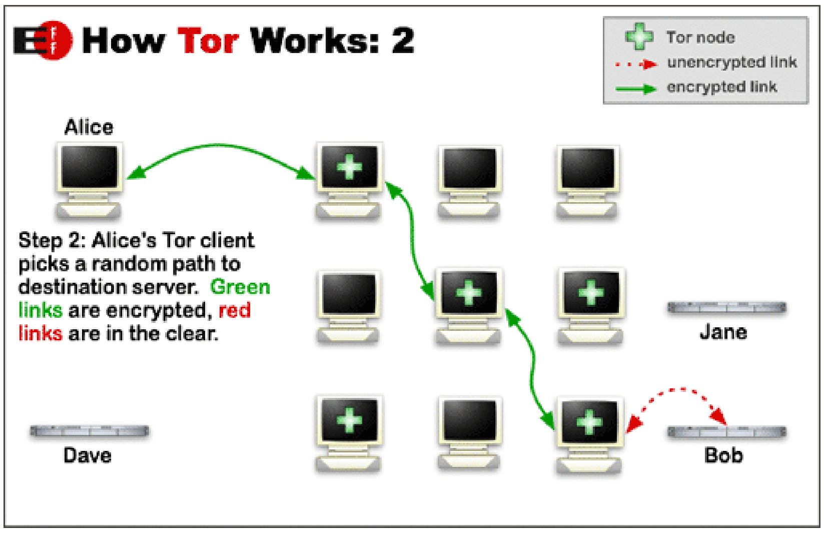

How Tor Works: 2
Tor node
unencrypted link
encrypted link
Alice
Step 2: Alice's Tor client picks a random path to destination server. Green links are encrypted, red links are in the clear.
Jane
Dave
Bob

Figure 2.2: Path from Tor user (Alice) to the destination (Bob). When all the keys between Alice's onion proxy and onion routers are established, path becomes a circuit. Reproduced from [27].

Onion router

Onion router is a relay in the Tor network and its responsibility is to relay traffic. It can be run by anyone with reasonably fast and reliable Internet connection as it is just a normal user-level process. There are three kinds of relays according to the position in the circuit:

Guard relay – first relay in the circuit, communicates with the onion proxy and the middle relay.

Middle relay – second relay in the circuit, communicates with the guard relay and the exit relay.

Exit relay – third and last relay in the circuit, communicates with the middle relay and the target server.

Since Tor uses a leaky pipe circuit topology, it is not always the exit relay that communicates with the target server, but it can be the guard relay or the middle relay. Therefore, different streams can exit via different onion routers in the same circuit. This is helpful when the user does not want to use the exit relay because of its exit policy and it also hides the fact that the streams belong to the same user [6].

There is also a special kind of relay called **Bridge relay** or **bridge** which is not listed publicly, thus allowing use of Tor to users whose ISP blocks known Tor relays [25].

Every onion router has multiple pairs of private and public keys [5]:

- Long-term 1024-bit RSA identity key for signing TLS certificates and server descriptors.
- Short-term onion key for decrypting user requests to build a circuit and for negotiating ephemeral keys. Older versions use 1024-bit RSA key, the latest version uses Curve25519 key but accepts the RSA key too.
- Short-term Ed25519 link authentication key for authenticating TLS handshake. In older versions this was called connection key and was a 1024-bit RSA key. Connection key handshake is still supported in the latest version.

Onion proxy

Onion proxy is a Tor client software that fetches directory protocol documents (see Section 2.3), handles connections from user applications and establishes a circuit in the network. When a circuit is established, data from user applications is sent encrypted with three symmetric keys negotiated with the three relays in the circuit. User applications ask the onion proxy to make a connection via SOCKS[6] protocol [6]. The onion proxy then chooses an open circuit or creates a new one and opens a stream on this circuit. After opening a stream, the onion proxy accepts application data, encapsulates it in cells and sends it along the circuit. Using SOCKS protocol has a downside because some applications try to resolve a hostname and then pass an IP address to the onion proxy. By resolving a hostname, the application reveals the destination server to the DNS server. The solution is to pass a hostname to the onion proxy, so it can be resolved by one of the relays. But because this is not easy to force in all applications, users have to be careful and check this behavior.

2.3 Directory protocol

For the Tor network to function, onion proxies have to know about the running onion routers so they can construct circuits. This is handled by the directory protocol, which defines how the information about the network is distributed among the nodes. This information is publicly available and contains a list of active onion routers, their addresses and ports, nicknames, public keys, exit policies and more. Directory protocol is transferred over the network via HTTP[7] protocol. There are four types of documents defined in the directory protocol:

- **Server descriptor** – document describing the onion router's keys, capabilities and other information.

[6]SOCKS – Socket Secure, more at https://tools.ietf.org/html/rfc1928

[7]HTTP – Hypertext Transfer Protocol, more at https://tools.ietf.org/html/rfc2068

- **Microdescriptor** – stripped down version of a server descriptor containing only the most relevant parts.
- **Status vote** – document containing a summary of current descriptors and status for onion routers. It is generated by a single directory authority.
- **Consensus status** – document that is computed by the directory authorities from multiple status votes. It represents the current state of the network.

There are three important times that are defined for each consensus:

- **Valid-after** – all directory authorities have computed the consensus, which is multiply signed by them. Valid-after time precedes the fresh-until time.
- **Fresh-until** – at this moment, a newer consensus becomes valid. Therefore, the older one is not the freshest one, but still valid. Fresh-until time precedes the valid-until time.
- **Valid-until** – at this point in time the consensus becomes invalid.

Typically, the fresh-until time is one hour after the valid-after time and the valid-until time is two hours after the fresh-until time.

Consensus status

Consensus status document contains the preamble, the authority section and router status entries. In the preamble there is information about the document itself, such as directory protocol version, whether the document is a consensus or a vote, valid-after, fresh-until, valid-until timestamps, recommended client versions, parameters and more [4]. All timestamps are in *YYYY-MM-DD HH:MM:SS* format in UTC time zone. The authority section has information about the authorities that contributed to the document. This includes [4]:

- **Nickname** – a convenient identifier for the authority.
- **Identity fingerprint** – an uppercase hexadecimal SHA-1[8] hash of the authority's current identity key.
- **Hostname** – DNS name of the authority.
- **IP address** – current IP address of the authority.
- **Directory port** – the port that the authority listens on for directory protocol connections.
- **Onion router port** – the port that the authority listens on for onion routing connections.

Router status entries in consensus status document describe onion routers currently known within the network. Each router entry has this information [4]:

- **Nickname** – onion router's nickname, a string of alphanumeric characters of length 1–19 that identifies an onion router.

[8]SHA-1 – Secure Hash Algorithm 1, more at https://tools.ietf.org/html/rfc3174

- **Identity fingerprint** – a SHA-1 hash of the onion router's identity key, encoded in Base64[9].
- **Digest** – a SHA-1 hash of the onion router's most recent descriptor (server descriptor), encoded in Base64.
- **Publication time** – a publication time of the onion router's most recent descriptor.
- **IP address** – the current IP address of the onion router.
- **Onion router port** – the port that the onion router listens on for onion routing connections.
- **Directory port** – the port that the onion router listens on for directory protocol connections.
- **Status flags** – a series of space-separated status flags such as „Running" if the router is currently usable over all its published onion router ports, „Exit" if the onion router is an exit relay, etc. Status flags are in lexical order.
- **Version** – the version of the Tor protocol that the onion router is running.
- **Proto entries** – list of Tor protocols (directory protocol, server descriptor, etc.) with versions that the onion router supports.
- **Bandwidth** – an estimate of the bandwidth of the onion router, in kilobytes per second.
- **Policy** – list of patterns that specify ports which the onion router accepts or rejects traffic on.

```
r seele AAoQ1DAR6kkoo19hBAX5K0QztNw QUykpnS54fRnXu0OyHP9sWIbMGI 2019-01-10
   14:09:32 67.174.243.193 9001~0
s Running Stable V2Dir Valid
v Tor 0.3.5.7
pr Cons=1-2 Desc=1-2 DirCache=1-2 HSDir=1-2 HSIntro=3-4 HSRend=1-2 Link=1-5
    LinkAuth=1,3 Microdesc=1-2 Relay=1-2
w Bandwidth=18
p reject 1-65535
```

Figure 2.3: Example router status entry from consensus status document.

Directory authority

The directory authority is an HTTP server that collects and provides information about the Tor network. Onion routers periodically send their server descriptors to every directory authority they know. The authority provides server descriptors indexed by the router identity or by the hash of the descriptor [4]. The directory authority periodically generates

[9] https://tools.ietf.org/html/rfc4648

a status vote with the current descriptors and sends it to other authorities. All authorities then compute the consensus status, which is used by onion proxies, directory authorities and directory caches (see Subsection *Directory cache*) to detect that their list of server descriptors is out of date. Onion proxies download the descriptors from directory caches and directory caches and directory authorities download them from directory authorities.

Directory cache

Directory cache is an onion router that downloads, caches, and provides consensus documents to reduce load on the directory authorities [4]. Onion proxies prefer directory caches over directory authorities when downloading server descriptors. Directory cache also serves microdescriptors, because some onion proxies download only those. Directory cache always downloads consensus document if it does not have one or its current consensus is not valid any more. If the directory cache has a valid consensus, it will try to download a new one after the current one is no longer fresh.

2.4 Existing tools for Tor

There are tools that enable viewing and searching for information about the Tor network, but they provide either detailed information about the current state or brief information about the history. This thesis aims to create a tool that combines both features, i.e. enable querying for detailed information in any point in time.

ExoneraTor

ExoneraTor is a tool that provides a way to determine if a relay with a given IP address was active in the Tor network on a given day. IP address can be either IPv4 or IPv6 address. Search results contain not only information from the given day, but also the day before and the day after it. This is the only tool that enables to examine the history of the network, but it only provides information about the timestamp, IP address, identity fingerprint, nickname and flag determining if the router is an exit relay.

Onionite application and Onionoo API

Information in this subsection is taken from [3] and [28]. Onionite is a web application that shows current information about nodes in the Tor network. It allows the user to search for a node by its nickname, IP address or identity fingerprint. It is possible to enter only a part of the search term and Onionite will display all corresponding results. Onionite uses the Onionoo protocol[10] for acquiring the information. Onionoo is a RESTful[11] API that accepts HTTP GET requests and returns responses in JSON[12] format. Onionoo provides these documents:

- **Summary** – short summary of relays with nicknames, fingerprints and IP addresses.
- **Details** – similar information as in the consensus status document.

[10] https://metrics.torproject.org/onionoo.html

[11] REST – Representational state transfer, more at https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[12] JSON – JavaScript Object Notation, more at https://tools.ietf.org/html/rfc7159

```
147/
  83/
    147.83.29.168
    147.83.32.100
2001/
  ac8/
    2001:ac8:21:2::190
    2001:ac8:ab9:7::21
```

Figure 2.4: Example directory structure for IP addresses 147.83.29.168, 147.83.32.100, 2001:ac8:21:2::190 and 2001:ac8:ab9:7::21.

- **Bandwidth** – statistics of a consumed bandwidth for different time intervals.
- **Weights** – statistics for a different time intervals of a relay's probability to be selected for building a path.
- **Clients** – estimate of an average number of users connecting to the bridge per day.
- **Uptime** – fractional uptimes for different time intervals.

Requests to Onionoo API may contain parameters in the query component of the URL to filter the results. It is possible to filter by relay type, countries, autonomous systems, flags and many more. Onionite also supports these parameters in the search in the form *parameter-name:value*.

2.5 Consensus Parser

Consensus Parser is a tool for acquiring and processing consensus status documents and providing information from these documents via Toreator[13] REST API or by the tool itself. This tool is a part of TARZAN[14] research project which is developed at Faculty of Information Technology at Brno University of Technology. Consensus Parser is implemented in Python 3.6 programming language with Flask[15] framework used for the REST API part. Information from consensus status documents is complemented by geolocation data from GeoLite2 database created by MaxMind[16] and reverse DNS records.

Consensus Parser stores the data from consensus status documents on the file system in the form of a preprocessed consensus file for each IP address. These files use similar format as router status entries from consensus status documents (see Figure 2.3) but contain only information about the specific IP address. Each preprocessed consensus file is named exactly as the IP address it contains information about. They are stored hierarchically in directories by the first two octets (for IPv4) or first two hextets (for IPv6). For example, if there is information about IP addresses 147.83.29.168, 147.83.32.100, 2001:ac8:21:2::190 and 2001:ac8:ab9:7::21, the directory structure will look like the one in Figure 2.4.

[13]http://toreator.fit.vutbr.cz/
[14]http://www.fit.vutbr.cz/units/UIFS/grants/index.php.en?id=1063
[15]http://flask.pocoo.org/
[16]https://www.maxmind.com

```
x-inconsensus-valid-after 2019-01-10 19:00:00
x-inconsensus-fresh-until 2019-01-10 20:00:00
x-inconsensus-valid-until 2019-01-10 22:00:00
```

Figure 2.5: Lines with valid-after, fresh-until and valid-until times in a preprocessed file.

```
x-dns-reverse tor1.relay.cstone.com 2019-01-10 21:08:36
```

Figure 2.6: Reverse DNS record in a preprocessed file.

There are some additional lines in preprocessed consensus files in addition to router status entries in consensus status documents. Specifically, there are always lines with valid-after, fresh-until and valid-until times for each entry in a preprocessed file as can be seen in Figure 2.5.

Optionally there might also be a reverse DNS record if Consensus Parser was executed with the `--additional_info` or `-a` argument. An example of a reverse DNS record in a preprocessed file is displayed in Figure 2.6.

