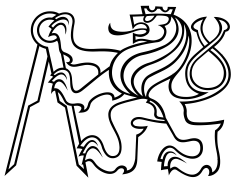
Czech Technical University in Prague
Faculty of Civil Engineering

Master thesis

Prague 2016

Bc. Matěj Krejčí

Czech Technical University in Prague
Faculty of Civil Engineering
Geodesy and Cartography
Geomatics



Master thesis

Processing of vector data using distributed database systems in GIS

Využití distribuovaných databázových systémů pro správu vektorových dat v GIS

Supervisor: Ing. Martin Landa, Ph.D.

Department of Geomatics

Prague 2016

Bc. Matěj Krejčí

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta stavební
Thákurova 7, 166 29 Praha 6

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: Krejčí Jméno: Matěj Osobní číslo: 384659

Zadávající katedra: Katedra geomatiky

Studijní program: Geodézie a kartografie

Studijní obor: Geomatika

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce: Využití distribuovaných databázových systémů pro správu vektorových dat v GIS

Název diplomové práce anglicky: Processing of vector data using distributed database systems in GIS

Pokyny pro vypracování:
Diplomová práce se věnuje analýze technologií pro zpracování velkého množství dat (tzv. bigdata). Konktrétně je zaměřena na uložení a správu časoprostorových vektorových dat v prostředí distribuovaných databázových systémů jako je např. Hadoop. Cílem praktické části práce je jejich zpřístupnění a umožnění analýz v prostředí desktopového open source GIS nástroje GRASS GIS. Testování navrženého řešení bude prováděno s využitím virtualizované sítě uzlů tvořících cluster.

Seznam doporučené literatury:
Hadoop: The Definitive Guide O'Reilly Media, Inc. 2012 ISBN: 9781449311520
Programming Hive 1st O'Reilly Media, Inc. 2012 ISBN: 9781449319335
Big Data: Techniques and Technologies in Geoinformatics ISBN: 9781466586512

Jméno vedoucího diplomové práce: Ing. Martin Landa, Ph.D.

Datum zadání diplomové práce: 22.2.2016 Termín odevzdání diplomové práce: 22.5.2016

[redacted] [redacted]

Podpis vedoucího práce Podpis vedoucího katedry

ZDE VLOŽIT ORIGINÁLNÍ ZADÁNÍ

III. PŘEVZETÍ ZADÁNÍ

Beru na vědomí, že jsem povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je nutné uvést v diplomové práci a při citování postupovat v souladu s metodickou příručkou ČVUT „Jak psát vysokoškolské závěrečné práce“ a metodickým pokynem ČVUT „O dodržování etických principů při přípravě vysokoškolských závěrečných prací“.

[redacted] [redacted]

Datum převzetí zadání Podpis studenta(ky)

Abstract

The goal of presented thesis lies in the design and development of workflow for management and processing of big spatial data within GRASS GIS environment. The thesis explains necessary fundamentals to understand aspects of Hadoop ecosystem. After that, the distributed spatial processing is analyzed as well as available spatial frameworks for Hadoop. Moreover, the configuration and deployment of Hadoop cluster using cloud platform is provided.

Several command line modules for interaction between GRASS and Hadoop/Hive are implemented in developed GRASS Hadoop Framework. The framework allows controlling Hadoop spatial libraries using GRASS. The connection manager for different drivers is also included. Therefore, both side data conversion put/get data to HDFS and Hive table management is provided. As a selected case of developed framework the Europe extraction of Open Street Map history dataset, which included approx. 1.3 billions of points, has been processed and visualized from GRASS GIS.

Keywords: Hadoop, GRASS GIS, Big Data, Spatial Processing

Abstrakt

Cílem práce je návrh řešení pro zpracovaní vektorových dat velkého objemu z prostředí GRASS GIS a jeho implementace. V práci jsou popsány principy systému Hadoop a jeho komponent. Dále text navazuje rešerší nastrojů, které umožňují prostorové analýzy s využitím systému Hadoop a jejich porovnáním. Závěr první kapitoly se zabývá seznámením s cloudovými řešeními pro konfiguraci a spuštění Hadoop clusteru.

V rámci praktické časti byl implementován GRASS Hadoop Framework, který obsahuje moduly umožňující komunikaci mezi GRASS a Hadoop/Hive. Tímto nástrojem se ovládají knihovny umožňující prostorové analýzy s využitím Hadoop a je zajištěna konverze vektorových map, jejich přenos a správa tabulek v databázi. Pro správu účtů jednotlivých připojení je implementován modul s SQL rozhraním. Vytvořený nástroj byl otestován nad datasetem OpenStreetMap pro území Evropy, kde bylo zpracováno a vizualizováno 1.3 miliardy bodů z prostředí GRASS GIS.

Klíčová slova: Hadoop, GRASS GIS, Big Data, prostorové analýzy

Declaration of authorship

I declare that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged. Formulations and ideas taken from other sources are cited as such.

In Prague ....

(author sign)

Acknowledgment

I would like to thank my parents for their support during my studies. Great gratitude, I would like to express Martin Landa, my supervisor, for giving sense to my university studies.

Contents

Introduction

Context

Over the years, the capacity of hard drives have increased massively and access speed too. According to the study undertaken by International Data Corp, since 2007 we produce more data than we can store. Furthermore, the generators of data growing fast which is proofed by the fact that 90% of data volume has been generated since 2010. Current annual global data production is 7.9 zettabytes and the trend points 35 zettabytes in 2020 due to 30 times faster generation of data[1]. The term *big data* became as widely used expression for the volume of data which beyond the ability of commonly used software to store and process them. During decades of innovation, SQL relational databases reached limits thank it's architecture design hand by hand with the limitation of hardware components. Software engineering has been naturally pushed to investigate new form for handling big amount of data efficiently. As the new phenomenon of storing and managing big data in data centers over the word became approach based on distributed file system and parallel computing. At the beginning of this stage, big companies developed prototypes of software which was strongly dependent on maintenance by top experts from the field. The turning point of the blocker has changed by releasing Hadoop framework, especially after it became a part of Apache project. Since that, many side projects based on Hadoop are introduced and together became a synonym of distributed ecosystem. The significant generator of the total data amount are sources of spatial data. Geoscience in last decades points strong dependency on geographical information systems (GIS). This concept, firstly proposed in 1960, gone through the long process of development, and with increasing computational hardware capacity became as the standard effective tool for user workstations.

With technological improvements in the field of measuring, data are currently captured with higher spatial and temporal resolution. To satisfy requirements of efficient data manipulation , analysis and data storage come out several methods based expensive high-performance hardware accompanied with technology for parallel processing. Several libraries for parallel processing such an MPI or OpenMP and Hadoop came with master and node architecture. The design is built for scaling up a cluster. In contrast to other, Hadoop due to amiable interface for developers reflect massive boom of its usage. In

the last years, several libraries for raster and vector data manipulation and analysis for Hadoop and its whole ecosystem of extensions has been developed. Due to the wide range in clusters scalability new room for high-performance computing of spatial data became as reality.

Motivation

Several spatial frameworks for Hadoop has been published in last years and brings the new scope of processing big data in valuable time. Compare to desktop GIS tools which are suited for users with different scale of experience, Hadoop environment, its configuration, deployment and usage of MapReduce is still challenging task. Effective storing of data expressive control is ensured by developed spatial frameworks, where the main functionality can be controlled within the extension such Hive or Pig. Thus more expressive way of interaction is supported and like-SQL approach on the top of Hadoop is provided.

The workflow of processing spatial data using Hadoop and its spatial libraries consists several steps, such as configuration cluster, configuration Hadoop, network and security configuration, exporting spatial data to serialization format, transferring data to the cluster and distributed filesystem, knowledge of Hive data warehouse, querying data using spatial framework, and finally exporting and visualization of the result in GIS. Non automated processing cost time, and without well designed process workflow is not effective.

The Aim and Contribution

The main motivation of the work is to develop framework which simplify the process workflow. Generally, to get familiar with the background of technology, select of appropriate components and develop tools which simplify each step as is possible. Thus, the aim is to design effective work flow and developed its components for processing big data in valuable time. The essential task is to deploy cluster. Under the hood of that is understanding of Hadoop ecosystem and its configuration. In addition, to utilize the benefits of Hadoop it is suitable to deploy cluster using cloud services which ensure fair computation power. Second task of the project, that meet requirements of the process simplification, consists of implementation of the bridge between Hadoop/Hive systems and environment

of GIS. More specifically, according to the thesis assignment framework for GRASS GIS (free open source geo-spatial framework) is implemented.

Background of Related Work

The chapter *Background of related work* is focused on familiarization with theoretical aspects which are essential for fulfiling the aim of the work. Firstly, introduce the Hadoop system, which supports distributed filesystem for manipulation with data and computational framework MapReduce [1] for parallel processing. The next section is focused on review and comparison of spatial frameworks for Hadoop which support geoprocessing tools of big data. For fulfiling the capacity of Hadoop and its extensions is essential to dispose cluster of computer machines. Thus, the last part of theoretical introduction is aimed on cloud services and its challenges.

1 Hadoop system

Hadoop

"The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures."(Apache Hadoop,[22])

History of Hadoop started in 2007 but the roots go to Apache Nutch project.

[1] MapReduce version 2 is also called YARN

Figure 1.1: Hadoop Logo.

At the beginning of October 2003, Apache Nutch[23] has been lunched. Apache Nutch is a web search engine. In the short time (in January 2006) was moved to the new Hadoop sub project. At the same time distributed filesystem called Google filesystem[15], with the specific, permitting, efficient and reliable access to the huge amount of data, has been created. This abstract filesystem, widely called "user level" filesystem, runs as a service that is accessible via APIs and libraries. In 2008, Hadoop made own top-level project at Apache.[24] By this time, Hadoop was being used by many other companies besides Yahoo!, such as Last.fm, Facebook, and the New York Times.

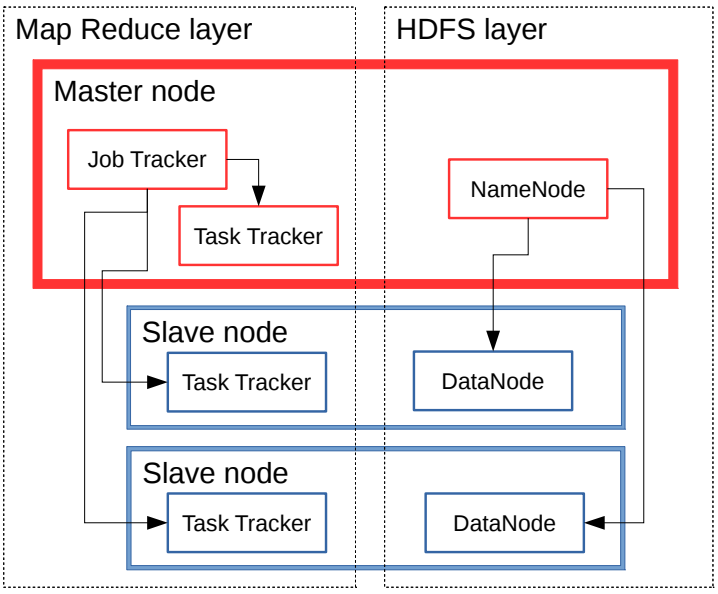


Figure 1.2: Hadoop HDFS and MapReduce layers. Fig. author Matej Krejci

The base Apache Hadoop framework written in Java is composed of the following modules (Hadoop Apache [22]):

- **Hadoop Common** - The common utilities that support the other Hadoop modules;
- **Hadoop Distributed File System (HDFS)** – A distributed filesystem that provides high-throughput access to application data;

- **Hadoop YARN** - A framework for job scheduling and cluster resource management;
- **Hadoop MapReduce** - an implementation of the MapReduce programming model for large scale data processing. Since version 2 the MapReduce is YARN-based system for parallel processing of large data sets.