Geolocation data is not stored in preprocessed consensus files but in CSV files in a directory specified by the `--geolite_dir` or `-g` argument. Geolocation data is added only when the IP address is looked up either by Toreator API or by Consensus Parser executed with the `--ipaddress` or `-i` argument.

Command line arguments

Consensus Parser has two main functionalities – the first one is parsing and storing consensus status documents and the second one is searching the data that is already stored by IP addresses. Therefore, command line arguments are divided into two groups corresponding to the functionalities as can be seen in Table 2.1 and Table 2.2 (Note: in tables 2.1, 2.2 and in the list of endpoints, date formats are specified according to `strftime`[17] function in Python).

Long form	**Takes value**	**Description**
`--consensus_path`	Yes	The input file or directory with consensuses.
`--update`	No	Fetch and parse new consensuses.
`--update_keeprunning`	No	Keep running and fetch and update new consensuses.
`--write_preprocessed`	Yes	The path where preprocessed consensus will be written.
`--additional_info`	No	For each IP address gather additional info like reverse DNS.
`--email`	Yes	E-mail address to be notified when a consensus is not available on time.

Table 2.1: Arguments related to parsing and storing.

[17] https://docs.python.org/3/library/time.html#time.strftime

Long form	Takes value	Description
`--ipaddress`	Yes	The IP address to be searched for.
`--preprocessed_input`	Yes	The preprocessed directory with consensuses sorted by IP address.
`--geolite_dir`	Yes	The directory with geolite CSVs. The CSVs are expected to be in subdirectories contained in the downloadable zip files.
`--time`	Yes	Time of the search for the IP address (UTC).
`--date_prefix`	Yes	Specify the date prefix for the search – prefix of *%Y-%m-%d*.

Table 2.2: Arguments related to searching by IP address.

Toreator API

Toreator is a REST API that provides information about relays acquired from consensus status documents. Since it is a module of Consensus Parser, it can directly use all the functionality that Consensus Parser provides by calling its Python methods. It can return results in HTML or JSON format. In HTML response, list of addresses consists of HTML links that lead to other URLs[18] of the API. In JSON response there is a list with two elements for each IP address in a list. First element is the address itself and the second is the relative URL it leads to. Details about a relay contain the same information in both HTML and JSON. Example of an HTML result can be seen in Figure 2.7. Provided endpoints are the following (`<address>` is an IPv4 or IPv6 address. `<time>` is in *%Y-%m-%d %H:%M:%S* format. `<date>` is in *%Y-%m-%d* format. `<month>` is in *%Y-%m* format. `<year>` is in *%Y* format.):

`/`

Link to the `/addresses/` endpoint.

`/addresses/`

List of CIDR IP addresses of all /8 IPv4 and /16 IPv6 networks that contain addresses of relays.

`/addresses/<address>-<mask>/`

List of CIDR IP addresses of all networks or hosts specified by the given IP address and mask. `<mask>` is a digit in range 0-32.

`/addresses/<address>/`

Details from all consensuses about the relay with the given IP address. See Figure 2.7 for example of details about a relay from one consensus.

`/addresses/<address>/date/`

List of all dates which the relay with the given IP address was active on.

`/addresses/<address>/date/<date>/`

Details about the relay with the given IP address from all consensuses from the given date.

[18]URL – Uniform Resource Locator, more at https://tools.ietf.org/html/rfc3986

`/addresses/<address>/time/`

Redirects to `/addresses/<address>/date/`

`/addresses/<address>/time/<time>/`

Details about the relay with the given IP address from all consensuses that were valid at the given time.

`/addresses/<address>/month/`

List of all months which the relay with the given IP address was active in.

`/addresses/<address>/month/<month>/`

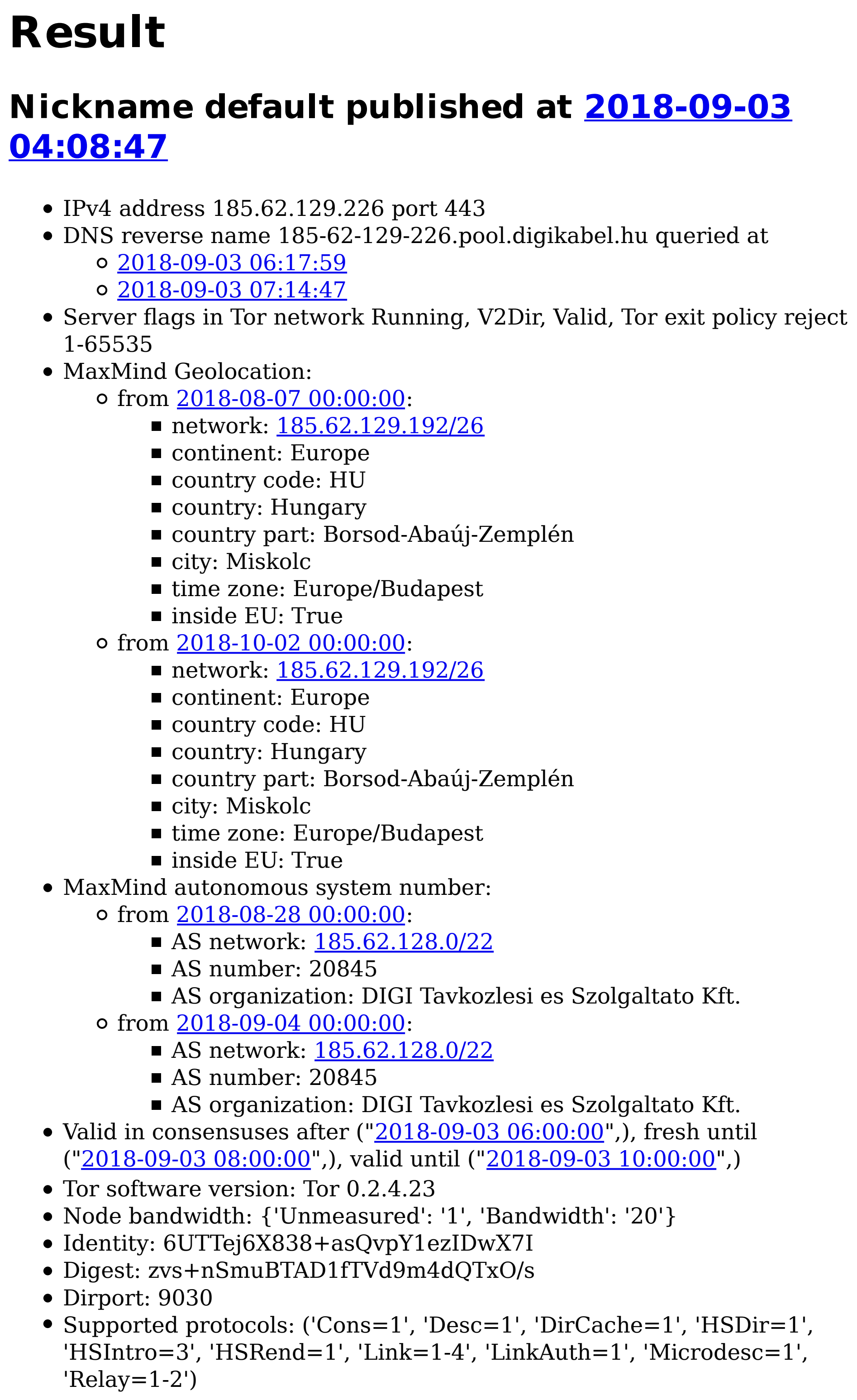
Details about the relay with the given IP address from all consensuses from the given month.

`/addresses/<address>/year/`

List of all years which the relay with the given IP address was active in.

`/addresses/<address>/year/<year>/`

Details about the relay with the given IP address from all consensuses from the given year.



Result

Nickname default published at 2018-09-03 04:08:47

- IPv4 address 185.62.129.226 port 443
- DNS reverse name 185-62-129-226.pool.digikabel.hu queried at
  - 2018-09-03 06:17:59
  - 2018-09-03 07:14:47
- Server flags in Tor network Running, V2Dir, Valid, Tor exit policy reject 1-65535
- MaxMind Geolocation:
  - from 2018-08-07 00:00:00:
    - network: 185.62.129.192/26
    - continent: Europe
    - country code: HU
    - country: Hungary
    - country part: Borsod-Abaúj-Zemplén
    - city: Miskolc
    - time zone: Europe/Budapest
    - inside EU: True
  - from 2018-10-02 00:00:00:
    - network: 185.62.129.192/26
    - continent: Europe
    - country code: HU
    - country: Hungary
    - country part: Borsod-Abaúj-Zemplén
    - city: Miskolc
    - time zone: Europe/Budapest
    - inside EU: True
- MaxMind autonomous system number:
  - from 2018-08-28 00:00:00:
    - AS network: 185.62.128.0/22
    - AS number: 20845
    - AS organization: DIGI Tavkozlesi es Szolgaltato Kft.
  - from 2018-09-04 00:00:00:
    - AS network: 185.62.128.0/22
    - AS number: 20845
    - AS organization: DIGI Tavkozlesi es Szolgaltato Kft.
- Valid in consensuses after ("2018-09-03 06:00:00",), fresh until ("2018-09-03 08:00:00",), valid until ("2018-09-03 10:00:00",)
- Tor software version: Tor 0.2.4.23
- Node bandwidth: {'Unmeasured': '1', 'Bandwidth': '20'}
- Identity: 6UTTej6X838+asQvpY1ezIDwX7I
- Digest: zvs+nSmuBTAD1fTVd9m4dQTxO/s
- Dirport: 9030
- Supported protocols: ('Cons=1', 'Desc=1', 'DirCache=1', 'HSDir=1', 'HSIntro=3', 'HSRend=1', 'Link=1-4', 'LinkAuth=1', 'Microdesc=1', 'Relay=1-2')

Figure 2.7: Example result from Toreator REST API with reverse DNS and geolocation data. The result can be obtained from this endpoint: `/addresses/<address>/`, e.g., `/addresses/185.62.129.226/`.

Chapter 3

Online Analytical Processing

In the beginning of this chapter, Online Analytical Processing (OLAP) is defined and compared to Online Transaction Processing (OLTP). Then, multidimensional model and OLAP operations are described. Next, column-store or column-oriented database is presented and explained. This is followed by a comparison of OLAP support in PostgreSQL and MariaDB. After that, MariaDB ColumnStore architecture is described and the chapter closes with a short summary.

3.1 What is OLAP

The definition of OLAP by the OLAP Council[1] is the following [18]:

> *On-Line Analytical Processing (OLAP) is a category of software technology that enables analysts, managers and executives to gain insight into data through fast, consistent, interactive access in a wide variety of possible views of information that has been transformed from raw data to reflect the real dimensionality of the enterprise as understood by the user.*

As the definition states, OLAP is used by analysts and managers to see the data from multiple views. It is also often used in data warehousing because it is concerned with historical and summarized data that provide better means for decision making. These are the two main types of OLAP [7]:

ROLAP – relational OLAP. Data is stored in a relational Database Management System (DBMS). The multidimensional model is mapped to the relational model using a special type of schema. It is more scalable than MOLAP and allows to use SQL, but the performance is worse than with MOLAP. Another advantage is the amount of literature about the relational model and the experience with relational database usage and management. This is the type chosen for this thesis because of the ability to use an open-source relational DBMS and SQL.

MOLAP – multidimensional OLAP. It is based on an ad hoc logical model that can represent multidimensional data directly. There are mostly proprietary solutions, for example Microsoft Analysis Services. It has better performance than ROLAP but poor storage utilisation.

[1] www.olapcouncil.org/

There are also other types such as Hybrid OLAP, Desktop OLAP and Web-based OLAP, but those are not relevant to this thesis.

Comparison to OLTP

OLTP is an operational system that processes tasks such as order entry, retail sales and financial transactions [2]. Data is detailed and up-to-date and the main goals are consistency, reliability and transaction throughput [2]. The size of an OLTP database is much smaller than the size of an OLAP database because OLTP is not concerned with historical data. Operations in OLTP include querying, inserting, updating and deleting. Queries in OLTP system are usually simple.

In comparison, OLAP is targeted on providing data for complex analyses, so queries are more complex than in OLTP [2]. Data is not so detailed but the volume is much larger since there is historical data which might even come from different sources (multiple OLTP systems, flat files) [2].

3.2 Multidimensional data model

Multidimensional data model is the core of OLAP because it allows to view data in multiple dimensions. It is often visualized as a data cube that might have an arbitrary number of dimensions. If the number of dimensions is more than three, it is called *hypercube* [20]. Two most important terms in a multidimensional model are:

Fact – a factor affecting the decision-making process. Instance of a fact is an event and each fact is described by the values of measures that provide quantitative description of events [7]. For example, shipment is a fact, a single carried out shipment is an event and the amount shipped is a measure [7]. Within the context of this thesis, router status entry is a fact, a specific router status entry in a consensus file is an event and the bandwidth is a measure.

Dimension – an axis of n-dimensional space that events are placed in. Dimensions define different perspectives to single out events [7]. A shipment may have products, dates and destinations dimensions. In a data cube, dimensions are the edges of the cube while events are the cube cells [7]. For router status entry there are published dates, IP addresses or nicknames, among other dimensions.

Typical operations to reduce the quantity of data and obtain useful information in multidimensional data model are [7]:

- **Roll-up** – increasing the aggregation level of a dimension and therefore getting more summarized data. For example, if there is a time dimension with aggregation level of months, by doing roll-up it is changed to years.
- **Drill-down** – decreasing the aggregation level of a dimension and thus getting more detailed data. It is a complementary operation to roll-up.
- **Slice** – decreasing cube dimensionality by setting a specific dimension to only one value. Effectively eliminates the dimension from the model. If all dimensions are set to a particular value, they define a single event.