All the modules in Hadoop are designed with an expectation that hardware failures are common and thus must be automatically managed by the architecture of software.

The Hadoop framework is written in Java with some native code in C and command line utilities written as shell-scripts. Moreover, for development *Map* and *Reduce* parts with using "Hadoop Streaming" any programming language can be apply. Beside main modules, there are many Hadoop extensions for cluster management, data access and helpers for storing data in HDFS. As suitable example for using spatial libraries is Apache Hive, which expose higher level user interfaces and provide SQL-like query syntax.

There are two main components of the Apache Hadoop 1.x: (1) the Hadoop Distributed File System (HDFS) and (2) the MapReduce parallel processing framework. For Hadoop 2.x have been developed new MapReduce framework named YARN, which handle better the week point of MapReduce v.1.

1.1 HDFS: Hadoop Distributed File System

As the main source for describing HDFS is used the official documentation (*hadoop.apache.org* [22]) and *Hadoop: The Definitive Guide*[2]

Hadoop comes with a filesystem and since it manages the storage of files on several machines, it is called Hadoop Distributed filesystem (HDFS). HDFS is designed for handling very large files with streaming data access. As example suits well CSV, JSON or any data structure which can be serialized. [25] In HDFS, large files are broken down into smaller blocks (128MB, by default) which are stored as independent units. The architecture of HDFS is a highly fault-tolerant and provides high permeability for access. A short overview of main characteristics of HDFS design is described in (Hadoop: The Definitive Guide[2]) as five features: **Very large files** - With relevant hardware data of amounts petabytes can be accessed on Hadoop clusters effectively. **Streaming data**

access - Efficient data processing is based on read once and copied many times pattern. **Commodity hardware** is suitable for running Hadoop. At the end, this fact helped with the decision to invest value of money to the development of Hadoop instead of operating on expensive and highly reliable hardware. **Low-latency data access** - HDFS is not suitable for performing low-latency access to data. Primary, HDFS is optimized for delivering a high throughput of data. **Lots of small files** allows to read and operate over more files at the same time. Limitation of number of stored directories, files and block in filesystem is limited by *NameNode* memory.

HDFS is based on traditional hierarchical file model. As in other filesystems, HDFS allows basic file operations: read, write, rename, move etc. However it doesn't support hard and soft links.

```
#copy file in distributed file system
$ hadoop fs -cp /user/hadoop/file1.csv /user/tmp/file1.csv
```

The example above shown copying file between HDFS folders.

1.1.1 Organization of Data

Similarly to a common filesystem, HDFS is based on the disk blocks as well. The traditional filesystem is based on blocks which define the minimal size of the amount of data to read and write. HDFS blocks have the similar concept based on blocks, but the minimal unit is larger. The size of the block is 128 MB by default. Blocks are broken and distributed over disks on the cluster like blocks over a single disk in the filesystem. The ideal size of stored files is the same as the size of the block. With increasing the block size time cost for computing increases as well. On the other hand, a large number of blocks is expensive for the preparation of files. The important task is to find the optimized ratio between the data preparation and the computational time by setting suitable blocks size. The idea of block abstraction helps to bring several benefits. The abstraction design allows to store larger file than the physical disk unit, even the fact that for Hadoop it is unusual.

The second benefit of fixed size is simplifying storage management. Even file size not fully fills the filesystem block, the block is not used for other files. Obviously, it makes easy to hold calculating of size discernibility and eliminating metadata concerns (not necessary to store metadata of tree in data blocks, even in the same system) Furthermore, the blocks

are suitable for replication and for providing fault tolerance. Protection against corrupted blocks, disks or machines is based on replication of block over a cluster. Moreover, this approach ensures the integrity of HDFS checksums when writing as well as reading data.

Replication selection process is suited to minimize bandwidth consumption over a cluster. To minimize read latency, HDFS is primary try to read the closest replica to the reader. Priority is the same for rack as for node reader. [25]

Rack Awareness on Hadoop serves the maximum potential of production, it means that it knows the network topology. The purpose of rack-aware replication is ensured by policy component. It controls data reliability, availability, and network bandwidth. For a multi-rack cluster it is suitable to map and link nodes to a rack[26]. This can be ensured manually or using program for mapping hierarchy of IP addresses. The priority of transfers is a size of bandwidth availability. Within transfers where there is more bandwidth available as compared to off-rack transfers for MapReduce jobs on a node. By theory, the better network bandwidth is between machines in the same rack, due to hardware specifics.

1.2 Hadoop Server Components

Machines for Hadoop deployment count several server components such as Master Servers and Slave Servers and for the access Client Machines. Hadoop consists five main components:

- NameNode
- DataNode
- Secondary NameNode
- JobTracker
- TaskTracker

These are basically daemons or programs that run on different physical servers. The figure 1.3 describes the main components and connections between them.

Client computers have installed Hadoop with all the configuration but they are not hosted on Master or Slave server. The Clients task is to load data into the cluster, submit jobs and get the result or looking for jobs when finished.

1.2.1 NameNode, DataNode and SecondaryNameNode

Hadoop comes with cluster architecture based on master (*NameNode*) and slave (*DataNode*) design pattern. The *NameNode* handles metadata of abstract filesystem (namespace), which include information about the structure of files and directories in the tree. It does not hold any cluster data itself. The *NameNode* only knows blocks which make up a file and where those blocks are located in the cluster. Information about filesystem is stored on local disk in two files: the namespace image and edit log file. The *NameNode* knows all locations of *DataNodes* blocks where data are stored. It also provides the primary user interface to access HDFS. By design *NameNode* is a single point of failure and should be never overloaded and must be the most reliable node of the cluster. Without *NameNode*, HDFS is totally unserviceable. In recent Hadoop releases, there is also a

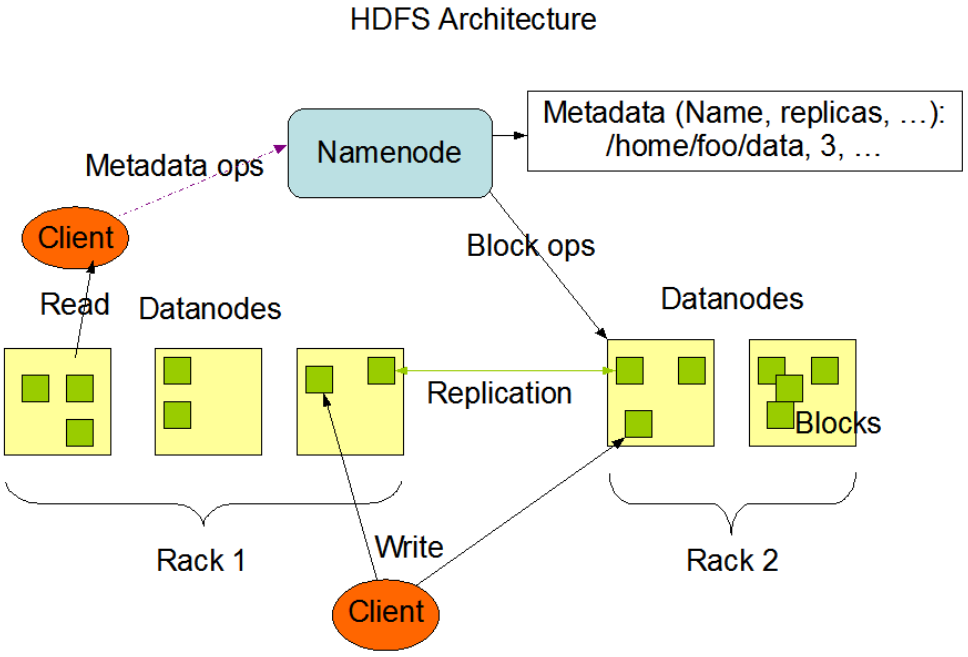


Figure 1.3: HDFS architecture [2]

backup node - *SecondaryNameNode*, always up to date with latest (per 1 hour by default) *NameNode* status. It receives all the operations done by *NameNode* and stores them in local memory. This permits to have the latest, up to date namespace status, when *NameNode* fails.

[2]Figure source: `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`

As has been mentioned, the blocks of a file are independently stored in nodes, which are called *DataNodes*. Each *DataNode* in the cluster makes registration process to the *NameNode* during start. Besides that, each *DataNode* informs *NameNode* about blocks availability by sending a block report. Block reports are sent periodically or when a change event happens. Moreover, every *DataNode* sends *relevant* messages to the *NameNode* to confirm that it remains operational and that the data is safe and available. If a *DataNode* stops operating, the error mechanisms designed to defend the failure and data loss maintain the availability of the block. *Relevant* messages also hold information, which allows *NameNode* to run the cluster efficiently e.g. load balancing. One important concept of design is that *NameNode* never directly calls data.

1.2.2 JobTracker and TaskTracker

Above the filesystems is the MapReduce layer, in other words, MapReduce engine, which consists of JobTracker, which ensures client applications to submit MapReduce jobs. The JobTracker handle work out to available TaskTracker nodes in the cluster, aiming to keep the jobs close to a data block. Thus, JobTracker and TaskTracker are two types of nodes for managing the execution process of task. Client submits MapReduce job to the JobTracker to process a particular file. JobTracker choose the DataNodes which store block with the desired file by asking the NameNode where metadata of filesystem are stored. JobTracker appends tasks to TaskTracker according to the information obtained from NameNode and monitors the status of each task.

1.3 HDFS access

Several ways how to use HDFS interactively are available. In analogy to filesystem, the main task is manipulation with data, access rules and additional to load data to distributed filesystem. As in standard filesystem, the permission model is provided in HDFS as well. In paragraphs below there are described accesses HDFS methods and permission models.

Permission model The permission model of HDFS has several levels. The level of files and directory is similar to POSIX model. Each file and directory are associated with an owner and a group. Each item in HDFS, such a file and directory, has separate permission.

Thus, individual permissions for the user, for other users that are member of group and the rest of users can be assigned. Permission is r for reading, w for writing and x not for execution, but for permission to access a child of the directory.[18]

There are two different models for checking the user's identity.

- **Simple** In this model is an identity of user derived from host operating system.
- **Kerberos** In Kerberos model, the identification process is managed by 'tickets' based on Kerberos credentials system which allow nodes communication over a non-secure network to prove their identity to one another in a secure manner.[3]

The identity mechanisms are extrinsic to HDFS itself. Thus, within HDFS there is no handler for creating user identities, establishing groups, or processing user credentials.

Access mechanisms The basic mechanism for interaction with HDFS is *hadoop fs* command line tool *bin/hadoop fs <args>*. All fs shell commands take path URIs as arguments. Most of the commands in fs shell are derived from Unix commands.

The other mechanism for accessing HDFS is through application programming interfaces such a APIs: Native Java API, which has a base class *org.apache.hadoop.fs .FileSystem*; C API that works through the *libHDFS* library, and there's a header file, *hdfs.h* which has information on the API calls.

In addition, interaction according standard REST API is available. REST API is an architectural style consisting set of rules and constraints applied to components within distributed systems. The WebHDFS REST API components fully cover the standard fs tool. The API consists four main groups of HTTP requests, such as: HTTP GET for fetching information, HTTP PUT for manipulation request, HTTP POST for append or concat file and HTTP DELETE for deleting files and directories.[20] The Unix tool *curl* allows accessing API from the Unix terminal.

```
$ curl -i -X POST -T <LOCAL_FILE> "http://<DATANODE>:<PORT>/webhdfs/v1/<P..."
....

HTTP/1.1 200 OK
Content-Length: 0
```

The example above demonstrates submit HTTP POST request using the URL in the Location header with the file data to be appended. The client receives a response with zero content length.

1.4 Parallel computing - MapReduce

The *MapReduce* is a programming model for processing and generating large data sets. The *MapReduce* abstraction is inspired by the Map and Reduce functions, which are commonly found in functional programming languages, such as LISP [16]. Users can easily express their computation as a series of Map and Reduce functions. The Map function processes a series of < *key, value* > pairs to generate a set of intermediate < *key, value* > pairs.

$$\text{Map(keyA, valueA)} \rightarrow \text{list (keyB, valueB)}$$

Reduce function aggregates all intermediate values that associate to the same intermediate key to produce the final output, also in the form of < key, value > pairs

$$\text{Reduce(keyB, list(valueB))} \rightarrow \text{list (key C, valueC)}$$

Thus the *MapReduce* framework transforms a list of (key, value) pairs into a list of values.

As MapReduce demonstration the data from cellular microwaves links (MWL) are used. After the data are serialized and stored in text files each file represents captured data for 24 hours time interval. The size amount of weakly data is relatively small (100Mb) which suits well to the default block size(128Mb). The aim is to compute mean of differences between transmitted and received signal for each link in the period between 2014-07-07 and 2015-07-07.

Below is the sample of data stored as CSV.

```
linkid;data;rx;tx
324;"2014-07-07 11:14:56.552";-48.9;10
256;"2014-07-07 11:14:59.703";-99.9;7
...
324;"2015-07-07 17:10:56.578";-50.1;7;
256;"2015-07-07 17:10:56.484";-85.3;10;
```

The text lines above are presented to map functions as key-values pairs. The map function extracts data from date 2014-07-07 and creates pairs link:(rx-tx)

```
{324:-58.9}
{256:-106.9}
...
{324:-57.1}
{256:-95.3}
```

The output is processed by MapReduce framework before is being sent to reduce function. Mentioned process sorts pairs by key as shown bellow.

```
{324:[-58.9,-57.1]}
{256:[-106.9,-95.3]}
```

Finally reduce program iterates over the list of values for each link and compute mean. The final results are stored in Hadoop filesystem.

2 Spatial Processing in Parallel

2.1 Introduction to Spatial MapReduce Query

Figure 1.4 shows a simple example of distributed spatial query processing. The main idea grows up from the basis of MapReduce. On the figure, the dataset is partitioned into four tiles. After query job is started, each tile is analyzed separately using Map (M1-M4) function. For demonstration, query analyzes intersection of two datasets (red, blue) on each tile. Thus, relation between objects from the different dataset is detected. The result is done by Reduce (R) function, where results from each map phase are processed.

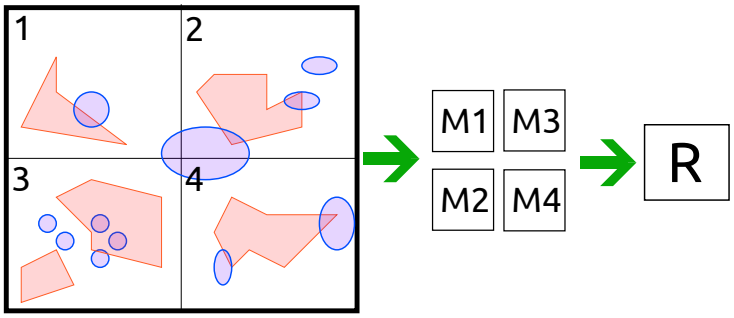


Figure 1.4: Detection of spatial relations using Map (*M1-M4*) and Reduce (*R*) functions

The motivation for spatial processing in parallel is querying data in reasonable time. Firstly, it is essential to introduce characteristics of queries for solving different tasks. In HadoopGIS (F. Aji, F. Wang, et al.[11]) five major query cases of spatial data is defined: (1) *Feature Aggregation queries* which doesn't fall into spatial queries, however, they are evenly important for spatial frameworks as others. As a common example of Feature Aggregation query, a function for finding mean values of attributes. (2) *Fundamental Spatial*

Queries covers groups of tasks like: including point based queries, containment queries, and spatial joins. (3) Next groups *Complex Spatial Queries* includes more challenging task. A query which solves advanced Spatial Join: spatial mismatching or overlay; or neighbor query. (4) Integrated spatial and feature queries, which combine query from more categories, such feature aggregation queries in a selected spatial regions. Finally (5) *Global Spatial Pattern Queries*, for example, queries on finding high-density regions, or queries to find directional patterns of spatial objects.

Anyway, classification of a query to five groups is based on characteristics of a task. But, as in RDBMS and also in distributed databases, the most challenging is to solve cost-intensive query. Join Queries and Nearest Neighbour queries can be classified as cost-intensive to compare to the rest. That fact points these query as the interesting and most challenging task for a scientist and developers. Naturally it affects the direction of the most of academic works and developments.

Speeding up a process of solving *Spatial Join* which is classified as classic GIS problem is a motivation for the parallel processing of spatial data. Spatial Join is an operation used to combine two or more datasets with respect to a spatial relationship. Detection of relations between spatial objects using serial (used in RDBMS) computing approach started to be limited by its design. On the scene comes solutions based on parallel and distributed models. In the past, many distributed solutions as OpenMP[9] have been presented, for instance, Intel TBB and Bulk Synchronous Parallel[10]. Because of their complexity, they have been used insignificantly to compare with Hadoop framework. Already introduced MapReduce parallel model opens doors for developing efficient spatial query engine with less developing investments. In the last years, several Spatial MapReduce projects came up on the scene. *SpatialHadoop*[12], *HadoopGIS*[11] and *ESRI Spatial Framework for Hadoop*[19] are three open source libraries that are designed to process large scale spatial data within the integration of Hadoop. All three systems come with significantly different design of implementation but they all are based on MapReduce framework from Hadoop environment and basically all project have the same goal. To realize such systems, it is essential to identify time-consuming spatial query components, break them down into small tasks, and process these tasks in parallel. Design of the first two systems, *SpatialHadoop* and *HadoopGIS* extensions are well described in publications. Background of ESRI library is not described in an available literature, on the other hand, the user documentation seems to be complex and described in detail. ESRI laboratory

published article[13] about QuadTree indexing, which is used for building indexes in their *ESRI Spatial Framework for Hadoop*. In the section below are described fundamentals of spatial processing in parallel and discussed the comparison of solutions of developed and published frameworks is provided. The main source for features analyses of spatial frameworks design are already mentioned literature sources (*SpatialHadoop*[12], *HadoopGIS*[11] and *ESRI Spatial Framework for Hadoop*[19]).

2.2 Spatial Join

Assume two sets of multi-dimensional object in *Euclidean space*. Relation of spatial join between sets R and S can be defined[7]:

$$R \bowtie_{pred} S = \{(r, s|r) \in R, s \in S, pred(r, s) is \ \ true)\}$$

where $\bowtie_{pred}$ is a spatial predicate for the relationship of two spatial objects. *Spatial join* finds all pairs of object which satisfying a spatial relation between given objects. For further explanation of the spatial join concept assume that objects s from set S and analogically r from R are rectangles. Intersection operation between each rectangle s and r will report set R of intersected r rectangles.

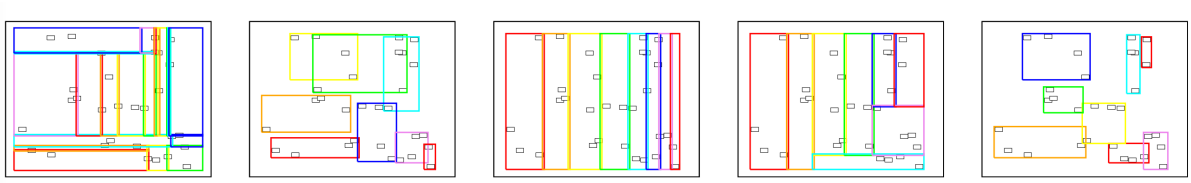
Solution of described *spatial join* is trivial. General spatial join problem has been extended by complex spatial join, known as *spatial overlay join*[4]: (a) The set of objects can be another character than a rectangle, such as point, segments or polygon. (b) The dimension of a set can be more than three. (c) The relationship between pairs of objects may be any relationship between objects which includes spatial elements, such as intersections, nearness, enclosure, or a directional relation.

To speed up spatial join query different ways based on filtering are commonly used. Typical trivial solutions are reached by usage of Minimal Bounding Box (MBB) as a first filter for defining the subset of candidate object satisfying a spatial predicate. For defining MBBs for sets of points Convex Hull algorithm can be used. To speed up construction computation of Convex Hull based on heuristic [8] is more efficient. In Spatial Join technique [4] different methods of spatial objects filtering according the specific requirements are used. For large dataset stored in GIS, the filtering is essential. Objects such a polygon or point cloud can reach millions or more features. Boolean vectors operation on detecting relation without filter stage can be extremely expensive in a meaning of I/O performance.

Usually, in the first stage there is filtering data; spatial objects are read from hard drive which is sufficient for computation of MBRs, then created MBRs are stored in memory and the spatial join test is performed faster.