- **Dice** – generalization of slicing, it limits dimensions to a subset of their possible values, making the cube smaller.

In ROLAP, facts are stored in a fact table which contains measures and foreign keys to dimension tables. When the fact table does not contain any measures, only foreign keys, it is called a *factless* table [20]. Dimensions are stored in dimension tables that consist of a primary key and attributes describing the dimension. The attributes in a dimension table may form hierarchies [20]. Since relational model does not include concepts of dimension, measure and hierarchy, multidimensional model must be represented by specific types of schemas [20]:

- **Star schema** – a schema with a single fact table which is a virtual center of the schema and multiple dimension tables that surround the fact table. Dimension tables in a star schema are denormalized (contain redundant information) to prevent many join operations and thus improving performance. Star schema is named so because when it is visualized, it resembles a star. An example of a star schema can be seen in Figure 3.1.

- **Snowflake schema** – a schema based on a star schema but with normalized dimension tables. It is called snowflake schema because normalizing dimension tables results in more tables and the schema visualisation then looks like a snowflake.

- **Fact constellation** – a schema where there are multiple fact tables sharing the dimension tables [2].

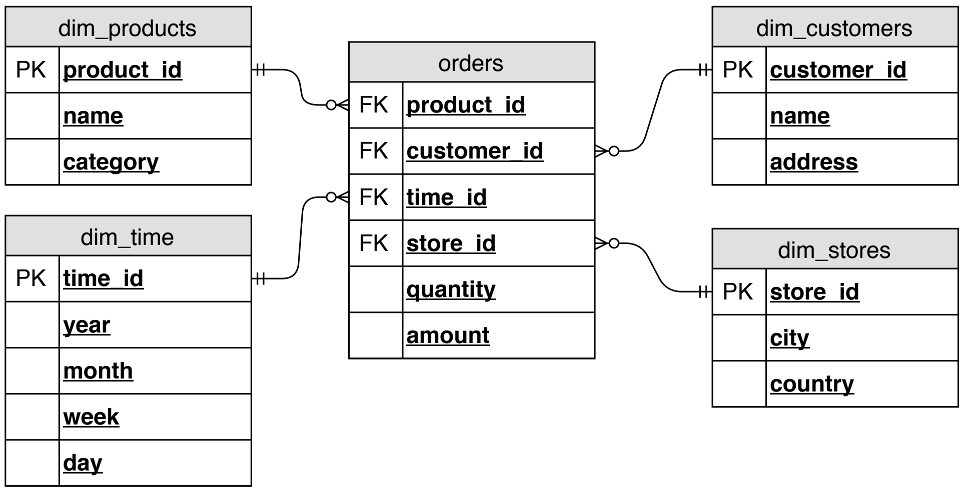
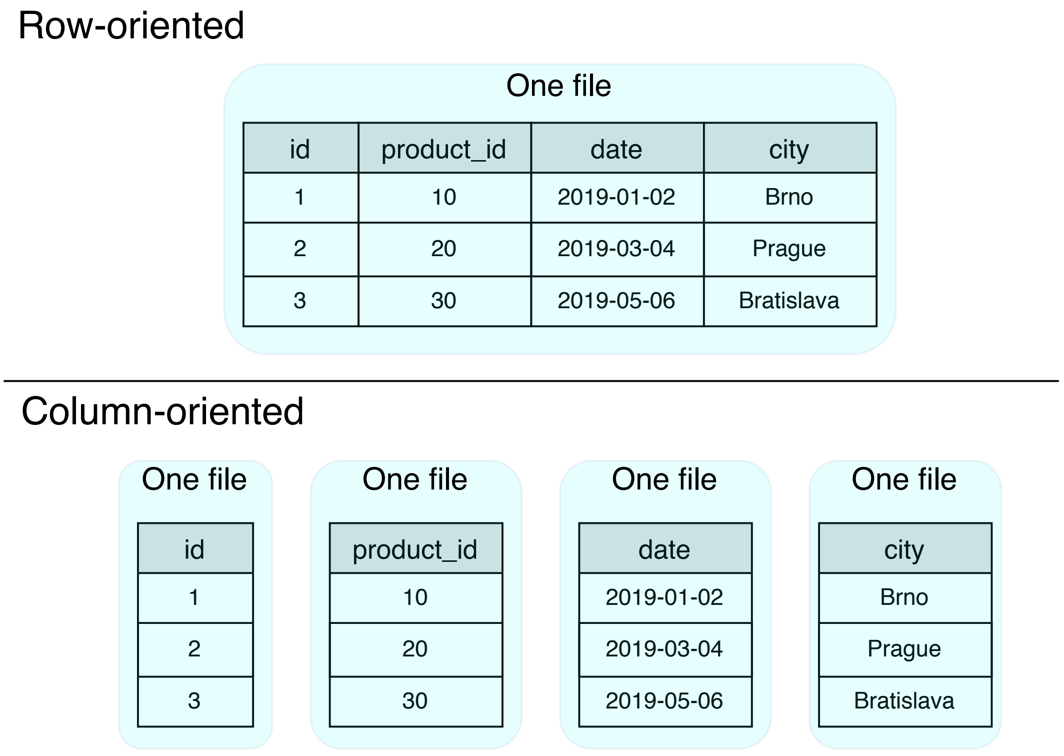


Figure 3.1: A star schema example. `orders` table is a fact table that contains measures (`quantity` and `amount`) and foreign keys pointing to the dimension tables (`dim_products`, `dim_customers`, `dim_time` and `dim_stores`). Dimension tables in a star schema contain redundant data (`country` will be the same for multiple `city` records) on purpose because it improves query performance.

3.3 Column-store

Information in this section is taken from [1]. With analytical queries it is typical to access very large amounts of data on disk. Because reading this much data is often slow, it is desirable to reduce the amount of accessed data as much as possible to speed up the execution of the query. In traditional row-oriented DBMS, data for a single table is stored together in one or multiple files on disk. This means that if a query is only concerned with one column, other columns still have to be read and then discarded. For transactional workloads this is not a problem because the user is often interested in many or all columns and the size of accessed tables is not that big. But with the size of the data that analytical queries usually access, this introduces a significant performance penalty. This is where the column-oriented DBMS or *column-store* comes into play.

In column-store, each column is stored in a separate file on disk (see Figure 3.2 for difference between row-oriented and column-oriented physical layout). This allows the system to read only the columns that the query needs rather than reading entire rows and discarding the data after. In consequence, memory bandwidth and overall utilisation of the available I/O improve because less data is transferred between main memory and CPU registers. And, most importantly, this leads to reduced query execution time. The wider the table, i.e. the more columns the table has, the more time and resources are saved.



id	product_id	date	city
1	10	2019-01-02	Brno
2	20	2019-03-04	Prague
3	30	2019-05-06	Bratislava

id
1
2
3

product_id
10
20
30

date
2019-01-02
2019-03-04
2019-05-06

city
Brno
Prague
Bratislava

Figure 3.2: Illustration of the difference between the physical layout of row-oriented and column-oriented DBMS. Row-oriented DBMS stores the data for the whole table in one file on disk while column-oriented DBMS stores each column in a separate file. This allows column-oriented DBMS to only access the columns that are needed for the query, saving time and resources.

The next advantage of column-store is compression of the data which causes substantial increase in performance and available disk space. Compression rates are good because the data in a specific column has low information entropy and therefore is easier to compress than the data in a whole table which has high information entropy. And since CPUs are generally faster than memory bandwidth, it is preferable to spend CPU cycles on decompressing the data rather than transferring more data from disk to memory and from memory to CPU. In many cases it is even possible to operate directly on compressed data, e.g. with run-length encoding it is sufficient to multiply run-lengths and values when doing a sum operation instead of summing each value independently. Another performance gain is achieved by leveraging vectorized processing which uses vectors of N tuples that fit into the L1 cache instead of one tuple at a time.

3.4 OLAP support in MariaDB and PostgreSQL

PostgreSQL[2] is a very popular open source object-relational database system released under the PostgreSQL license[3]. Its advantages are a large and active community of users, extensive and detailed documentation, robust feature set, reliability, extensibility and support for all major operating systems [21]. It has been developed for over 30 years [21].

As for the OLAP support, because of PostgreSQL's almost full conformance to ISO/IEC 9075 "Database Language SQL", it supports GROUPING SETS, ROLLUP and CUBE clauses since version 9.5 [23, 22]. There is also a module that implements a data type for representing a multidimensional cube called *cube*[4]. However, these are just syntax constructs that help the user write more concise queries. PostgreSQL is not really an analytical database and its focus is on transactional workloads which it does very well. For analytical queries, however, it cannot compete with column-oriented DBMS, as can be seen in this benchmark[5]. There is an extension called *cstore_fdw*[6] that implements column-store for PostgreSQL. This improves performance for analytical queries but it still has serious disadvantages compared to native column-store systems. Maybe the biggest disadvantage is the use of the normal PostgreSQL query optimizer that is tuned for row-oriented storage. Other drawbacks are that update and delete operations are not supported.

MariaDB[7] is an open source fork of a really popular and widely used relational database system called MySQL[8]. It is developed by MariaDB Foundation[9] which employs some of the developers that founded and worked on MySQL and it still maintains a very high compatibility with MySQL [15]. MariaDB is fast and scalable, has a rich feature set and a strong focus on security [8]. It is released under the GNU General Public License, version 2[10] [14]. OLAP support in MariaDB comes with ColumnStore, the column-oriented storage engine that is designed for big data scaling and performance with real-time response to analytical queries. It is based on InfiniDB[11] and utilizes a parallel distributed data architecture [13]. ColumnStore is the reason MariaDB is better suited for OLAP than PostgreSQL or

[2] https://www.postgresql.org
[3] https://opensource.org/licenses/postgresql
[4] https://www.postgresql.org/docs/current/cube.html
[5] https://tech.marksblogg.com/benchmarks.html
[6] https://github.com/citusdata/cstore_fdw
[7] https://mariadb.com
[8] https://www.mysql.com
[9] https://mariadb.org
[10] https://www.gnu.org/licenses/old-licenses/gpl-2.0.html.en
[11] https://github.com/infinidb/infinidb

MySQL. As mentioned in Section 3.3, ColumnStore allows to access only columns needed for the query, leverages compression and vectorized processing.

3.5 MariaDB ColumnStore architecture

MariaDB ColumnStore is a column-oriented storage engine designed for distributed, massively parallel processing, such as big data analytics. It is composed of three modules – User Module, Performance Module and Storage. These are described below.

User Module

User Module manages and controls the operation of queries. It maintains a state of each query, issues requests to one or more Performance Modules and resolves the query by aggregating results from Performance Modules into one [12]. It contains various processes [12]:

- **MariaDB Server** – normal MariaDB server, runs as *mysqld* process. It is responsible for parsing SQL statements, generating SQL plan and distributing the final result-set. It also converts MariaDB query plan into ColumnStore query plan format which is a parse tree with added execution hints from the optimizer.
- **Execution Manager** – process that converts the parse tree from MariaDB Server into a Job List, i.e. a sequence of instructions that have to be executed to satisfy the query. Job List is comprised of job steps such as application of a column filter, processing table joins or projection of returned columns. Each step can run on User Module only, Performance Module only or combination of both. It listens for query parse trees from MariaDB Server on TCP port 8601. To determine which Performance Module to send work orders to, Extent Map is used (see Subsection *Storage* in this section).
- **Distribution Managers** – DDLProc for distributing Data Definition Language (DDL) statements to Performance Modules; DMLProc for distributing Data Manipulation Language (DML) statements to Performance Modules; cpimport for distributing source files for high-speed bulk loading to Performance Modules.

Performance Module

Performance Module stores, retrieves and manages data, processes block requests and passes them back to the User Module for finalisation [10]. It selects data from disk and caches it in a shared-nothing, LRU-based buffer [10]. There can be many Performance Modules; each one increases the size of the cache and the amount of processing power of the database. A heartbeat mechanism ensures all Performance Modules are running and if one fails, there is a transparent failover [10]. Performance Module is composed of these processes [10]:

- **Process Manager** – starts, monitors and restarts all ColumnStore processes on the Performance Module.
- **Process Monitor** – is used by Process Manager to monitor the processes of ColumnStore.
- **Primary Process** – handles query execution by executing instructions from the User Module as block oriented I/O operations. Performs predicate filtering, join processing and the initial aggregation of data, then sends the data back to the User Module.

- **WriteEngineServer** – coordinates DML, DDL and imports for each Performance Module.
- **cpimport** – updates database files when loading bulk data and thus allows to support fully parallel loads.

Storage

Information in this subsection is taken from [11]. Tables in MariaDB ColumnStore are created with at least one file per column in the table. The file is stored on disk of a Performance Module. Each column is stored in a logical measure of eight million rows called an **Extent**. This makes 8 MB for one byte data types, 16 MB for two byte data types, 32 MB for four byte data types and 64 MB for eight byte or variable size data types. String columns with more than eight bytes store only indexes to separate dictionary files. When an Extent becomes full, a new one is created. An Extent is physically stored as a collection of 8 KB blocks, each uniquely identified by its Logical Block Identifier (LBID). The physical file made up of these blocks is called a segment file. There is a configurable maximum value of extents that one segment file can store, default is two. All segment files of a column form a partition that is stored in a directory on disk. Default value of segment files per partition is four.