2.2.1 Spatial Partitioning

Data partitioning is a powerful mechanism for improving an efficiency of data management systems, and it is a standard attribute in modern database systems. Partitioning data into smaller units provide room for query processing in parallel and further improved performance. In addition, with proper partition scheme, I/O operations may be substantially reduced by scanning only a few sections that contain relevant information to answer a question. Within the architecture of Hadoop, the important step and the first one is defining a suitable format for storing data which influences speed of their access and query execution. In general, partitioning produces sub-datasets based in smaller regions-tile. There are two main reasons for the partitioning of spatial data. The first one is to avoid tiles with high density. This is mainly due to the potential high skew[6] of data which could imbalance workers in a cluster environment. Another aspect is to properly handle boundary of intersecting objects. As MapReduce provides custom scheduling for balancing tasks the problem of load imbalance can be partially mitigated to a task of schedule planning. In work (Effective Spatial Data Partitioning for Scalable Query Processing [5]) there has been introduced and compared six methods of spatial data partitioning for parallel processing.



(a) BSP (b) HC (c) SLC (d) BOS (e) STR

Figure 1.5: Spatial partitions generated by different algorithms BSP: Binary split partitioning, FG: Fixed grid partitioning, SLC: Strip partitioning, BOS: Boundary optimized strip partitioning, STR: sort-tile-recursive partitioning, HC: Hilbert curve partitioning. Source of fig: [5]

Spatial data skew is common challenging problem in a spatial application. As example [11] assume dataset of $1000 \cdot 1000$ tiles. The maximum count of the object is 10000 objects, but the average count is 1020. This fact growing from the real situation where a high density of objects are e.g. in cities and less dense in farmlands. In parallel spatial processing where a distribution of processes is based on tiles, the data skew can significantly increase the response time.

Boundary object Spatial partitioning generates boundary objects that cross multiple partitions. Thus, it influences the independent relation of each partition. In a usual case, a spatial object has complex boundary and extent. Even more, from real experience is evident that in spatial datasets object can cross multiple partitions. The problem is solved with different methods; replicating spatial object to multiple partitions and filtering duplicates during query process or another approach, creation sectors with putting weight on minimizing boundary objects.

Developed spatial libraries comes with different approaches for solving, partitioning, boundary overlay and data skew:

- **HadoopGIS** in partitioning stage is focused on breaking high-density tiles into smaller ones with using recursive partitioning. Firstly, they provide two-dimensional data partitioning and generates a set of tiles. These preprocessed tiles are a source for query tasks. The preprocessing is done with the using of distributed computing. Week point of tile-based approach is non-adaptable algorithm on non-uniformly distributed data. Because that, critical problem of *data skew* with these spatial partitioning is known. *HadoopGIS* ensures this problem by cutting high-density tiles into small ones, thus it use recursive partitioning approach. The maximal and a minimal number of objects per tile is defined for controlling number of objects per tile . Splitting recursion finds optimal direction(x or y) for creating half-sized tiles which fulfil the thresholds by a number of objects in each half. Taking into account default HDFS blocks size (128MB), the final tiles are not stored in small files.
- **SpatialHadoop** implemented the solution based on the same tiles idea as HadoopGIS, but few features are significantly different. The main difference is storing partitions (tiles) in HDFS blocks instead of using big batch file. Three main characteristics

arise from that fact. SpatialHadoop counts a size of HDFS block and the size of tiles should fit that size. It avoids data skew of spatial datasets. Secondly, a design of partitioning method ensures spatial locality; a spatially close object is assigned to the same partition. The last feature attempts to balancing the size of a partition. In an ideal case, all partition are the same size. The number of partitions n is computed as $n = S(1 + \alpha)/B$, where S is the size of input file, B is the HDFS block size and α is an overhead ratio (0.2 by default). Next step is the definition of boundaries by of partitions. Boundaries are represented by rectangles. Thus, a skew of data is not considered. The output of this step is set of rectangles representing boundaries of partition. Together it represents cover of whole space domain. The result is input for initialization of physical partitioning procedure. The method solving the problem of an object with spatial extents (e.g. polygon) which overlaps more than one partitions. Determination of solution is based on selected indexing method. If some of them assign object to all overlapping partitions and the others find the best matching partition. Replicated records are managed later by query processor. Finally, for each record from partition the map function write the pair <partition,record> which are grouped by partitions and sent to reduce function for the next task- indexing phase.

2.2.2 Spatial Indexing

One essential requirement for spatial queries is a fast response. In general, the filtering and partitioning of spatial objects are widely related to indexing. Spatial indexing helps to avoid of sequential browsing. In other words to avoid full table scanning. Generally, in RDBMS for creation index over specified column or multiple columns is available multiple indexing methods like btree, hash, gist, and gin. On spatial indexing problem in RDBMS have been done many scientific works and different algorithms have been shown. Suitability of different indexing methods is linked to the characteristics and distribution of data. Adaptation of serial indexing method to parallel processing points non-trivial challenge. RDBMS has the ability to use information from multiple indexes to determine how best to search for the records that satisfy all query criteria. A NoSQL key-value store, in contrast, has only a single index. That is built atop the constraint that all records are ordered lexicographically. Traditional Spatial Indexes from the field of RDBMS e.g. Grid file or R-tree are not suitable for parallel processing.

The architecture of traditional indexing method is not designed for effective usage on Hadoop, where developers implement indexing in a procedural way and the execution runs on multiple threads. Since Hadoop is based on MapReduce layer to implement sequential construction instead of incremental one is necessary. MapReduce is a scan based data processing framework which does not utilize any form of a disk-based index and the performance is limited as the input has to be scanned. The design of spatial indexing methods must be hand to hand with filtering approach and selected partitioning pattern.

Developed spatial libraries comes with different approaches that solve spatial indexing challenge:

- **SpatialHadoop** indexing model is composed from three phases. The first phase, partitioning is already described in section 2.2.1.

  The purpose of the second phase is to build *local index* for each physical partition of data. In contrast to global index, the local index is built for each partition separately. In addition, each local index is stored in one HDFS block. It ensures that Hadoop uses load balancer for relocating blocks across machine. Additionally, it allows a spatial operation to access local indexes where each local index is processed in one map task.

  Last phase, building the *global index* provide index structure of all partitions. As next step, all local indexes to one file which represent the final index of spatial data are concatenated. This process provides using MapReduce layer. After this part is done, NameNode creates a global index and store it in memory. The index is based on dictionary- key and value.

```
-179.3248215,-54.934357,6.9290401,71.2885321,part-00000_data_00001
-171.773529,-54.81145,6.9261512,65.1480999,part-00000_data_00001_1
6.9225032,-46.44586,179.3801209,78.0657531,part-00000_data_00002_2
```

  Key is represented by HDFS block and rectangular boundaries as value. This index keeps all the time in the main memory. That ensure faster access. In critic situation when fail of master or restart of a cluster is possible to rebuild index from rectangular boundaries of the file blocks. For accessing the master index it is not necessary to parse it yourself. Implementation offers API which retrieves the global index as an object. SpatialHadoop offers tree methods for building indexes.

The simple one, **Grid Index** is a flat and partitions of data according to a grid such that records overlapping each grid cell are stored in one file block as a single partition. Grid index is suitable for uniformly distributed data. Otherwise, the data screw is critical by the principle of MapReduce.

Besides grid index, SpatialHadoop framework offer building index based on method **R-tree** and **R+-tree**. Fig 1.6 (b) shows results of R-tree algorithm.

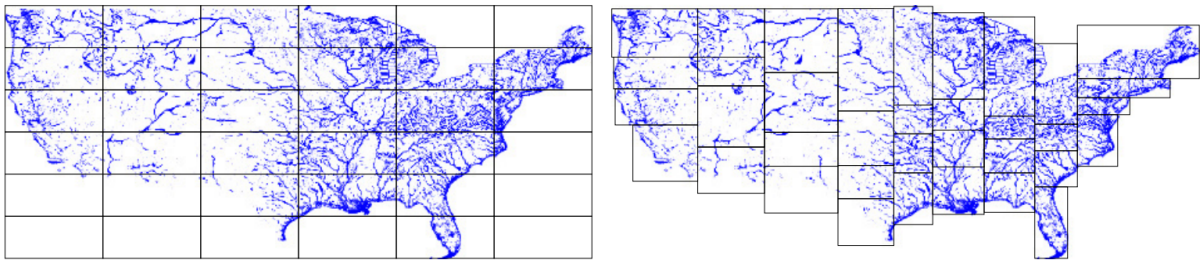


Figure 1.6: (a) Grid Partitioning (b) R-tree Partitioning.[3]

- **HadoopGIS** indexing is based on hierarchical space partitioning and MBB based region filtering. Similarly, the Grid Index design of SpatialHadoop is built from combination of local indexes and global index. Recursively each tile can be further partitioned into even smaller regions. In contrast to SpatialHadoop, HadoopGIS does not pre-generate indexes into files. Finally, HadoopGIS supports uniform grid index, which is efficiently applicable only in the rare case of uniform data distribution.

2.2.3 Spatial Operations

Three mentioned spatial processing frameworks (section *Spatial Join* 2.2) for Hadoop show different design for storing and accessing data. Similarly, the interface for configuration, data management and query data varies.

Operation on *HadoopGIS* and *ESRI Spatial framework for Hadoop* are based on like-SQL layer. Particularly on the top of spatial MapReduce library there is implemented data warehouse Hive based on like SQL commands. *SpatialHadoop* supports spatial extension Pigeon a high level SQL-like language which provides OGC-compliant spatial data types

[3]Source: SpatialHadoop: A MapReduce Framework for Spatial Data; Eldawy A., Mokbel,F.M [12]

and operations making it easier to adopt by users. It makes the program simpler and more expressive as it uses spatial data types (e.g. POINT and RECTANGLE) and spatial functions (e.g. ST_Overlaps, ST_Contain). The implementation of spatial operators are designed to be used as defined functions (UDFs) which are seamless to integrate with existing non-spatial operations in Hive or Pig for SpatialHadoop. The spatial operators of all three libraries fulfil implementation rules according to OGC standard.

framework	HadoopGIS	SpatialHadoop	Spatial framework for Hadoop(ESRI)
integration with extension	HiveSP	Pigeon(spatial extension for Apache Pig)	Hive
build-in data types	point, polygon, pox and lineString	point, rectangle and polygon	inherit from ESRI Java Geometry Library: Point, MultiPoint, Polyline, Polygon, Envelope. Also includes OGC Wrappers.
input data	CSV	CSV, custom data types	JSON, GeoJSON, text, WKB
user-defined data types	-	- for Pigeon + SpatialHadoop	-
visualisation	-	+*HadoopViz	*Geoprocessing Tools for Hadoop (ArcMap)
index	uniform grid index	R-tree, R+-tree, Grid index	QuadTree
accessible index	-	+	-

Table 1.1: Overview of spatial frameworks for Hadoop

2.3 Summary

SpatialHadoop, HadoopGIS and ESRI Spatial Framework for Hadoop are three open source systems that are designed to process large scale spatial data on Hadoop. Although these features of the implementation of each framework are different, the overall design is suited to Map and Reduce function principals. Spatial cross join is well suited to parallelizable and fit to MapReduce.

All three spatial frameworks come with indexing, where SpatialHadoop is the most customizable. The design of SpatialHadoop implementation is open to extending available indexes. ESRI and SpatialHadoop offer more advanced indexing methods, such as QuadTree (ESRI) and E-tree (SpatialHadoop). One of the main weakness of HadoopGIS is an limited indexing support. It disposes only by grid index method which can be effectively used only for uniformly distributed data.

To compare with others, SpatialHadoop comes with the different MapReduce approach for solving cross-spatial join. In SpatialHadoop, both sides in a spatial join are partitioned and implemented as only Map job. SpatialHadoop makes pairs from spatially overlapping partitions, which are finally assigned to Map tasks for parallel and distributed execution.

In addition, SpatialHadoop implements filtering using MapReduce as a preprocessing step. HadoopGIS as the first step joins both datasets, reorders and assigns its data to the same HDFS block. Accessing data are provided by key-value pairs. Partitions are represented by keys and item as values.

When decision which framework is suitable for a specific application must be made, the key factor can be access of data or programming language choice. While SpatialHadoop uses binary format for representation data in memory and disk, HadoopGIS uses Hadoop streams. Thus, HadoopGIS representing all side results as text and access of data is only sequential. In contrast, SpatialHadoop binary approach allows accessing data randomly. These facts make SpatialHadoop more efficient since parsing text are an expensive task. On the other hand, HadoopGIS allows writing MapReduce function in different languages than Java which can be crucial factor.

Generally, all frameworks allow customization on the level of developing MapReduce functions. As ESRI framework allows interaction with core libraries, the library is more focused or GIS users than developers. ESRI covers the wide portfolio of built-in spatial operators to compare SpatialHadoop framework, and at least significantly more than

HadoopGIS. On the other hand, the leader for customization and development of additional functionality is SpatialHadoop.

3 Google Cloud Platform

Cloud in computer technology means service which provide capability of storage and computation engine within data center . Services allow to developers work efficiently with requisite resources which are necessary for an application to work.

Google Cloud Platform supports building, testing and deploying an application on highly-scalable infrastructure. Developing products using the platform is based on building blocks of particular functionality. Advantages of the platform design is a quick development of desired product by combining separate but compatible services. Portfolio of services is parted into six main categories which are overview in table 1.2 [27].

The user interface offers three different ways for managing services of Google Cloud. Web-based interface, command-line interface and programming API. With web interface user can quickly get overview and a better understanding of options over each service. On the other hand, the range of configuration possibilities is limited compared with the rest interfaces. *Google Cloud SDK* is a package of tools for managing resources and application on Google Cloud Platform. Package includes *gcloud* commandline tool for Google Compute Engine resources and *gsutil* for working with Cloud Storage.

Command-line interface called *gcloud* is for managing Google Compute Engines resources in faster way. It uses concept of configurations for managing different accounts on Google Cloud. Interface of *gcloud* is suited for handling services of Google cloud platform using command line. For example, deploying VM, handling containers, configuring network and the other.

Configuration allows to set default general variables: account, project, default region and zone, proxy etc. To set up configuration profile: default zone, region is appropriate. The variables can be set by *gcloud* or with using local variables. Below is an example of deploying VM instance with default configuration. The instance is accessible by *gcloud* ssh tool (e.g. instance run in zone asia-east1). Access can be also provided by standard ssh protocol.

Group	Framework	Description
Compute	Compute engine	large-scale workloads on virtual machines
	Preemptible VMs	short-lived instances for batch and fault-tolerant jobs
	Custom Machine Types	customized deployment of virtual machine
	App Engine	engine for building scalable web app and mobile backends
	Container engine	powerful cluster manager for running Docker containers
Storage	Cloud storage*	simple, cost effective data storage
	Cloud storage Nearline	highly-durable storage for data archiving and online backup
	Cloud SQL	storing data using relational MySQL database
	Datastore 3.3	scalable, NoSQL database for non-relational data
	Bigtable	fast, scalable NoSQL database service
Networking	Cloud networking	load balancing, VPN, DNS,
Big Data	BigQuery	real time analysis(100,000 rows)of data per second
	DataFlow	real-time data processing for batch and stream data processing
	Dataproc 3.4	management for Spark and Hadoop services. Quick deployment
	DataLab	analysis and visualisation of large-scaled data
	Pub/Sub	real-time messaging service for sending up to 1 milion messages per second
Services	Cloud Endpoinds	API for access to backend servers for mobile platforms
	Translate API	allows to develop apps for translating languages programatically
Management	Cloud monitoring	monitoring performance and availability of cloud apps
	Cloud deployment manager	manager for repeatable deployment using templates
	Container registry	private Docker image storage
	Cloud logging	Mannager for log data and engine for debug system issue. Supports Google App Engine and Google Compute Engine.

Table 1.2: Overview of Google Cloud platform

```
$ gcloud compute instances create my-instance
...
$ gcloud compute ssh my-instance --zone asia-east1
```

gcloud dataproc 3.4 tools support deployment of clusters with limited configuration compare to *bdutil*. For deploying cluster with distributed filesystem, like Hadoop and Spark, is can be advantage to use *bdutil* 3.4.1 tools.

Google Cloud Client Libraries are for programmatic access to Google Cloude Platform services. For development applications linked to cloud services better language integration is provided. In addition, security handling and easy general access as well. Libraries are currently available in five languages: Go, Java, Note.js, Python and Ruby. Below is shown example of code written with Python API library. The sample of code allows to initialize instance of Cloud Dataproc and get JSON object. The object representing list of clusters.

```
from oauth2client.client import GoogleCredentials
project = 'my-test-project'
region = 'global'
credentials = GoogleCredentials.get_application_default()
result = dataproc.projects().regions().clusters().list(
projectId=project,
region = region).execute()
```

3.1 Geography and Regions

Google Cloud data centers are available across three *locations*: North America, Europe, and Asia. Each location includes *regions* and *zones*. The choice of place, where service is running is on developer. The statistics data and live stream of center resources, latency, availability, durability are available. Thereunto, using API for manipulation with resources problematically is possible.

Hierarchy Design of network is suited for maintaining of custom services efficiently. The hierarchical model is build on separated levels of failure. The room for distributing resources across multiple regions or zones is supported as well. Main abstract unit is *location*. Each location consists *regions*. To handle availability, locations are independent to

each other. It make network latency under 5m (on the 95th percentile) between locations. Lower level of hierarchy, thus subset are *zones*. As well as location, zones are independent to each other. Location consists of zones which should be classified as single point of failure. Thus, to build up fault-tolerant application is necessary to deploy it over different zones. Furthermore, choice of region according to geographic location of accessing points is essential. In addition, to eliminate fail of service on the level of *region*, Google suggests to have a backup plan for migration to another one. Fully qualified address of zone for API access is <region>-<zone> e.g. *europe-west1-d*.

```
## view list of available zones.
$ gcloud compute zones list
NAME            REGION        STATUS NEXT_MAINTENANCE
asia-east1-c    asia-east1    UP
..

## check available regions.
$ gcloud compute regions list
NAME          CPUS          DISKS_GB     ADDRESSES RESERVED_ADDRESSES STATUS
asia-east1       0.00/8.00       0/2048       0/23       0/1            UP
..

## get information about desired zone e.g asia-east1.
$ gcloud compute regions describe asia-east1
creationTimestamp: '2014-05-30T18:35:16.514-07:00'
description: asia-east1
id: '1220'
kind: compute#region
name: asia-east1
quotas:
- limit: 8.0
metric: CPUS
usage: 0.0
..
```

3.2 Identity and Access Management

Cloud Identity and Access Management (Cloud IAM) provide way for handling privacy and permissions for *Google Cloud* services. Currently, alpha or beta versions are supported for nine services over Google Cloud portfolio, includes *Google Cloud Storage*. IAM design is based on three main pillars, such *Roles*, *Policy* and *Resources*. The identity of a user is handled by four different authentication methods such a Google account, Service Account, Google Group and Google Apps domain.

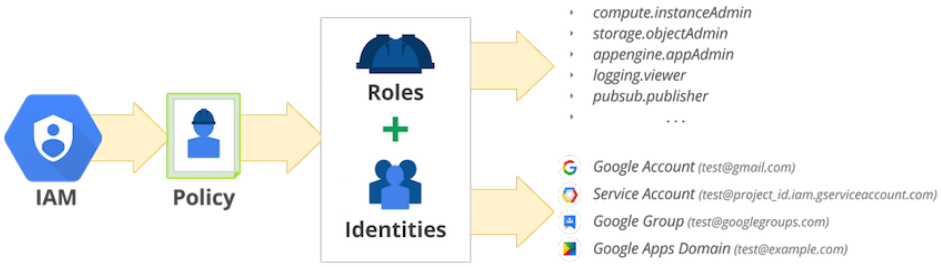
Resources allows grant access to users for Cloud Platform services and its resources. To allow permissions for using given service is declared by syntax <service>.<resource>.<verb>, for example *pubsub.topics.publish*.

Roles A role represents the collection of permissions. Two kinds of roles are available, such as Primitive roles and Curated roles. The first mentioned are concentric roles where *Owner role* can also edit and *Edit role* includes *Read role*. This approach is applied e.g. in Cloud Storage as default. Second, curated roles are new and allow more advanced policy management. Currently (28.3.2016), *Curated roles* (1.7, (a) are in alpha or beta development version and are not suggested to use for production use. However, these Curated roles provide additional granular access to specific services from Google Cloud portfolio and prevent unwanted access to other resources.

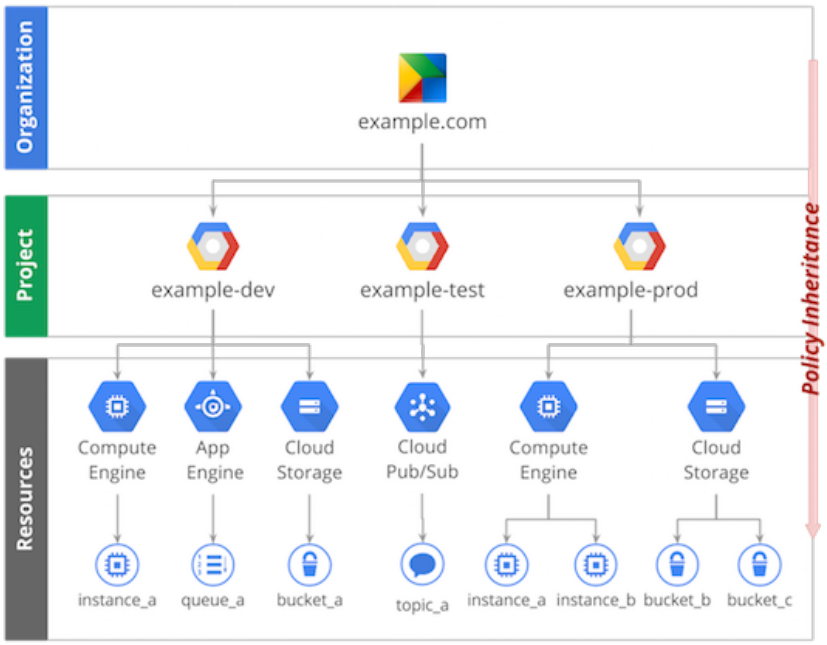
Role	Description	Permissions
roles/storage.objectCreator	Allows users to create objects. Does not give permission to delete or overwrite objects.	storage.objects.create
roles/storage.objectViewer	Grants access to view objects and their metadata, excluding ACLs.	storage.objects.get storage.objects.list
roles/storage.objectAdmin	Grants full control of objects.	storage.objects.*
roles/storage.admin	Grants full control of objects and buckets.	storage.buckets.* storage.objects.*

Table 1.3: Cloud Storage Roles covered by IAM

Policy of Cloud IAM can grant roles to users. It is designed as a collection of statements that specify who has what type of access. The policy is linked to resources and control it's accessed. In Figure 1.7, (b) is shown policy principle based on the relation between the collection of statements IAM and Resources of Google Platform. Resources of Google Platform are organized hierarchically. Figure 1.7, (c) demonstrated example of the nodes in hierarchy. *Organization* represent the root of the hierarchy and each child has only one parent. Inheritance of hierarchy handle sets and subsets of abstract classes, such as Organization, Project and Resources.