MariaDB ColumnStore uses a structure called **Extent Map** to map Extents to their corresponding blocks (LBIDs) and also keep information about minimum and maximum values for the column's data within the Extent. The primary Performance Module has a master copy of the Extent Map and on system startup it is copied to other User and Performance modules for failover. Other modules keep the Extent Map in memory and receive broadcast updates when Extents are modified.

Thanks to the Extent Map, MariaDB ColumnStore can perform logical range partitioning and retrieve only the blocks that are necessary for the query. This is done via eliminating Extents that do not meet the join and filter conditions of the query. Elimination is done by extracting partitioning information and minimum and maximum values of extents and then comparing minimum and maximum values to join and filter conditions. If the values do not meet the conditions, extent is eliminated, i.e. not accessed on disk at all.

MariaDB ColumnStore uses compression for all tables and columns by default. It can be disabled or enabled for each table and column. Since data in each column is similar, compressibility is excellent and compression saves between 65% and 95% of disk space. Compression is optimized for read performance from disk.

3.6 Chapter summary

This thesis employs OLAP technology because it has to deal with large amounts of data and present it in a multidimensional view. The specific type of OLAP that is used is called Relational OLAP since it stores data in a relational database. There were two database systems considered for use, PostgreSQL and MariaDB. Both are well known open source databases with large communities, lots of features and support for multiple operating systems. PostgreSQL, however, is suited mainly for OLTP and has support for OLAP only with an unofficial extension which still does not reach the performance of systems that support OLAP natively. MariaDB, on the other hand, has a storage engine called

ColumnStore which is suited exactly for OLAP and analytical workloads. For this reason, MariaDB is used for this thesis.

Chapter 4

Design of the data schema

This chapter presents the data schema for storing information about onion routers in the Tor network. The first section explains the chosen arrangement, the second section describes the fact table and the third section presents the dimension tables.

4.1 Schema arrangement

Data is arranged in a typical star schema (see Figure 4.1) with some additional tables for GeoLite2 data (see Figure 4.2). There is one fact table which contains keys pointing to the dimension tables, and facts. There are ten dimension tables that describe router status entries in consensus status documents from different perspectives. Full GeoLite2 database is stored in `geolite_blocks` and `geolite_locations` tables. Since `geolite_blocks` is quite large, containing about 3.6 million records, there is a dimension table `dim_geolite` which combines the data from `geolite_blocks` and `geolite_locations`, but only for existing IP addresses in `dim_ip_addresses` dimension table. This results in higher usage of disk space but makes data retrieval faster. One special table that maps IP addresses to `dim_geolite` dimension data is needed for inserting data to the fact table (see Section 5.3).

Some tables that store IP addresses of type `VARBINARY` use MyISAM storage engine instead of ColumnStore because ColumnStore does not support `VARBINARY` data type. If this thesis only worked with IPv4 addresses, they could be stored as `INTEGER` but this is not possible since IPv6 is also required. The default MySQL engine InnoDB also supports VARBINARY but it uses row-level locking while MyISAM uses table-level locking [19]. Table-level locking uses less memory and is faster for read-mostly workloads which this thesis has.

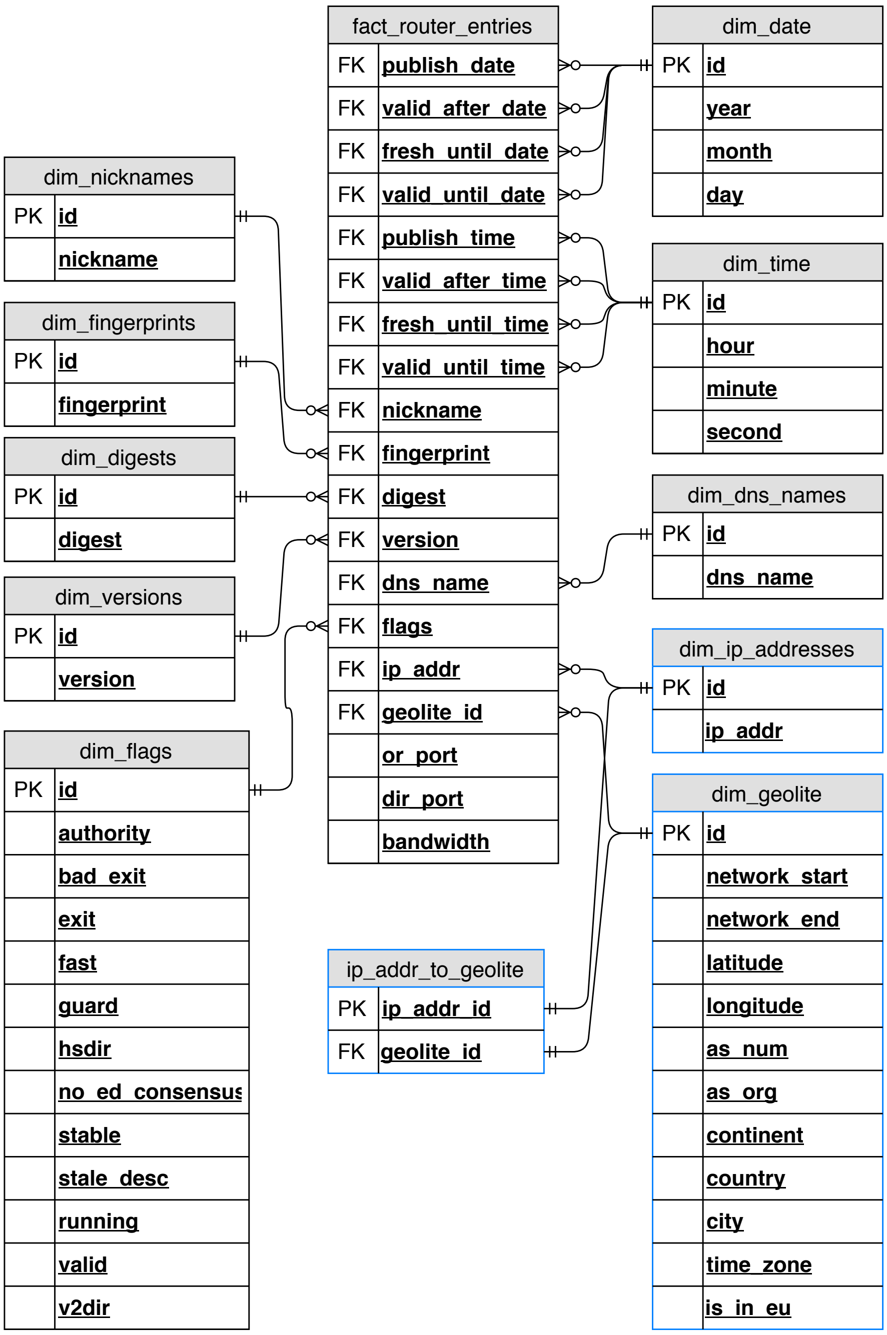


Figure 4.1: Data schema for storing information about onion routers in the Tor network. Tables with black border lines use ColumnStore engine and tables with blue border lines use MyISAM engine. The schema is a star schema with the fact table `fact_router_entries` which contains sixteen keys to the dimension tables and three facts. The keys to the dimension tables are without foreign key constraints typically set in OLTP databases because ColumnStore does not support foreign key constraints.

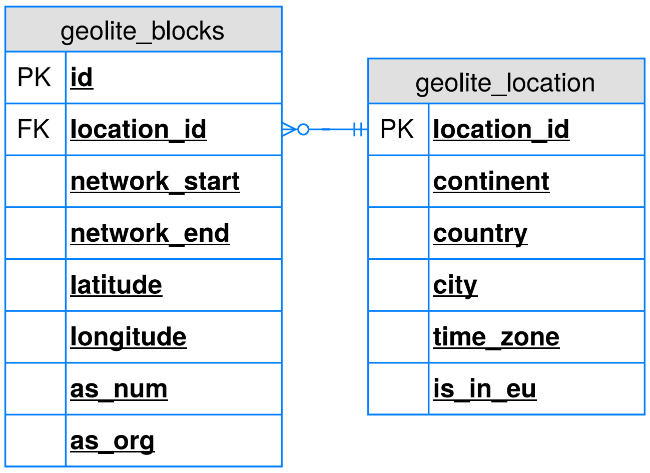


Figure 4.2: GeoLite2 database is stored in two tables and both use MyISAM engine because `geolite_blocks` table columns `network_start` and `network_end` are IP addresses of type `VARBINARY` which is not supported by ColumnStore. `geolite_location` table does not contain any column of type `VARBINARY` but it is always joined only with `geolite_blocks` and joins within the same engine are faster than cross joins between engines.

Data type	Size [B]
`BOOLEAN`	1
`TINYINT`	1
`SMALLINT`	2
`INTEGER`	4
`FLOAT`	4
`BIGINT`	8
`VARCHAR`	N
`VARBINARY`	N

Table 4.1: MariaDB data types with sizes [9, 16]. Data types that are not used in the design were omitted.

4.2 Fact table

The fact table is named `fact_router_entries` and it contains three facts and sixteen foreign keys pointing to the dimension tables. Table columns with corresponding types are shown in Table 4.2. The fact table contains the most columns of all tables in the schema and is supposed to be the largest on disk too. Foreign keys data types (see Table 4.1 for MariaDB data types sizes) correspond to data types of `id` columns of the dimensions. Although the name foreign key is used here, keys are not bound by `FOREIGN KEY` constraints because ColumnStore does not support them. Since there are multiple timestamps in a consensus status, there are multiple foreign keys pointing to `dim_dates` and `dim_times` dimension tables. Foreign keys `dns_name_id` and `geolite_id` may be `NULL` because the IP address of an onion router is not guaranteed to have PTR DNS record or be in GeoLite2 database.

The facts in the fact table are onion router port, directory port and bandwidth (see Subsection *Consensus status* in Section 2.3). Onion router port and directory port have `SMALLINT` data type since port is a two-byte unsigned integer. Bandwidth has type `INTEGER` because it is an estimate of the onion router's bandwidth in kilobytes per second.

fact_router_entries	
`publish_date_id`	`SMALLINT`
`valid_after_date_id`	`SMALLINT`
`fresh_until_date_id`	`SMALLINT`
`valid_until_date_id`	`SMALLINT`
`publish_time_id`	`INTEGER`
`valid_after_time_id`	`INTEGER`
`fresh_until_time_id`	`INTEGER`
`valid_until_time_id`	`INTEGER`
`nickname_id`	`INTEGER`
`fingerprint_id`	`INTEGER`
`digest_id`	`INTEGER`
`version_id`	`SMALLINT`
`dns_name_id`	`INTEGER`
`flags_id`	`SMALLINT`
`ip_addr_id`	`INTEGER`
`geolite_id`	`INTEGER`
`or_port`	`SMALLINT`
`dir_port`	`SMALLINT`
`bandwidth`	`INTEGER`

Table 4.2: `fact_router_entries` table column names and types.

4.3 Dimensions

Dimensions are different perspectives which router status entries in consensus status documents can be seen in. They jointly describe the router status entry. From a storage perspective, dimension tables contain data common for multiple entries in a fact table which only contains a key to each dimension table. Therefore, each dimension table has an `id` column that is referenced by the key in the fact table. `id` column data type differs among dimensions because some tables contain fixed number of elements and do not need a large data type. Every integer type is unsigned since there is no need for negative numbers to be stored. Each dimension table uses ColumnStore storage engine except where noted. These are all dimensions from the schema, each with a description and a table containing dimension table name, its columns and their types:

- **Dates** – holds combinations of years, months and dates. `SMALLINT` type for the `id` column was chosen because it supports years up to 2049, which is sufficient. Years are in range 2007–2049, months in range 1–31 and days in range 1–31. 2007 is the first year consensuses are available for in the archive[1]. This dimension table is referenced from the fact table by `publish_date_id`, `valid_after_date_id`, `fresh_until_date_id` and `valid_until_date_id` columns.

[1] https://metrics.torproject.org/collector/archive/relay-descriptors/consensuses

dim_dates	
id	SMALLINT
year	SMALLINT
month	TINYINT
day	TINYINT

Table 4.3: dim_dates table column names and types.

- **Times** – holds combinations of hours, minutes and seconds. id column is of type INTEGER because there are $24 \times 60 \times 60 = 86400$ combinations which exceeds SMALLINT capacity. This dimension table is referenced from the fact table by publish_time_id, valid_after_time_id, fresh_until_time_id and valid_until_time_id columns.

dim_times	
id	INTEGER
hour	TINYINT
minute	TINYINT
second	TINYINT

Table 4.4: dim_times table column names and types.

- **Nicknames** – nickname is a string of length 1–19 alphanumeric characters that identifies an onion router [4]. This dimension table is referenced from the fact table by nickname_id column.

dim_nicknames	
id	INTEGER
nickname	VARCHAR(19)

Table 4.5: dim_nicknames table column names and types.

- **Fingerprints** – fingerprint is a SHA-1 hash of an onion router's identity key (see Subsection *Onion router* in Section 2.2) encoded in Base64. This dimension table is referenced from the fact table by fingerprint_id column.

dim_fingerprints	
id	INTEGER
fingerprint	VARCHAR(27)

Table 4.6: dim_fingerprints table column names and types.

- **Digests** – digest is a SHA-1 hash of an onion router's most recent descriptor encoded in Base64. This dimension table is referenced from the fact table by digest_id column.

dim_digests	
id	INTEGER
digest	VARCHAR(27)

Table 4.7: dim_digests table column names and types.

- **Versions** – version of the Tor protocol running on an onion router. Version is stripped of the „Tor“ string. This dimension table is referenced from the fact table by `version_id` column.

dim_versions	
id	INTEGER
version	VARCHAR(20)

Table 4.8: `dim_versions` table column names and types.

- **DNS names** – reverse DNS name retrieved by querying the PTR record of the onion router's IP address. This dimension table is referenced from the fact table by `dns_name_id` column.

dim_dns_names	
id	INTEGER
dns_name	VARCHAR(80)

Table 4.9: `dim_dns_names` table column names and types.

- **Flags** – combinations of flags that the authorities assigned to the onion router. The fact table references the record that has all the assigned flags for the onion router set to `True`. `id` column is of type `SMALLINT` because there are only $2^{12} = 4096$ combinations. This dimension table is referenced from the fact table by `flags_id` column.

dim_flags	
id	SMALLINT
authority	BOOLEAN
bad_exit	BOOLEAN
exit_	BOOLEAN
fast	BOOLEAN
guard	BOOLEAN
hsdir	BOOLEAN
no_ed_consensus	BOOLEAN
stable	BOOLEAN
stable_desc	BOOLEAN
running	BOOLEAN
valid	BOOLEAN
v2dir	BOOLEAN

Table 4.10: `dim_flags` table column names and types.

- **IP addresses** – current IPv4 or IPv6 address of the onion router. `ip_addr` column is of type `VARBINARY(16)` since it can store both IPv4 and IPv6 addresses and convert between their string and binary representations with `INET6_NTOA` and `INET6_ATON` functions. This dimension table is referenced from the fact table by `ip_addr_id` column and uses MyISAM storage engine because ColumnStore does not support `VARBINARY` data type.

dim_ip_addresses	
id	INTEGER
ip_addr	VARBINARY(16)

Table 4.11: `dim_ip_addresses` table column names and types.

- **GeoLite2** – geolocation data from GeoLite2 database for existing IP addresses in `dim_ip_addresses` dimension table. It contains information about the network and autonomous system which the onion router IP address belongs to, along with the location information. Full GeoLite2 database is in `geolite_blocks` and `geolite_locations` tables, this table holds only a subset of data from those tables. It is denormalized on purpose (there are duplicate column values) which results in larger size but faster data retrieval. This dimension table is referenced from the fact table by `geolite_id` column and uses MyISAM storage engine because ColumnStore does not support `VARBINARY` data type.

dim_geolite	
id	INTEGER
network_start	VARBINARY(16)
network_end	VARBINARY(16)
latitude	FLOAT
longitude	FLOAT
as_num	INTEGER
as_org	VARCHAR(256)
continent	VARCHAR(14)
city	VARCHAR(64)
country	VARCHAR(64)
time_zone	VARCHAR(30)
is_in_eu	BOOLEAN

Table 4.12: `dim_geolite` table column names and types.

There is a special table named `ip_addr_to_geo` that maps IP addresses to GeoLite2 data. This table is updated before the data is inserted to `fact_router_entries` table. For each IP address a respective record from `dim_geolite` table is found by checking which network the IP address belongs to. Then the IP address `id` column from `dim_ip_addresses` and GeoLite2 `id` column from `dim_geolite` are stored in this table. This is required for the join of `tmp_router_entries` table with `dim_geolite` table when inserting to `fact_router_entries` table (see Section 5.3).

ip_addr_to_geo	
ip_addr_id	INTEGER
geolite_id	INTEGER

Table 4.13: `ip_addr_to_geo` table column names and types.

Chapter 5

Extending Consensus Parser with MariaDB

This chapter describes the implementation of new parts of Consensus Parser that interact with MariaDB database and original parts that were modified for the purposes of this thesis. First, used technologies with their respective versions are listed. Next, custom database functions are described, followed by the connecting to the database and its initialisation. Then, the insertion of data from consensus status documents to database is presented. After that, extensions to the Toreator API are described. The chapter closes with a section about the limitations of ColumnStore storage engine.

Consensus Parser is implemented in Python 3.6[1] programming language and therefore all implementation done within this thesis is in Python 3.6 too. For REST API functionality, Conensus parser used Flask[2] framework version 1.0.2 and that was not changed. MariaDB[3] Server version is 10.3.13 and MariaDB ColumnStore[4] version is 1.2.3-GA. SQLAlchemy[5] 1.3.1 and PyMySQL[6] 0.9.3 are utilised for interacting with MariaDB. SQLAlchemy automatically handles connection pooling, it supports SQL statements creation with Python methods and can abstract the specific database to some extent. It has two main components – object-relational mapping (ORM) and Core. ORM is not used in the implementation because of the limitations of ColumnStore, such as a lack of support for `FOREIGN KEY` constraints, and because Core provides more fine grained control over SQL statements.

The root directory with the source code and the name of the repository is `TorConsensus-Parser`. Inside this directory there are all source code files for the original Consensus Parser, some of which were modified for the needs of this thesis. Main module of the application is `consensus_parser.py` which accepts various command line arguments and is the only executable module along with `rest.py` and `profilerest.py`. Consensus Parser was modified to use Python's `logging` module. The configuration is in YAML[7] format in `logging.yml` file and it's loaded in the `logging_config.py` module. The source code files concerning the database reside in the `database` Python package. Full directory structure can be seen in Appendix A.

[1] https://www.python.org/downloads/release/python-360
[2] http://flask.pocoo.org
[3] https://mariadb.com
[4] https://mariadb.com/kb/en/library/mariadb-columnstore
[5] https://www.sqlalchemy.org
[6] https://github.com/PyMySQL/PyMySQL
[7] https://yaml.org

5.1 Custom database functions

This section describes the database functions that were created for the purposes of this thesis when some functionality was required that MariaDB did not provide. Their usage will be further described in this chapter. List of the functions with respective descriptions follows:

`get_start_ipv4_of_cidr`

: Takes a `VARCHAR(19)` containing an IPv4 network address in CIDR notation and returns a `VARBINARY(4)` with the first IP address in that network.

`get_end_ipv4_of_cidr`

: Takes a `VARCHAR(19)` containing an IPv4 network address in CIDR notation and returns a `VARBINARY(4)` with the last IP address in that network.

`get_start_ipv6_of_cidr`

: Takes a `VARCHAR(50)` containing an IPv6 network address in CIDR notation and returns a `VARBINARY(16)` with the first IP address in that network.

`get_end_ipv6_of_cidr`

: Takes a `VARCHAR(50)` containing an IPv6 network address in CIDR notation and returns a `VARBINARY(16)` with the last IP address in that network.

`get_start_ip_of_cidr`

: Takes a `VARCHAR(50)` containing a network address in CIDR notation, determines if it is IPv4 or IPv6 and calls `get_start_ipv4_of_cidr` or `get_start_ipv6_of_cidr` accordingly.

`get_end_ip_of_cidr`

: Takes a `VARCHAR(50)` containing a network address in CIDR notation, determines if it is IPv4 or IPv6 and calls `get_end_ipv4_of_cidr` or `get_end_ipv6_of_cidr` accordingly.

`get_geoblock_id_for_ip`

: Takes a `VARBINARY(16)` with an IP address and returns an `INTEGER` with the `id` of the block from `geolite_blocks` table that the address belongs to.

`get_dim_geolite_id_for_ip`

: Takes a `VARBINARY(16)` with an IP address and returns an `INTEGER` with the `id` of the block from `dim_geolite` table that the address belongs to.

`get_asn_num_for_range`

: Takes two `VARBINARY(16)` variables with start and end IP addresses of a network and returns an `INTEGER` with the number of the autonomous system that the network is a part of. Autonomous system number is looked up in a temporary table `tmp_asn_blocks` (see Figure 5.1) that contains information from GeoLite2 database.

`get_asn_org_for_range`

Takes two `VARBINARY(16)` variables with start and end IP addresses of a network and returns a `VARCHAR(256)` with the name of the organization of the autonomous system that the network is a part of. Organization of the autonomous system is looked up in a temporary table `tmp_asn_blocks` (see Figure 5.1) that contains information from GeoLite2 database.

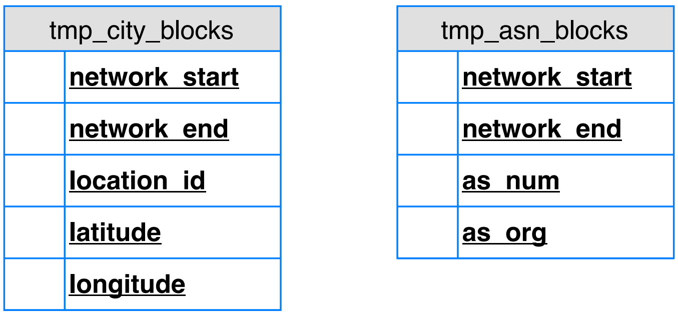


Figure 5.1: Temporary tables that are created when loading GeoLite2 data to MariaDB. They are filled with city and ASN blocks data from GeoLite2 CSV files and then merged into the `geolite_blocks` table. These tables use MyISAM storage engine since they store network addresses in `VARBINARY` data type.

5.2 Database connection and initialisation

Encapsulation of database connection is done in the `DBWrapper` class. It handles the creation of SQLAlchemy Engine object with database configuration from `db.yml` file. Configuration is in YAML format and contains connection parameters such as hostname, port, username, password and database name, and connection pool related settings. `DBWrapper` provides database connections by the `get_connection` method and loading CSV files by the `load_csv_file` method. The connection is automatically retrieved from the connection pool and returned there after the caller finishes its work. `load_csv_file` creates and executes a `LOAD DATA LOCAL INFILE` statement that reads the data directly from the file.

The initialisation of the database is composed of four main parts:

- **Creation of the tables** – all persistent tables that can be seen in Figure 4.1 are defined and created by the `TorTable` class from the `tortable.py` module. Each table is an instance of SQLAlchemy `Table` class and a class variable of the `TorTable` class. The `TorTable` class has only one method – `create_all` – that takes a connection as an argument and creates all tables.

- **Creation of the database functions** – all SQL functions are defined and created by the `Function` class that is a subclass of the `Enum` class. It is defined in the `function.py` module. Each function is an enum member that is a tuple with the function name and `CREATE FUNCTION` SQL statement. Functions are created by the `create_all` method that takes a connection as an argument and creates all functions using their SQL statements. The `Function` class contains methods `get_name` and `get_create_sql` for extracting respective parts of the tuple, and the `get_str_call_with_args` method that takes an argument list and returns a string with function call, e.g. „`func(arg1, arg2)`".

- **Initialisation of date, time and flags dimensions** – these three dimensions are not dynamic and do not depend on the inserted data. In consequence, they can be initialised on database creation. Initialisation happens in the `init_dimensions.py` module. Each dimension is truncated at first and then a CSV file with all necessary combinations (e.g. all possible combinations of hours, minutes and seconds) is created, loaded into the database and deleted.

- **Downloading GeoLite2 data and loading it to MariaDB** – downloading and loading GeoLite2 data is in the `load_geolite.py` module. First, the data is downloaded from MaxMind's[8] website and extracted from the ZIP archives. See Table 5.1 for the description of the files. Then, location data is inserted directly from the CSV file `GeoLite2-City-Locations-en.csv` into `geolite_locations` table. Next, temporary tables `tmp_city_blocks` and `tmp_asn_blocks` are created (see Figure 5.1). These are filled with the data from city and ASN blocks CSV files respectively, both IPv4 and IPv6 versions. Data from both tables is then merged and inserted into `geolite_blocks` table. But since network blocks in temporary tables are different, `get_asn_num_for_range` and `get_asn_org_for_range` functions are used for retrieving ASN data when joining the temporary tables.

Database initialisation can be executed by running `consensus_parser.py` with `--init_db` argument. Inividual steps can also be executed with `--create_db_tables`, `--create_db_functions`, `--init_dimensions` and `--load_geolite` arguments.

Name	**Content**
`GeoLite2-ASN-Blocks-IPv4.csv`	IPv4 network address in CIDR format, autonomous system number and autonomous system organization.
`GeoLite2-ASN-Blocks-IPv6.csv`	IPv6 network address in CIDR format, autonomous system number and autonomous system organization.
`GeoLite2-City-Blocks-IPv4.csv`	IPv4 network address in CIDR format, location id, latitude and longitude.
`GeoLite2-City-Blocks-IPv6.csv`	IPv6 network address in CIDR format, location id, latitude and longitude.
`GeoLite2-City-Locations-en.csv`	Location id, continent, country, city, time zone and a binary value indicating if the location is in the European Union.

Table 5.1: GeoLite2 database CSV files content description. Some fields that are not used in this thesis were omitted.

[8]https://www.maxmind.com

5.3 Consensus status data insertion

Consensus status data is parsed from consensus status documents in the `consensus_-parser.py` module when it is executed with `--consensus_path` argument with the path to the directory that contains consensus status documents. This directory is then recursively scanned and each file that has valid-after timestamp newer than the latest one stored is processed. First, an instance of the `ConsensusStatus` class is created by parsing the file's content and storing consensus status valid-after, fresh-until and valid-until timestamps and a list of the `OnionRouter` class instances with all routers that it contains. The `OnionRouter` class encapsulates information about an onion router from consensus status document (see Subsection *Consensus status* for router entry description and Figure 2.3 for router entry example). If `consensus_parser.py` is executed with `-a` or `--additional_info` options, the `DnsReverseResolver` class from `dns.py` module is used to resolve reverse DNS name for each onion router. To make resolving faster, it is done in parallel using Python's `Pool` class and its methods `map_async` and `map`. From this point on, consensus status may be stored to preprocessed files (see Section 2.5) or to the database. This is controlled by `--write_to_db` option of Consensus Parser. Since this thesis is about storing consensus status data to the database, the following text describes the process with this option.

```sql
INSERT INTO dim_nicknames
  (nickname)
SELECT DISTINCT j.nickname
  FROM tmp_router_entries AS j
       LEFT JOIN dim_nicknames AS dim
                 ON j.nickname = dim.nickname
 WHERE dim.nickname IS NULL
   AND j.nickname IS NOT NULL;
```

Figure 5.2: SQL code snippet for updating a dimension table with values from the temporary table `tmp_router_entries` that are not yet present in the dimension table. This example shows an update of the `dim_nicknames` dimension table.