(a) Cloud IAM Policy and Roles management



(b) Policy hierarchy

Figure 1.7: Concepts of Cloud Identity and Access Management. Source: `https://cloud.google.com/iam/docs/overview`

3.3 Cloud Storage

Google Cloud Storage represents service with high level of availability and durability over the world. Storage of data is highly persistent by replication data over Google infrastructure. Service security protection is based on model end-to-end, thus in-flight and at rest. Several ways for controlling storage are available. Command line interface *gsutil* and programming API for build of reliable and fast networking application. The pricing model consists of different tariffs according to data durability, availability and performance of storage.

Storage classes are represented by three types of storages. The classification is derived from an characteristic of particular storage. *Standard storage* with high availability and low latency. The Standard storage suits well for frequent access such as data of web services or mobile applications. Next class, *Durable Reduced Availability* is characteristics with less availability than *Standard* one. This class is suited for jobs, where less availability is accepted and for a cost-sensitive project as well. Last class, *Cloud Storage Nearline* fills the gap between mentioned classes. Attributes are of slightly lower availability and slightly higher latency than Standard storage with lower cost. Storage for data backup or archiving data are two explanation of use cases for last class.

Buckets location are configurable by three properties, specifically: global unique name, storage class and geographical location where the bucket is stored.

```
#creating two new buckets using command-line tool gsutil
$ gsutil mb -p my-project gs://mwdata gs://mwprocessed -c
durable_reduced_availability
```

All buckets are covered under a single namespace. Thus, the name of bucket must be unique. Google defined rules for naming buckets which are based on DNS naming conventions. Validator for process creation responds by error message in the wrong case.

Objects holds data information placed in storage. Object has two components: user data and metadata of quality. All *objects* are immutable after creation. Thus, incremental changes are not available. However, an object can be overwritten. Before the new version of an object is available, the old version is still reachable for reading. Overwriting of a

single object can proceed up to one second. Otherwise *503 Service Unavailable errors* is responded.

Consistency of data handles global Google network of data centres. When success response is received, uploaded data are available from any location in Google network. Google has designed and constructed a private network for fast data transfer between their data centres around the word. Thus, data throughput of Google backbone network is higher than the capability of the internet-facing network. The latency for writing is slightly higher for replicated store than non-replicated. Restriction of accessing metadata of object immediately after deletion object is good example of consistent behavior. It is handled by *404 Not Found* status code.

3.3.1 Access control

The configuration of access control can be set using *gsutil* command-line tool or API. Cloud storage consists three different ways of buckets access management.

- **Access Control Lists (ACLs)** provide way to manage read or write access for specified Google accounts and groups.
- **Signed URLs (query string authentication)** provide a way to set time limited read or write access to anyone who disposes of URL address. This way is a choice for users without Google account.
- **Signed Policy Documents** allows specifying what can be uploaded to a bucket. Size, content type and other specification can be set by policy handler.

Additionally there is available access control on the Project level. For controlling user's rules to access bucket on the project level there is developed *Google Cloud Identity and Access Management* (IAM). This service is described on the end of the section 3.3.1.

Permissions is divided to read and write control. For *objects* there is available option to let user download *object* and modify metadata by owner. By default *objects* are owned by original requester who upload the *object*. As *object*, *Buckets* are readable. In addition, list of user let them to create, overwrite and delete objects in *bucket*. Owner can let user

to read and write permission on the bucket include metadata. By default, *buckets* are owned by the project owner group.

Scopes of authentication consist of six methods for specifying ACL such Google Storage ID, Google account email address, Google group email address, Google Apps domain, Special identifier for all Google account holders and Special identifier for all users.

```
#Identify an object that the user uploaded.
gsutil acl get gs://mwdata
```

Access control list is configurable by command line tools *gsutil acl* or by API based on JSON or XML structures. The management of ACL is based on concentric permissions, which provide faster configuration. If user grant write permission, automatically get also read permission. In case of grant of owner permission, read and write is also reachable.

```
#usage of a predefined ACL to an object during object upload
gsutil cp -a bucket-owner-full-control mwbck.zip gs://mwdata
```

In addition, for quick settings of access restrictions such as project-private, list of predefined ACL is available: private, public-read, public-read-write, authenticated-read, bucket-owner-read and bucket-owner-full-control.

3.4 Dataproc: Cluster Services

Dataproc is a package of tools for managing Hadoop 1 and Spark[4]. The process of a deploying cluster is fully automatized using the default configuration. Hadoop and Spark deployment are compatible with official releases. With default configuration, the configuration of automatic deployment is simple. On the other hand in the most cases, it is necessary to configure cluster manually to fulfil custom requirements. Next to the standard command-line tool *gcloud* there is helper software *bdutil*. The tool allows complex configuration including the definition of hardware configuration, combining services from wide Google cloud portfolio and custom additional tasks for installation and configuration cluster. This command-line utility are based on authentication provided by *gcloud* interface.

[4]Apache spark: `http://spark.apache.org/`

In the list of the main characteristic of *Dataproc* solution is a low cost, quick starting and integration with Google Cloud services. The pricing is $0.01 per CPU for an hour (19.3.2016). Dataproc service can include *Preemptible instances* which offers pricing per minutes (min 10 minutes). This choice is suitable for batch jobs and fault-tolerant workloads. The time for starting cluster is up to 90 seconds in average. To compare e.g with Amazon cloud services(EMR) [21], starting Google cluster is 3 times faster.

Create and manage cluster Deploying process can be handled by three main ways. The first one uses *gcloud* command-line tool, the second by running *bdutil* 3.4.1 script and the last option with using *Cloud Dataproc API - cluster.create*. Google Cloud web interface offers framework for cluster deployment as well. On the other hand, the configuration and management is considerably limited. Command-line tool *gcloud* consist of parameters: *cluster* for managing and describing cluster, *jobs* for submitting and managing jobs on cluster, finally parameter *operation* for handling operation, like cancel or delete active operation by *operation_id*. For more advanced cluster deployment there is suited script tool *bdutil* which is package for custom configuration of cluster. The advantages is easy redeployment and reproducibility custom configuration of cluster.

```
#creation of cluster with default settings using command-line tool gcloud
$ gcloud dataproc clusters create <cluster-name>
```

Cluster Properties The configuration behind is not trivial but Google *Dataproc* provides semi-automatic deployment without big configuration investments. The configuration of open source software as Spark, Hadoop and extensions as Hive are based on several XML files and plain text configuration files. For cluster deployment using *Dataproc*, the configuration is based on the same configuration files. Some properties are reserved by Google and cannot be overridden to protect functionality of Cloud Dataproc. The crucial feature of cluster is immutable configuration after start. Thus, all configuration must be set properly before daemons on cluster are started. For applying changes must be cluster restarted.

File system Google Cloud Dataproc offers two different storages based on distributed filesystems. In chapter Hadoop 1.1 is described Hadoop Distributed File system (HDFS) which is default filesystem for Apache Hadoop framework and is as an optional in Google

Services. As default filesystem for Hadoop and Spark deployment provided by Google Cloud is *Google Cloud storage*.

Google Cloud Storage offers direct data access in available storage for the rest of Google services. In addition the interoperability provide access between Spark and Hadoop on Google. Next advantage is high data availability provided by Cloud Storages, with high availability and global replication without reduction of performance. The key feature is data access after cluster is shut down. The pricing of cluster instance is based on time. Thus, to save costs, cluster should run only when is in use. With using HDFS as distributed filesystem, after shutting down clusters VM instances, data are removed and lost as well. In contrast, data stored on Cloud Storage are available after cluster is stopped and can be linked again in further cluster deployment.

Interaction between Hadoop Distributed System and Dataproc cloud is limited to compare to default Google Storage. HDFS is scalable across VM's, but doesn't scale per instance as well as Cloud Storage due to VM disk bandwidth limits.

Machine type Google Cloud Datagram clusters are built on Google Compute Engine instances. *Machine types* configuration defines the virtualized hardware resources available to an virtual machine instance. Two ways how to define configuration are available: *predefined instance* and *custom machine types*.

```
# print list of predefined instances
$ gcloud compute machine-types list
NAME            ZONE           CPUS MEMORY_GB DEPRECATED
f1-micro        asia-east1-c   1      0.60
g1-small        asia-east1-c   1      1.70
n1-highcpu-16   asia-east1-c   16    14.40
...
```

Customization of machines availability is suitable for project with workloads where the computing engine requires high performance machine resources or just where predefined machines are not fit great to workloads. Once a custom machine template is defined, the configuration is available for all connected services over Google Cloud portfolio.

3.4.1 bdutil: Spark and Hadoop on Google Cloud

As has been already mentioned, *bdutil* is a command line script designed for managing Hadoop instance on Google Compute Engine. It provides deployment, configuration and

shut-down of Hadoop instances in quickly and reproducible way, even for complex configuration. Script is based on Bash v3 or later which is available on VM of Google Compute Engine and the most of Linux distributions. In addition, shell command-line tool *gcloud dataproc*, *bdutil* includes configurable scripts for deployment Spark and Hadoop. The default configuration is Hadoop 1.x with Cloud Storage as the storage system. Furthermore, *bdutil* is designed to allow writing custom environment variable configuration files. For example: *CONFIGBUCKET* - path to Google Storage, *PROJECT* - ID of Google Platform project or *GCE_MACHINE_TYPE* - machine type of the Google Compute Engine.

4 Related Technologies Behind

4.1 Geospatial Framework GRASS GIS

*"**Geographic Resources Analysis Support System**, commonly referred to as GRASS GIS, is a Geographic Information System (GIS) used for data management, image processing, graphics production, spatial modeling, and visualization of many types of data. It is Free (Libre) Software/Open Source released under GNU General Public License (GPL) >= V2. GRASS GIS is an official project of the Open Source Geospatial Foundation."*[5]

Figure 1.8: GRASS GIS Logo

GRASS GIS vector map is a data layer which consists of number sets in geographic space. These objects are represented by points, lines, areas, and volumes. Usually, each feature in the map is connected with the set of attributes and attached values.