After the instance of the `ConsensusStatus` class is created, an instance of the `ConsensusStatusInserter` class is created and its `insert` method is called with the `ConsensusStatus` instance as an argument. This method does the following:

1. Create a temporary table `tmp_router_entries`. This table has columns that correspond to the dimension tables columns except IDs, and facts in the fact table. It uses MyISAM storage engine because one of the columns is the IP address.

2. Insert onion routers data to `tmp_router_entries`. A regular `INSERT` statement is used with `VALUES` clause containing values from all `OnionRouter` class instances in the `ConsensusStatus` instance that was passed to the `insert` method.

3. Insert values that are in `tmp_router_entries` but not in the dimension tables to the dimension tables (excluding `dim_dates`, `dim_times`, `dim_flags` and `dim_geolite`). This is done by doing a `LEFT JOIN` of `tmp_router_entries` and a dimension table with a `WHERE` condition eliminating existing values in the dimension and `NULL` values in `tmp_router_entries`. See Figure 5.2 for example.

4. Insert geolocation data from `geolite_blocks` and `geolite_locations` to `dim_geolite` dimension for all IP addresses in `tmp_router_entries` that do not have a corresponding block in `dim_geolite` yet. That is achieved by inserting geolocation values from the join of `tmp_router_entries` with `geolite_blocks` and `geolite_locations` using `get_geoblock_id_for_ip` function to find the ID of the geolocation block for the IP address.

5. Insert ID of the IP address from `tmp_router_entries` and ID of its corresponding block from `dim_geolite` to `ip_addr_to_geolite` for each IP address in `tmp_router_entries` which does not have its ID in `ip_addr_to_geolite` yet.

6. Insert consensus status data to the fact table by joining `tmp_router_entries` with the dimension tables and `ip_addr_to_geolite`, thus replacing dimensions values with numeric IDs. Join condition compares values but only dimensions IDs and facts are selected. `INSERT INTO SELECT` SQL statement is used for the insertion.

5.4 Extending the Toreator API

Consensus status data is provided by the Toreator REST API described in Subsection *Toreator API* in Section 2.5. One of the aims of this thesis was to extend this API with new types of requests. Toreator API functionality resides in `rest.py`, `rest_utils.py`, `rest_endpoint_creator.py` and `serializable.py` modules. The `serializable.py` module converts the data to HTML or JSON and the other modules are described later in this section.

Abstraction layer

To be able to make queries to the database with parameters from the endpoints, an abstraction layer was added between the REST API and the database. This layer consists of these classes from the `database.py` module:

- `Field` – a class that abstracts what to select, join or filter. It contains instances of all dimension tables from the `TorTable` class but with aliases. This is necessary for joins because publication, valid-after, fresh-until and valid-until keys in the fact table all point to `dim_dates` and `dim_times` dimension tables. Without aliases it would not be possible to distinguish them in a join (see Figure 5.3). There also members of this class that are instances of columns, such as `year` or `or_port`, to be able to filter on them.

- `SelectColumns` – a class that controls what will be in the `SELECT` clause of the query. Its constructor accepts `field_names` argument – a list of strings that correspond to aliases in the `Field` class except for date and time fields which are merged. For example, publication date field has „publish_date“ alias and publication time field has „publish_time“ alias but the string in `field_names` is just „publish“ and both fields are automatically included. Besides aliases from `Field`, `field_names` can contain fact names „or_port“, „dir_port“ and „bandwidth“. All columns (except IDs) of fields from `field_names` are chosen to be selected and if `field_names` is empty, all columns (except IDs) of all fields are chosen to be selected along with facts columns. Columns are instances of SQLAlchemy `Column` object and each is assigned an appropriate label that is used as an SQL alias. Columns chosen to be selected are provided by the `get_columns_to_select` method. `Field` instances that need to be joined with the fact table are provided by the `get_fields_to_join` method.

- `Filter` – a class that determines which tables will be joined with the fact table on what conditions and what conditions will be in the `WHERE` clause of the query. The constructor accepts a list of fields to join with the fact table and a dictionary `filter_dict` mapping the `Field` instances to their desired values. For all fields that are joined with the fact table, a condition that the fact table key equals the field's ID is automatically added to `JOIN ON` clause. `filter_dict` values are transformed into conditions used in `JOIN ON` and `WHERE` clauses by internal filtering functions. SQLAlchemy Join object is provided by the `get_join` method and SQLAlchemy AND conjunction is provided by the `get_where` method.

- `OnionRouterSelector` – a class that executes the `SELECT` statement and returns a list of `OnionRouter` instances. It takes list of columns to select, SQLAlchemy Join and SQLAlchemy AND conjunction as arguments in constructor. The `get_routers` method constructs SQLAlchemy Select object, executes it and returns `OnionRouter` instances. The `OnionRouter` instance is constructed from the database row by the `or_from_db_row` module method. If some fields are missing in the database row, this method just ignores them.

```
SELECT publish_date.year AS publish_date_year,
       publish_date.month AS publish_date_month,
       publish_date.day AS publish_date_day,
       valid_after_date.year AS valid_after_date_year,
       valid_after_date.month AS valid_after_date_month,
       valid_after_date.day AS valid_after_date_day
  FROM fact_router_entries
       JOIN dim_dates AS publish_date
            ON publish_date.id = fact_router_entries.publish_date_id
       JOIN dim_dates AS valid_after_date
            ON valid_after_date.id = fact_router_entries.valid_after_date_id
```

Figure 5.3: Example demonstrating the need for aliases for `dim_dates` dimension table. Since both `publish_date_id` and `valid_after_date_id` keys from the fact table point to `dim_dates`, aliases are necessary. With `dim_times` dimension table it is analogous.

New endpoints

The API was extended with new endpoints allowing to search by different fields than IP address, date and time. This is handled by `rest.py`, `rest_utils.py` and `rest_endpoint_-creator.py` modules. `rest.py` is a main module where the Flask application is started. This module uses the `rest_endpoint_creator.py` module to dynamically generate endpoint functions which are assigned to the `rest.py` module. `rest_endpoint_creator.py` contains classes that encapsulate different types of endpoints (see Subsection *Toreator API* in Section 2.5). Each class implements the `create_endpoints` method that returns a list of pairs. The first element of the pair is a function processing the endpoint type for a specific field or multiple fields. The second element is the path of the endpoint. So each pair in the list represents a specific endpoint for a specific field or multiple fields. The `rest.py` module then calls the `create_endpoints` method of all classes, iterates through all pairs, assigns the functions to itself and creates URL rules based on the paths.

The `rest_utils.py` module contains common functions that are used across multiple endpoints. Most important ones are:

- `get_routers` – accepts a list of parameters from the query component of the URL and a dictionary with fields to filter and their values. Returns a list of `OnionRouter` instances conforming to filter and containing fields specified by the parameters or all fields if no parameters were given. Router entries with identical fields except valid-after and valid-until times are merged it those times overlap.

- `get_field_activity` – accepts a dictionary with fields to filter and their values, and a date format. Returns all valid-after dates in a given date format (or default *%Y-%m-%d* if none was given) conforming to filter.

- `get_all_field_values_links` – accepts an instance of the `Field` class and returns links to the endpoints containing information about the field's values. This method is only usable for fields nickname, fingerprint, digest, IP address and reverse DNS name.

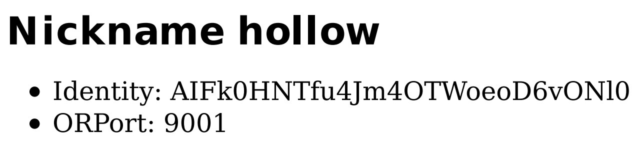
New endpoints are like the ones described in Subsection *Toreator API* in Section 2.5 (except the first three) but in place of `addresses` there can be `nicknames`, `fingerprints`, `digests`, `versions`, `dns_names`, `flags`, `countries`, `cities`, `as_nums` or `as_orgs`. `flags` is specified by a list of comma divided flag names, other fields are just strings or numbers. Of course, `<address>` has to be replaced with a proper value of the aforementioned fields. There are also some endpoints that allow to filter by more than one field. They are listed in Table 5.2.

URL
`/addresses/<address>/nicknames/<nick>/`
`/addresses/<address>/flags/<flags>/`
`/nicknames/<nick>/flags/<flags>/`
`/dns_names/<dns_name>/flags/<flags>/`
`/flags/<flags>/addresses/<address>/`
`/flags/<flags>/dns_names/<dns_name>/`
`/flags/<flags>/countries/<country>/`
`/flags/<flags>/cities/<city>/`
`/flags/<flags>/as_nums/<as_num>/`
`/flags/<flags>/as_orgs/<as_org>/`
`/versions/<version>/addresses/<address>/`
`/countries/<country>/flags/<flags>/`
`/cities/<city>/flags/<flags>/`
`/as_nums/<as_num>/addresses/<address>/`
`/as_nums/<as_num>/flags/<flags>/`
`/as_nums/<as_num>/versions/<version>/`
`/as_orgs/<as_num>/addresses/<address>/`
`/as_orgs/<as_num>/nicknames/<nick>/`
`/as_orgs/<as_num>/flags/<flags>/`

Table 5.2: New endpoints that provide searching by multiple fields. They were chosen according to the discussion with the supervisor of the thesis.

Specifying fields by parameters in the URL

Another new feature is a possibility to include only specified fields in the results. This is controlled by parameters in the query component of the URL with names of the desired fields. The names correspond (with some exceptions, see the `SelectColumns` class description above in this section) to aliases or column names in the `Field` class. For example, if the user is interested only in a nickname, a fingerprint and an onion router port, the request URL would look like this: `/addresses/158.255.3.14/?nickname&fingerprint&or_port`. This would filter the data by the IP address 158.255.3.14 but the result would contain only the desired fields (see Figure 5.4). Because of the abstraction layer, this feature is implemented in a simple manner. A list of parameters is just passed to the `get_routers` method in the `rest_utils.py` module which passes it to the constructor of the `SelectColumns` class. The `SelectColumns` instance then determines the appropriate columns to select. When applied, this feature also offers performance increase because ColumnStore does not have to scan so many columns and fewer joins are done.



Nickname hollow

- Identity: AIFk0HNTfu4Jm4OTWoeoD6vONl0
- ORPort: 9001

Figure 5.4: Result returned by the Toreator REST API when URL parameters are used to specify what should be included in the result. URL parameters are strings corresponding (with some exceptions, see the `SelectColumns` class description above in this section) to aliases and column names in the `Field` class. In this example the URL is `/addresses/158.255.3.14/?nickname&fingerprint&or_port`. Result with all fields can be seen in Figure 2.7.

5.5 MariaDB ColumnStore limitations

ColumnStore does not support `VARBINARY` data type which is a huge drawback since it requires using other engine for tables that need to store IPv6 addresses. This thesis has IPv4 and IPv6 in the same column; therefore, all tables that store IP addresses use MyISAM engine. Performance problems arise from this because cross joins between engines are slower and ColumnStore query optimizer does not seem to take MyISAM or InnoDB indexes into account. This is the reason that some workarounds were used which are not optimal, such as `dim_geolite` dimension. If ColumnStore optimizer took indexes into account and queries were faster, it would be sufficient to store `geolite_blocks` ID in `dim_ip_addresses` table and join on that. This would make data insertion faster and simpler. If `VARBINARY` was supported in ColumnStore it would make queries even faster.

Another problem is a bug in join between ColumnStore and MyISAM or ColumnStore and InnoDB which causes `INET6_NTOA` and `INET6_ATON` functions to not work when the IP address contains `0x00` byte or zero octet or hextet. This is evident especially with GeoLite2 data where the first address of the network often contains `0x00` byte. It also affects searching since no IP address which contains `0x00` byte can be joined with the fact table. Such addresses are only showed by `/addresses/` endpoint because it just selects the addresses from the dimension table without joining with the fact table. I filed a bug[9] in MariaDB JIRA system and got some reponse from a MariaDB developer so hopefully this will be resolved.

Nevertheless, ColumnStore is actively developed and there is a good chance that these problems will be fixed in the near future.

[9] https://jira.mariadb.org/browse/MCOL-2234

Chapter 6

Evaluation and comparison to the original version

This chapter presents the results of the experiments which were done to compare the original version of Consensus Parser to the one created as a part of this thesis. In the first section, various experiments showing performance differences are introduced. In the second section, storage utilisation of both versions is compared. The chapter closes with a section that compares the versions and lists their advantages and disadvantages.

Experiments were done on a machine running Ubuntu[1] Linux version 18.04.2 LTS with Linux core 4.15.0-47-generic. It was a virtual machine running in VMWare with 4 CPU cores and 16 GB RAM. Both versions were running locally on the machine, exposing the API via the loopback interface. MariaDB ColumnStore was installed as a single server configuration, i.e. User Module and Performance Module on the same machine. The database was filled with data from consensus status documents for 2017 and the same documents were given to the original version to transform to preprocessed files. Therefore, both versions were tested on the same data.