[5]Cited from: `https://grass.osgeo.org/documentation/general-overview/`

For manipulation of vector data in GRASS GIS database several drivers for interaction with different database backends are available. The main module for import and export of vectors datasets is v.in.ogr and v.out.ogr which is based on widely used GDAL library. In addition, new module for v.import is available. Its advantage is projection conversion on the fly during importing. In the current stable version of GRASS GIS is there implemented a driver for PostGIS support. PostGIS is an extension of object-rational PosgreSQL database which brings the support for geographic objects. In addition PostGIS extension consists of package of location-based function for building a spatial query. As PostGIS is a spatial extension for PostgreSQL, analogically ESRI Spatial Framework for Hadoop brings the support of spatial analysis for Hadoop and Hive.

The key feature for this project is the fact that GDAL supports reading and writing GeoJSON data format, which is suitable for storing and analyzing data using MapReduce framework. On the other hand, there is no support for reading and writing to the unenclosed GeoJSON, and some edits of file format must be additionally done. ESRI Spatial Frameworks for Hadoop allows exporting only to ESRI GeoJSON. The description of GeoJSON and its serialization is mandatory due to the design of MapReduce for parallel computing. GeoJSON and its features are described in the second part of the thesis.

Experimental Framework

In first part of the thesis there has been introduced background of technologies which are linked to further text, thus to the outputs of this work. The text below points the configuration of cluster, development of GRASS Hadoop Framework (GHF) and its usage. The configuration of network, cloud services and Hadoop together with case study of GHF usage explain full workflow of processing spatial data using Hadoop within desktop GIS.

In addition, within this work, side project focused on transformation of big data from SQL PostgreSQL to BigQuery is accomplished. Microwave data captured by cellular operator T-Mobile stored in PostgreSQL server consists 3 billions of rows which restricts valuable interaction with database.

1 Setup working environment

As the main resource of data storage and computing engine for practical framework Google Cloud Platform has been used. Thus, all configuration of Hadoop cluster, data storage, and networking is linked to cloud services. Additionally, development of GRASS client framework for Hadoop and consequential testing as well.

Google Cloud Platform offers free trial 60 days promotion for using their services. The offer is limited by two months or 300 $ for whichever services of Google Cloud portfolio.

1.1 Set up project: gcloud

Using Google Cloud Platform web interface has been created new project placed in App Engine location *europe-west*. The project is characterized by Project name, Project ID and billing account. On local computer has been installed Cloud SDK which includes

gcloud and *gsutil* tools for remote control Google Cloud services. Initialization of gcloud using command *gcloud init*[1] provide authentication process using web browser and request of access to the project.

Compute Engine uses a default zone and region based on information from the project metadata. In this case, configuration is inherited from the Project settings without an additional changes.

1.2 Data Management

Google Data Store is used as the main data house for the project. Command line tool *gsutil* is for object operations over buckets of Google Storage. For copying large object is very suitable to use a function for slicing which allows parallel downloading and uploading.

For downloading object using HTTP request gsutil performs slicing of object on Google Storage and preallocate space in the destination folder. The final file is renamed after all slices are processed. For downloading is not required additional space on local disk. The performing of slices is configurable by parameter *sliced_object_download_threshold* which define a maximum number of slices. It can be useful to protect the disk from overloading. For efficient uploading of large files is crcmod which required an installation of CRC32C package.

1.3 Networking

The essential network configuration is necessary for accessing Hadoop cluster from outside of Google sub-network. The VM instances has internal IP and external IP. Both of them are dynamically attached during deployment of instance. Optionally it is possible to reserve external static IP and attach it to the VM. For usage Hadoop for batch processing it is not necessary to configure user network but is necessary to configure firewall.

By default, incoming traffic from outside of your network is blocked. To allow incoming traffic a firewall rule must be set. Firewall rules regulate only incoming traffic to an instance. When a connection is established with an instance, traffic is permitted in both

[1]To initialize project on remote computer without X Window System forwarding must be used flag *–console-only*.

name	IP range	allowed protocol	targets tag
default-allow-icmp	0.0.0.0/0	icmp	Apply to all targets
default-allow-internal	10.128.0.0/9	tcp:0-65535,icmp	Apply to all targets
default-allow-rdp	0.0.0.0/0	tcp:3389	Apply to all targets
default-allow-ssh	0.0.0.0/0	tcp:22	Apply to all targets

Table 2.1: Default values of firewall configuration for accessing machines from external networks.

directions over that connection. On web of Google Cloud Platform there is a manager for defining of firewall rules. By default several profiles are predefined, how was already mentioned in table 2.1.

1.3.1 Network Configuration

For processing of data within desktop GIS to configure network is mandatory. If we assume that the cluster is private and mainly for batch processing the predefined network can be used. The network must be reconfigured to be accessible from outside of Google sub network. Configuration of two network characteristics is mandatory.

- Firewall profiles
- Local Hosts

Firewall For using machine capacity and services on Google Cloud, the installation process is described in section *Setup working environment* 1. The proper network configuration for connection to Hadoop cluster is necessary. There are several steps for basic configuration of Google network and local computer for batch processing. Firstly, in Google Cloud must be added configuration profile to firewall. Hadoop and Hive has several ports for their services. Java daemons have by default open ports for accessing it. Important ports for controlling Hadoop and Hive are 50700 for accessing NameNode, 50705 for DataNodes and for Hive 100000. In firewall there must be created a profile which allows access of these ports from outside of Google sub network. To restrict access to the opens port, in firewall profile must be set IP address limitation. There are several option how to limit IP. The most common way is to define allowed IP address range. This configuration is suitable for single user usage. Second way is to define sub networks of

Google Cloud. In addition is possible to allow tags of VM of Google Cloud. The last option is 0.0.0.0/0, thus any IP have access.

Local Hosts For accessing HDFS from GRASS Hadoop Framework the driver must know all external IP addresses of master and workers of cluster. Each server has external IP address and local host name. The local host name is automatically generated during cluster deployment. In Hadoop configuration files are registered local host names of workers. After the client accesses HDFS daemon (port 50700) then it receives message with local host and port of workers instead of IP address. If the client is running from different machine than master, these IP addresses and local host names must be defined. In Linux systems the configuration of local hosts are declared in file */etc/hosts*.

```
127.0.0.1       localhost
::1             localhost ip6-localhost ip6-loopback
ff02::1         ip6-allnodes
ff02::2         ip6-allrouters

10.132.0.2 cluster-2-m.c.hadoop-matej.internal cluster-2-m  # Added by Google
```

Master and workers, each of them has configuration similar to example of master machine in Google Cloud. Important is the last line. This line must be append to local machine *etc/hosts* file.

1.4 Hadoop Deployment

For deployment of Hadoop cluster there is used bdutil command line package builds from several bash scripts and XML configuration files of Hadoop and its extensions. In the section below a essential configuration of Hadoop on Google Dataporoc service is described. Mainly there is shown the important part which touches requirements for using ESRI Spatial Frameworks for Hadoop. The source of bdutil configuration is in digital attachment of the work 4.

Google Cloud Dataproc brings field automated deployment of Hadoop cluster to the cloud. To compare with manual configuration of Hadoop cluster, it allows fast and efficient deployment with wide options to interact with services of Google Cloud portfolio.

The configuration files for *'standard'* Hadoop are stored in **$HADOOP_HOME /conf/** and consist of primary xml files. Many of parameters values are defined by default and they may be omitted in configuration files. In addition, the default configuration

folder includes shell file which specified environment variables of Hadoop Daemon. Default configuration folder lists:

- **hadoop-env.sh** The file specifies environment variables that affects JDK used by daemons. For example **$JAVA_HOME**, *HADOOP_HOME*.
- **core-site.sh** This file informs Hadoop daemon where NameNode runs in the cluster. Furthermore, it informs about port number used for Hadoop instance, memory limit for data size, memory allocated for HDFS, and size of read and write buffers.
- **hdfs-site.sh** This file contains the configuration settings for HDFS daemons, the Name Node, the Secondary Name Node, and the data nodes. Usually the file configures block replication and permission checking on HDFS.
- **mapred-site.sh** This file contains the configuration settings for MapReduce daemons such the JobTracker and the TaskTracker.

In addition default configuration directory includes two text files: *masters* and *slaves*. Both includes hostname of slaves or master where each line represents one host name address. Master file informs about the Secondary NameNode location to Hadoop daemon. The *masters* file at Master server contains an IP address of Secondary Name Node servers. The second one, *slaves* on Slave server contains the IP address of the slave node. Notice that the *slaves* file at Slave node contains only its own IP address and not of any other Data Nodes in the cluster.

Hadoop modes Hadoop can be deployed in several modes for different aim of usage.

- **Non-distributed mode** Hadoop running as a single Java process. This mode is suitable for MapReduce development and debugging.
- **Pseudo-Distributed mode** configuration allows to run Hadoop on single-node and behave like multi-node. It is provided by running Hadoop daemons in separate Java processes. To configure pseudo-distributed Hadoop must be modify *core-site.xml* and *hdfs-site.xml* file [2].

```
#etc/hadoop/core-site.xml:
-
<configuration>
```

```
<property>
<name>fs.defaultFS</name>
<value>hdfs://localhost:9000</value>
</property>
</configuration>


#etc/hadoop/hdfs-site.xml:
-
<configuration>
<property>
<name>dfs.replication</name>
<value>1</value>
</property>
</configuration>
```

- **Fully distributed cluster** is an ideal approach for harnessing the potential of parallel computations. The nodes are on virtualized machines with assigned machine capability such a CPU, memory, and disk or deployed on physical machines. The most efficient approach which is used in the production or in large clusters is an installation of Hadoop directly on the machine with Linux without virtualised environment. This configuration is used for high performance spatial processing in case study of this work.

1.4.1 Configuration - bdutil

In contrast to standard Hadoop deployment, a configuration in *Dataproc* using bdutil is extended by parameters and configuration files which handle automated deployment and interaction with other Google Cloud Services.

Tool bdutil comes with several bash scripts which set configuration of the cluster: type of framework- Hadoop or Spark, version of Hadoop, Hadoop extension (Hive, Pig) and connectors to other services of Google.

bduitl_env.sh The main script called *bdutil_env.sh* consists of mandatory and free environmental variables for setting environment, hardware and configuration deployment.