6.1 Performance experiments

Experiments were focused on the performance of both versions and their comparison. A request to an endpoint was always sent five times and then mean, maximum and minimum durations were stored. The source code for running experiments resides in the `experiments.py` module in the `experiments` package. The module takes a CSV file with pairs of IP address and time as an input and writes durations of the requests for specific endpoint and version to another CSV file. It can also send requests to Toreator[2] server instead of the local machine. Note that in Chapter 2, Toreator is referred to as the name of the API. In this chapter, it is referred to as the name of the actual server hosted in Brno University of Technology network.

Displaying all information

The first experiment was done to examine the performance of the most computationally expensive task – displaying all information about the onion router with a specific IP address

[1]https://www.ubuntu.com/

[2]Toreator is accessible at http://toreator.fit.vutbr.cz

without any date or time restriction. Two IP addresses were chosen to retrieve information about according to the large amount of data that was stored in the system for both of them. Another criterion was for one address to take a short time to look up in the GeoLite2 CSV files and the other one to take a long time. Since GeoLite2 CSV files contain records ordered by the network from 1.0.0.0 to 255.0.0.0, chosen addresses were 5.79.68.161 and 217.191.97.45. This demonstrated the inefficiency of the original version's approach when the network that the IP address belongs to is located near the end of the file. The results for IP addresses 5.79.68.161 and 217.191.97.45 can be seen in Figure 6.1 and Figure 6.2 respectively.

Figure 6.1 shows that the original version achieves excellent performance when the search for the IP address in GeoLite2 CSV files is fast. In comparison, the new version does not perform so well due to the nature of ColumnStore which is not optimised for retrieving data for all columns. It would be interesting to see what the results would be if the new version supported multiple versions of GeoLite2 database. The fact is that the original version performs worse with more GeoLite2 database versions, as can be seen in Figure 6.3. If the new version supported multiple versions of GeoLite2 database, it might not degrade performance as much as in the original version, because querying the database is in most cases faster than searching directly in files.

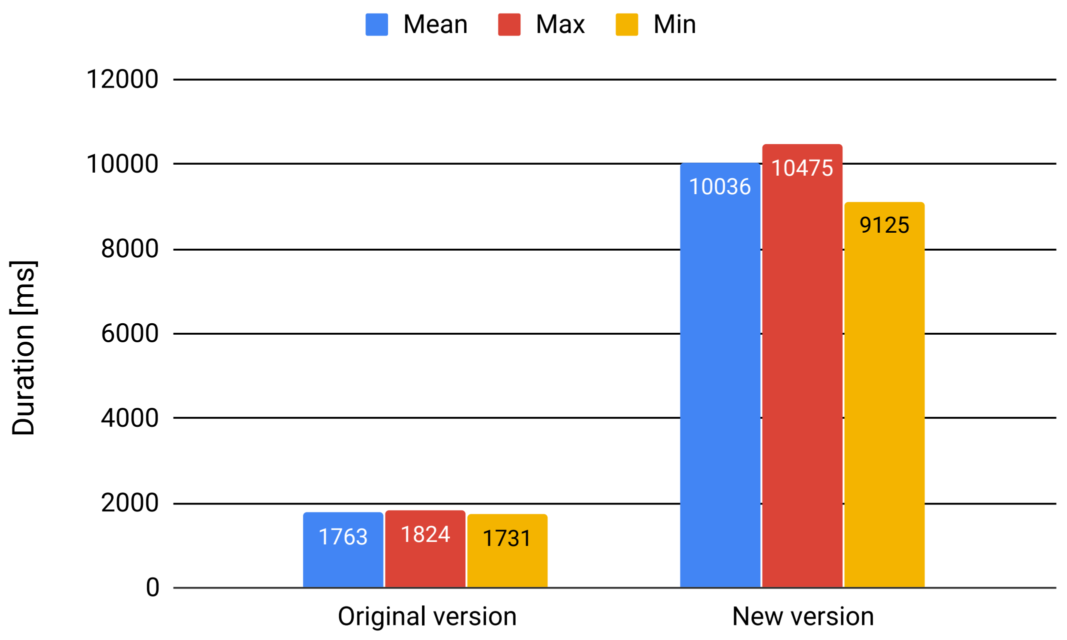


Figure 6.1: Comparison of durations of retrieving all information about the onion router with the IP address 5.79.68.161.

In figure 6.2, the differences between the two versions are much smaller since the old version took more time to search for the geolocation data, but it was still faster then the new version.

To illustrate the performance of the original solution with the data for multiple years and with multiple GeoLite2 database versions, Toreator server was queried too. Toreator runs the original version on a machine with all consensus data since 2008 and many versions of GeoLite2 database.

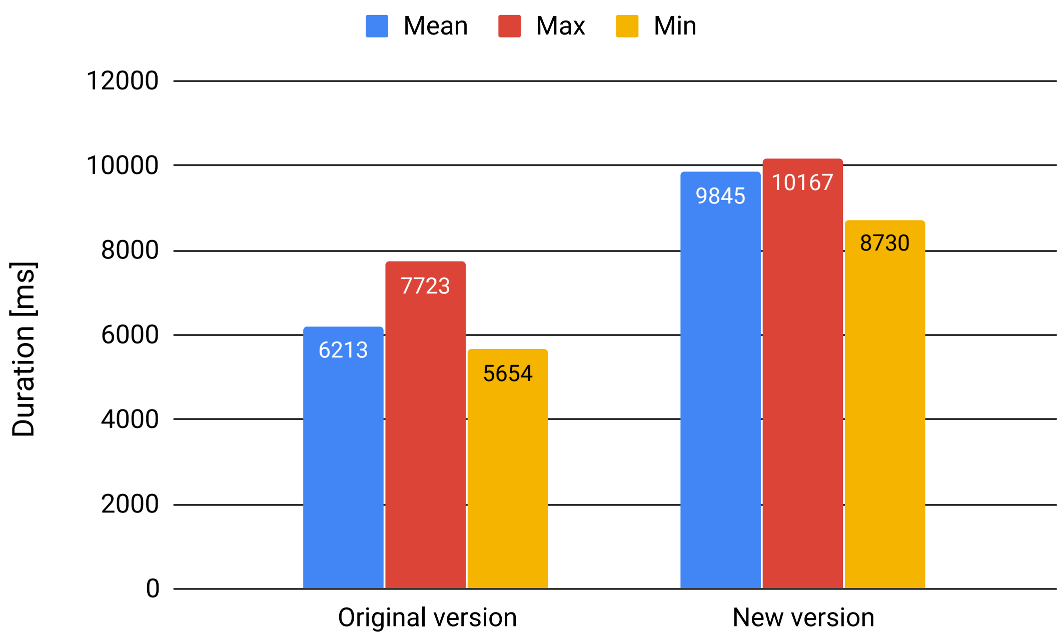


Figure 6.2: Comparison of durations of retrieving all information about the onion router with the IP address 217.191.97.45.

In Figure 6.3, the results of comparing Toreator performance to the original and the new version running on a local machine are displayed. IP address 82.229.26.235 was chosen because there is a lot of data for it for multiple years. It is evident from the figure that Toreator performance is significantly worse. It is caused by the volume of the data, the number of versions of GeoLite2 database and the fact that GeoLite2 database records are looked up directly in the CSV files.

Comparison of various endpoints

In this experiment, mean durations of various endpoints supported by both versions were compared. IP addresses were the same as in Subsection *Displaying all information* in this section. The results in Table 6.1 show that durations for the endpoints such as `addresses/5.79.68.161/date` or `addresses/5.79.68.161/month` that show only the onion router activity (links to endpoints that display information about an onion router for a time period) are relatively comparable for both versions. The endpoints with more notable differences between the versions are the ones that return information about the onion router. And the longer the time period, the longer the duration of the request, except for the original version endpoints `addresses/5.79.68.161/time/2017-07-01 12:00:00` and `addresses/217.197.91.145/time/2017-07-01 12:00:00`. These take more time because the original version has to parse valid-after and valid-until times as UNIX timestamps and compare them to the given time. For dates, it just checks the prefixes of valid-after and valid-until. For IP address 217.197.91.145, the differences are smaller because the original version took more time to retrieve the geolocation data than for IP address 5.79.68.161.

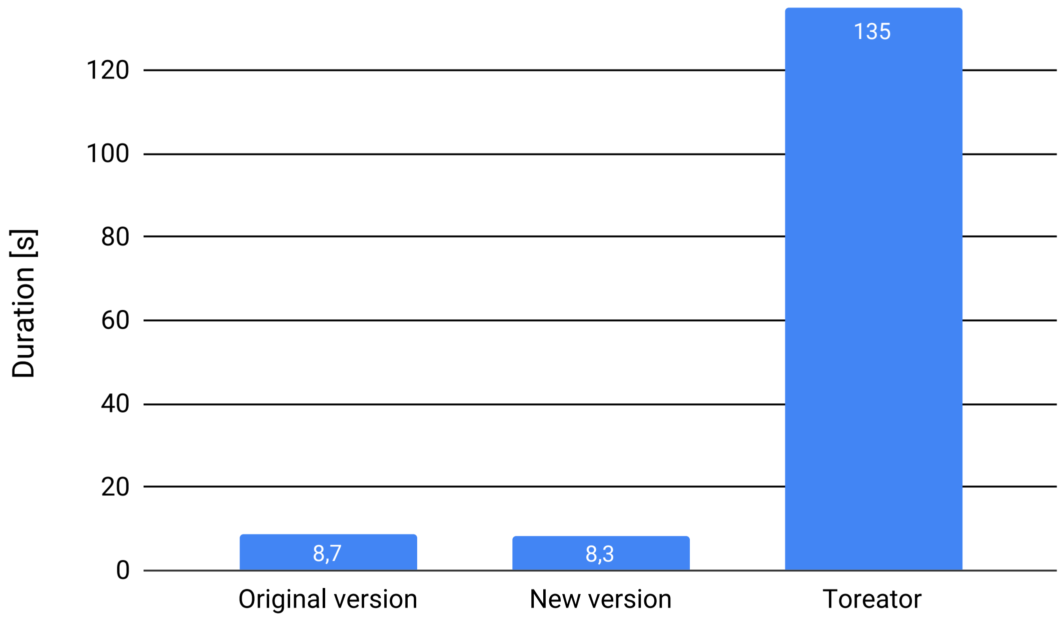


Figure 6.3: Comparison of mean durations of retrieving all information about the onion router with the IP address 82.229.26.235. Both versions running on a local machine were queried as well as the Toreator server.

Endpoint	**Original version**	**New version**
`addresses/5.79.68.161`	1763	10036
`addresses/5.79.68.161/date`	383	1297
`addresses/5.79.68.161/time`	388	1305
`addresses/5.79.68.161/month`	562	1284
`addresses/5.79.68.161/year`	595	1299
`addresses/5.79.68.0-24`	3	15
`addresses/5.79.68.161/time/2017-07-01 12:00:00`	2549	4764
`addresses/5.79.68.161/date/2017-07-01`	409	4375
`addresses/5.79.68.161/month/2017-07`	529	4824
`addresses/5.79.68.161/year/2017`	1791	10083
`addresses/217.197.91.145`	6213	9845
`addresses/217.197.91.145/date`	354	1305
`addresses/217.197.91.145/time`	369	1269
`addresses/217.197.91.145/month`	552	1296
`addresses/217.197.91.145/year`	530	1311
`addresses/217.197.91.0-24`	3	13
`addresses/217.197.91.145/time/2017-07-01 12:00:00`	6645	4556
`addresses/217.197.91.145/date/2017-07-01`	4270	4559
`addresses/217.197.91.145/month/2017-07`	4453	5059
`addresses/217.197.91.145/year/2017`	5726	9341

Table 6.1: Mean durations in milliseconds for requests to specific endpoints.

Limiting the returned fields

This experiment was done to prove that ColumnStore achieves a notable performance gain when the number of columns it has to read is limited. The feature described in Subsection *Specifying fields by parameters in the URL* in Chapter 5 was used to limit the information returned by the API. It is done by specifying parameters in the query component of the URL. Count of retrieved fields ranging from one to thirteen was tested. The results in Figure 6.4 show that limiting the returned fields indeed improves the performance and shortens the duration of the request.

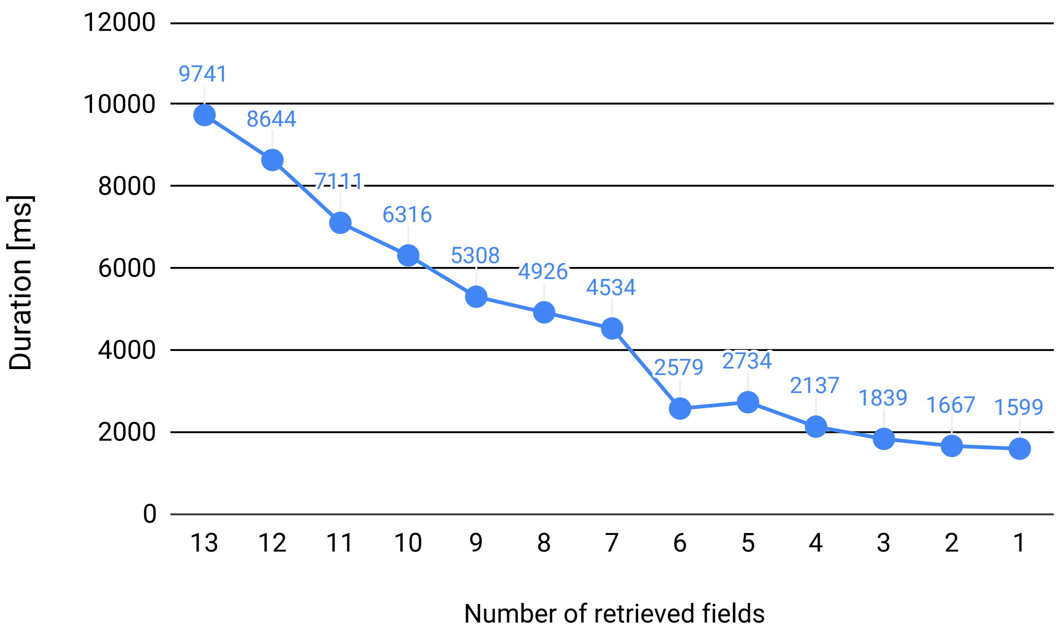


Figure 6.4: Mean durations of the request with respect to different number of retrieved fields.

6.2 Storage utilisation

Figure 6.5 displays the difference in storage size for both versions for 2017 consensus data. It is apparent that MariaDB database takes almost five times less space than preprocessed files. This is due to the design of the data schema which has a large fact table that contains only numeric keys that point to the dimension tables. ColumnStore also utilises compression, achieving even more space savings. In comparison, preprocessed files store all data redundantly as strings, so their size increases linearly with addition of more data.

Both versions use GeoLite2 database for geolocation data and the original version also supports multiple versions of this database. The size of one version (in CSV files) ranges from 250 MB to 340 MB. So for the whole year it is approximately $52 \times 300 = 15600$ MB, but since the new version does not support multiple GeoLite2 versions, only the size of one version was included. In the new version, GeoLite2 is stored in tables with MyISAM storage engine and it can be seen in Figure 6.5 that its size is about 300 MB. The original version uses the CSV files directly.

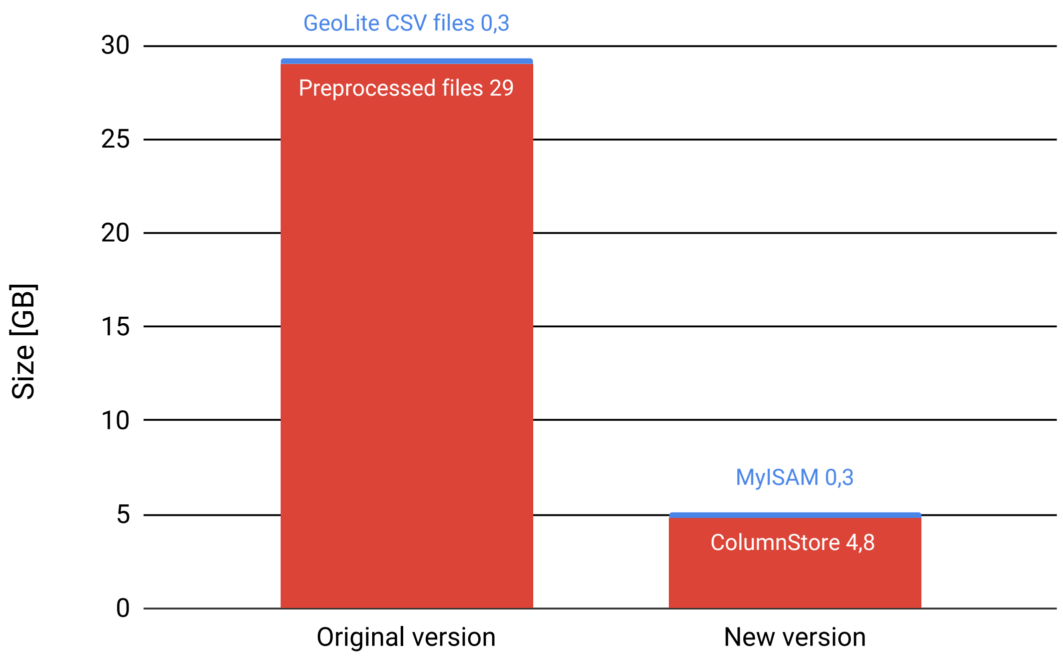


Figure 6.5: Comparison of the size of consensus data for 2017 in the original version and the new version.

6.3 Comparison of the versions

Main disadvantages of the original version are the limited searching options, poor storage utilisation and slow retrieval of GeoLite2 data. The biggest original version's advantage is the hierarchical structure of the directories with preprocessed files. Because of it, the duration of retrieving is shorter than in the new version if only one GeoLite2 database version is used. Another advantage is having multiple versions of GeoLite2 database which means the geolocation data reflect the state in a specific time period, not just the current state as in the new version. However, because geolocation data is looked up in CSV files, request duration suffers. And the more versions (more files), the worse the performance.

Main drawbacks of the new version are a lack of GeoLite2 database versioning and query performance. The performance problem is due to the fact that retrieving data from all columns is not a use case where ColumnStore is very efficient. Its advantages show more when the retrieved results are limited to only some columns because then it does not have to access the disk that much. The mixing of ColumnStore and MyISAM is also not ideal.

As for the advantages, the new version provides 121 endpoints while the original version provides only 12. That is a tenfold increase which extends the searching options considerably. The new version is much more effective in storage utilisation. The size of the database with all consensus data for the year 2017 is 5.1 GB (ColumnStore tables take up 4.8 GB, MyISAM tables take up 338 MB). Preprocessed files for the year 2017 take up 29 GB, almost five times more. An additional benefit of the new version is that it brings the power of SQL to a user that has direct access to the database and some SQL knowledge. This provides possibilities for various analytical queries, e.g. a calculation of the average bandwidth, finding the most frequently used port or the country with the most onion routers.

This also means that much more information can be extracted from the new version than from the original version.

Some performance improvement can be achieved in the new version by using a multi server configuration for ColumnStore and thus utilising distributed processing on multiple User and Performance Modules (see Section 3.5). Running ColumnStore on bare metal server instead of a virtual machine will probably increase the performance too.

Chapter 7

Conclusion

The aim of this thesis was to modify and improve Consensus Parser – a tool for storing and providing information about the Tor network – to use an OLAP database and improve its searching and filtering capabilities. This was accomplished by designing a database schema for storing Tor network consensus data that conforms to OLAP needs, creating a database based on the schema and integrating it into Consensus Parser. Created solution provides enhanced searching functionality which includes searching by multiple dimensions and retrieving only specified fields. To give context to the reader, the Tor network and the Tor directory protocol were introduced along with Consensus Parser and the data it stores. Further, OLAP and multidimensional data model was described and research was done to examine support for OLAP in open source databases PostgreSQL and MariaDB. Based on the research, the latter was chosen as an appropriate database for Consensus Parser because it has ColumnStore storage engine which is suited exactly for OLAP.

Results of the experiments revealed that the new version is a little bit slower than the original version when the original version uses only one version of GeoLite2 database. But when retrieving only specified fields – a new feature – is used, the performance of the new version increases significantly. Main accomplishment of the new version is the addition of 109 endpoints to 12 existing ones to allow more complex search, and reducing the disk usage by a factor of five.

Possible improvements of the new version of Consensus Parser are versioning of the GeoLite2 geolocation database, adding onion router policy and supported protocols information to stored information, and storing Tor protocol version as a numeric value instead of a string. Adjusting unit tests and adding integration tests would be a reasonable enhancement too.

Bibliography

[1] Abadi, D.: The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends® in Databases*. vol. 5, no. 3. 2012: pp. 197–280. ISSN 1931-7883. doi:10.1561/1900000024.

[2] Chaudhuri, S.; Dayal, U.: An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.*. vol. 26, no. 1. March 1997: pp. 65–74. ISSN 0163-5808.

[3] Childs, L.: Onionite. [Online; visited 30.12.2018].
Retrieved from: https://onionite.now.sh

[4] Dingledine, R.; Mathewson, N.: Tor directory protocol, version 3. [Online; visited 18.10.2018].
Retrieved from: https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt

[5] Dingledine, R.; Mathewson, N.: Tor Protocol Specification. [Online; visited 16.10.2018].
Retrieved from: https://gitweb.torproject.org/torspec.git/tree/tor-spec.txt

[6] Dingledine, R.; Mathewson, N.; Syverson, P.: Tor: The Second-Generation Onion Router. In *13th USENIX Security Symposium*. San Diego, CA, USA. August 2004.

[7] Golfarelli, M.: *Data warehouse design*. New York: McGraw-Hill. third edition. c2009. ISBN 978-0-07-161039-1.

[8] MariaDB: About MariaDB Software - MariaDB Knowledge Base. [Online; visited 8.4.2019].
Retrieved from: https://mariadb.com/kb/en/library/about-mariadb-software

[9] MariaDB: ColumnStore Data Types - MariaDB Knowledge Base. [Online; visited 28.4.2019].
Retrieved from: https://mariadb.com/kb/en/library/columnstore-data-types

[10] MariaDB: ColumnStore Performance Module - MariaDB Knowledge Base. [Online; visited 8.4.2019].
Retrieved from:
https://mariadb.com/kb/en/library/columnstore-performance-module

[11] MariaDB: ColumnStore Storage Architecture - MariaDB Knowledge Base. [Online; visited 8.4.2019].
Retrieved from:
https://mariadb.com/kb/en/library/columnstore-storage-architecture

[12] MariaDB: ColumnStore User Module - MariaDB Knowledge Base. [Online; visited 8.4.2019].
Retrieved from: https://mariadb.com/kb/en/library/columnstore-user-module

[13] MariaDB: MariaDB ColumnStore - MariaDB Knowledge Base. [Online; visited 8.4.2019].
Retrieved from: https://mariadb.com/kb/en/library/mariadb-columnstore

[14] MariaDB: MariaDB License - MariaDB Knowledge Base. [Online; visited 8.4.2019].
Retrieved from: https://mariadb.com/kb/en/library/mariadb-license

[15] MariaDB: MariaDB versus MySQL - Compatibility - MariaDB Knowledge Base. [Online; visited 8.4.2019].
Retrieved from:
https://mariadb.com/kb/en/library/mariadb-vs-mysql-compatibility

[16] MariaDB: VARBINARY - MariaDB Knowledge Base. [Online; visited 28.4.2019].
Retrieved from: https://mariadb.com/kb/en/library/varbinary

[17] Neal, H.: Onion diagram. [Online; visited 07.10.2018].
Retrieved from: https://en.wikipedia.org/wiki/File:Onion_diagram.svg

[18] OLAP Council: OLAP and OLAP Server Definitions. [Online; visited 21.1.2019].
Retrieved from: http://www.olapcouncil.org/research/resrchly.htm

[19] Oracle Corporation: MySQL :: MySQL 8.0 Reference Manual :: 8.11.1 Internal Locking Methods. [Online; visited 6.5.2019].
Retrieved from:
https://dev.mysql.com/doc/refman/8.0/en/internal-locking.html

[20] Ponniah, P.: *Data warehousing fundamentals*. New York: Wiley. 2001. ISBN 04-714-1254-6.

[21] The PostgreSQL Global Development Group: PostgreSQL: About. [Online; visited 6.4.2019].
Retrieved from: https://www.postgresql.org/about

[22] The PostgreSQL Global Development Group: PostgreSQL: Documentation: 11: Appendix D. SQL Conformance. [Online; visited 6.4.2019].
Retrieved from: https://www.postgresql.org/docs/current/features.html

[23] The PostgreSQL Global Development Group: PostgreSQL: Feature Matrix. [Online; visited 6.4.2019].
Retrieved from: https://www.postgresql.org/about/featurematrix

[24] The Tor Project: Abuse FAQ. [Online; visited 07.10.2018].
Retrieved from: https://www.torproject.org/docs/faq-abuse

[25] The Tor Project: Bridges. [Online; visited 30.12.2018].
Retrieved from: https://www.torproject.org/docs/bridges

[26] The Tor Project: FAQ. [Online; visited 06.10.2018].
Retrieved from: https://www.torproject.org/docs/faq

[27] The Tor Project: How Tor works. [Online; visited 07.10.2018]. Retrieved from: https://www.torproject.org/images/htw2.png

[28] The Tor Project: The Tor network status protocol. [Online; visited 30.12.2018]. Retrieved from: https://metrics.torproject.org/onionoo.html

[29] The Tor Project: Who uses Tor? [Online; visited 06.10.2018]. Retrieved from: https://www.torproject.org/about/torusers

Appendix A

CD contents

latex-source – A directory with LaTeX source codes of the thesis.

michal-chomo-thesis.pdf – The text of this thesis in Portable Document Format.

TorConsensusParser – A directory with source codes of Consensus Parser and a Python virtual environment with required libraries.

Below is a detailed view of `TorConsensusParser` directory. Directories are listed first, then files. `testfiles` and `venv` directories contents are omitted.

```
TorConsensusParser
 |__database
 |   |__consensus_status_inserter.py
 |   |__db.yml
 |   |__db_wrapper.py
 |   |__field.py
 |   |__filter.py
 |   |__function.py
 |   |__init_dimensions.py
 |   |____init__.py
 |   |__install-mariadb-columnstore.sh
 |   |__load_geolite.py
 |   |__onion_router_inserter.py
 |   |__onion_router_selector.py
 |   |__select_columns.py
 |   |__tortable.py
 |__testfiles
 |__venv
 |__consensus_parser.py
 |__consensus_status.py
 |__dns.py
 |__experiments.py
 |__geolite2.py
 |____init__.py
 |__LICENSE
 |__logging_config.py
 |__logging.yml
```

```
│
├── onion_router.py
├── parameterizable_tc.py
├── profilerest.py
├── README
├── requirements.txt
├── rest_endpoint_creator.py
├── rest.py
├── rest_utils.py
├── runserver.sh
├── serializable.py
├── test_additional_info.py
├── test_cp.py
├── test_files.py
├── test_geolite2.py
├── test_or.py
├── test.py
├── test_rest.py
├── test_serializable.py
├── test_time.py
└── time_parser.py
```