- **Environment variables** provide the setting which refers to the id of the project and to the default bucket in GCS. The defined bucket holds SSH keys and configuration of the filesystem. In the case of usage GCS as a filesystem for Hadoop, the

Bash script	Description
bdutil_env.sh	The base of configuration. It is always run by bdutil. Must be modified 1.4.1
single_node_env.sh	Deploys a pseudo-distributed cluster on a single VM. This approach is suitable for testing clients etc.
hadoop2_env.sh	Deploys a cluster with the latest stable version of Hadoop 2.x instead of the traditional Hadoop 1.x version.
bigquery_env.sh	Deploys a cluster with the BigQuery connector for Hadoop installed. Not compatible with hadoop2_env.sh.
extensions/querytools/querytools_env.sh	Deploys a cluster with Apache Pig and Apache Hive installed. Not compatible with hadoop2_env.sh.
extensions/spark/spark_env.sh	Deploys a cluster with Apache Spark installed.

Table 2.2: Configuration of shell scripts in bdutil [28]

defined GCS is linked during cluster deployment and all files are natively accessible by standard Hadoop commands e.g copy file *hadoop fs -cp /data/* /tmp/*.

- **Hardware configuration** allows to specify name, shape, location and size of a cluster. Of the same importance has definition of machine type, with reflecting final performance of cluster and pricing of service as well. In addition, the parameters for a number of workers and size of the disk are also crucial.

  Hardware configuration allows setting of different disk sizes and type to master and workers. Furthermore, extra persistent disks (PDs) can be linked. Available virtual machines of service account can be attached to a further cluster. For that case, the GCS connector must by allowed.

- **Deployment** section consists of parameters for customizing Hadoop installation on VMs workers and master. The configuration of GCS connector allows to use it as a distributed filesystem *GS* and *BIGQUERRY* connector allows read/write access to Google BigQuery. File system of a cluster is set by default to GCS, the second option is standard HDFS. This group consists of several paths of configuration for defining the destination of Hadoop installation and its related files. Furthermore, it is essential to configure permission of *HDFS* data access.

  Parameter CORES_PER_MAP_TASK and CORES_PER_REDUCE_TASK must be defined as a decimal number which controls the number of maps and reduces slots

on each node. The number is computed as a ratio of the number of virtual cores on the node. For example if the machine of specification *n1-standard-2* is used (2 cores) and the parameter is set to 1 would have $\frac{2}{1} = 2$ map/reduce slots.

File	Name	Value
bdutil_env.sh	CONFIGBUCKET	mwdata_export
	PROJECT	spatial-hadoop
	GCE_IMAGE	debian-7-backports
	GCE_MACHINE_TYPE	n1-standard-2
	GCE_ZONE	europe-west1-b
	GCE_NETWORK	default
	GCE_MASTER_MACHINE_TYPE	n1-standard-4
	PREEMPTIBLE_FRACTION	1.0
	PREFIX	hadoop
	NUM_WORKERS	2
	MASTER_BOOT_DISK_SIZE_GB	200
	DEFAULT_FS	gs
	ENABLE_HDFS	false
	ENABLE_HDFS_PERMISSIONS	false
	INSTALL_GCS_CONNECTOR	true
	CORES_PER_MAP_TASK	1.0
	CORES_PER_REDUCE_TASK	1.0
hdfs-template.xml	dfs.webhdfs.enabled	true

Table 2.3: Selected configuration parameters of bdutil tool

Hadoop configuration files are stored in *conf/hadoop1* for Hadoop version 1.x VM and *conf/hadoop2* for 2.x version both are placed within *bdutil* base directory. In compassion to standard Hadoop configuration files, these are extended by files for configuration Bigtable (Hadoop1.x), BigQuery (Hadoop2.x) and GCS. Particular values of parameters are linked over environment variables to the main script *bdutil_env.sh* and other bash files stored in directory *libexec*. The example below show configuration of task tracker where the value is set by global variable *MAP_SLOTS*. To call global variable must be used convention *envVar name=""*.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
<property>
<name>mapred.tasktracker.map.tasks.maximum</name>
<value><envVar name="MAP_SLOTS" /></value>
<description>The maximum number of map tasks that will
be run simultaneously by a tasktracker.</description>
</property>
</configuration>
```

Limitation of Hive/Pig and Hadoop initialized on the top of GS filesystem is that Pig and Hive will tend to rely on multi-stage pipelines more heavily than plain Hadoop MapReduce, thus they are vulnerable to eventual consistency. There is not known straight solution. As workaround can be accepted transfer data from GS using explicit *gs:// URIs* and likewise to write the final output to GS, letting any intermediate cross-stage items get stored in HDFS temporarily.

As mentioned in Dataproc there are available two version of Hadoop. For two spatial frameworks from ESRI and HadoopGIS must be used version 1.x of Hadoop due to dependency on Hive, which is in Google Dataproc and incompatible with versions Hadoop 2.x. The main limitation of the versions 1.x is unsupported YARN framework which is the next generation of MapReduce framework also called MapReduce 2.0 (MRv2)[29]. The main difference between is in the architecture where JobTracker, resource management, and job scheduling are split into separate daemons.

Running cluster is done by executing bdutil bash file with flags and parameters. Flag *–env_var_files* runs selected bash files from the predefined group 2.2. Besides predefined bash files it is possible to add a custom script which can be executed as well. The order of files execution is according to the sort in the list. The main script *bduitl_env.sh* 1.4.1 is executed as the first one without presence in the list. In example below there is in the list the script for downloading and compiling spatial frameworks for Hadoop and their dependency. The spatial1.sh script is in digital attachment within bdutil.

```
COMMAND_GROUPS+=(
'install_spatial:
spatial1.sh
'
)

COMMAND_STEPS+=(
'install_spatial'
)
```

The script is defined by command group where is defined a path to the bash for execution. List of command steps is a global variable and its items appends to execution the list.

```
ROLE=$(/usr/share/google/get_metadata_value attributes/role)
if [[ "${ROLE}" == 'Master' ]]; then
        do----
fi
```

The performance of cluster varies on particular task and user demands. Below there are described two cases of configuration cluster.

- **Case 1 - configuration for development** This configuration of cluster suits well to developing of Hadoop and Hive code or testing from client site. Cluster is configured as pseudo mode.

```
$ ./bdutil -P spatial-cluster --env_var_files \
extensions/querytools/querytools_env.sh,single_node_env.sh,spatial.sh deploy
```

- **Case 2 - configuration for computational** To compare with configuration above, Case 2 is fully distributed deployment of cluster. For modifying *bdutil* setup configuration of hardware, storage etc. serves flags, which overwrite it.

```
$ ./bdutil -P spatial-cluster --num_workers 10 \
--machine_type n1-standard-6\
--env_var_files \
extensions/querytools/querytools_env.sh,spatial.sh deploy
```

1.4.2 Configuration - gcloud dataproc

Second option for cluster deployment is command line tools *gcloud dataproc*. This tool is based on flags and parameters which define the specification of deployed cluster. In contrast to bdutil tools, gcloud dataproc is based on different system images which are releasing under versions. Currently, the latest version of image - 1.0 is built with Hadoop 2.x and Hive 1.2. Thus, this system image supports version combination wich bdutil not. Tool bdutil has been tested with latest versions of Hive and Hadoop and the GCS filesystem doesn't work properly.

Properties of configuration files mentioned above 1.4 and in table 2.4 are set before deployment by parameter *–properties* and its syntax. Below is shown example of deployment of cluster with two nodes. The last parameter configuring the file *hive-site.xml*, specifically defining the path for default source of *.jar libraries, which are automatically loaded to the Hive environment. [34]

file_prefix	File
core	core-site.xml
hdfs	hdfs-site.xml
mapred	mapred-site.xml
yarn	yarn-site.xml
hive	hive-site.xml

Table 2.4: Prefix for available configure files.

```
gcloud dataproc clusters create cluster-2
  --bucket mw_data --zone europe-west1-c
  --master-machine-type n1-standard-4
  --master-boot-disk-size 500
  --num-workers 2
  --worker-machine-type n1-standard-2
  --worker-boot-disk-size 300
  --image-version 1.0
  --scopes 'https://www.googleapis.com/auth/cloud-platform'
  --project spatial-hadoop
  --properties 'hive:'hive.aux.jars.path=file:///usr/local/spatial/<jar>'
```

There are several notes and limitations of the configuration, such as: 1) Google for protecting the functionality of Dataproc reserve same parameters and 2) the changes must be defined before daemons of Hadoop or its extensions start. For specifying more parameters it is possible to define several properties at once, by using a comma.

1.4.3 Additional Settings of Cluster

If the cluster is deployed using bdutil 1.4.1 with parameter *–env_var_files spatial.sh* this script on master server installs spatial packages, dependency and Java packages for serialization JSON. If the cluster runs by another way, the libraries must be compile manually. In attachment is described important configuration for packaging Java libraries B

Hive jar source For using external libraries from Hive console it is necessary to add Java packages to Hadoop environment path or to Hive. Available are several ways:

- Add jar from console

```
$hive add jar ${env:HIVE_HOME}/lib/ESRI-geometry-api.jar;
```

- Add destination <path> to Hadoop path; (works remotely)
- Add to *hive-site.xml* parameter *hd.hive.aux.jars.path* with value <path.jar>; (works remotely)
- Create in $HIVE_HOME directory auxlib and copy jar's there.
- As workaround is to export path- export HIVE_AUX_JARS_PATH= <path>; (works locally)

The third option is the most correct and for using GHF must be used. After change is applied, Hive and Hadoop daemons must be restarted.

2 GRASS Hadoop framework

Background Desktop application GRASS GIS became as a powerful environment for analyzing spatial data, which can reach characteristic of big data. In a box of GRASS GIS there are hundreds of modules. The main motivation behind GRASS GIS framework for Hadoop is a vision which allows user processing big vector dataset natively from desktop environment. In proprietary GIS field, ESRI comes with package *Geoprocessing tools for Hadoop*[31] which has been implemented for ArcMap desktop application as a native interface for interaction with their Java library *ESRI Spatial framework for Hadoop*. More specifically, the tools enable:

- exchange data between ArcGIS Geodatabase and a Hadoop system, and allows ArcGIS to run Hadoop workflow jobs,
- copy data files from ArcGIS to Hadoop, and copy files from Hadoop to ArcGIS, and
- run an Oozie workflow in Hadoop, and to check the status of a submitted workflow.

Described features of ESRI approach are the source of inspiration for GRASS Hadoop framework, which is developed within this work.

In the first, the chapter 2.3 of this work describes and summarizes features of three different frameworks for processing large-scale data. As the most suitable framework for this project became ESRI Framework, which can be deployed without extra customization, natively includes considerable package of spatial functions, furthermore, the documentation and examples of usage are the most comprehensive. The selected library is used for testing and for explanation of usage. Even other libraries have not been tested, the modular design of framework allows to partition each step of workflow which ensure easier future development. For example, user can generate GeoJSON from GRASS map and load it to HDFS using *hd.hdfs.in.vector*, and after that create Hive table using *hd.hive.json.table* and load data. These steps are independent. Thus the interface for interaction between GRASS and Hive/Hadoop can be used as library for interaction with different Hadoop spatial library. Due the fact that HadoopGIS is based on Hive, the GHF supports it as well. Within this work, the testing of HadoopGIS library is not provided.

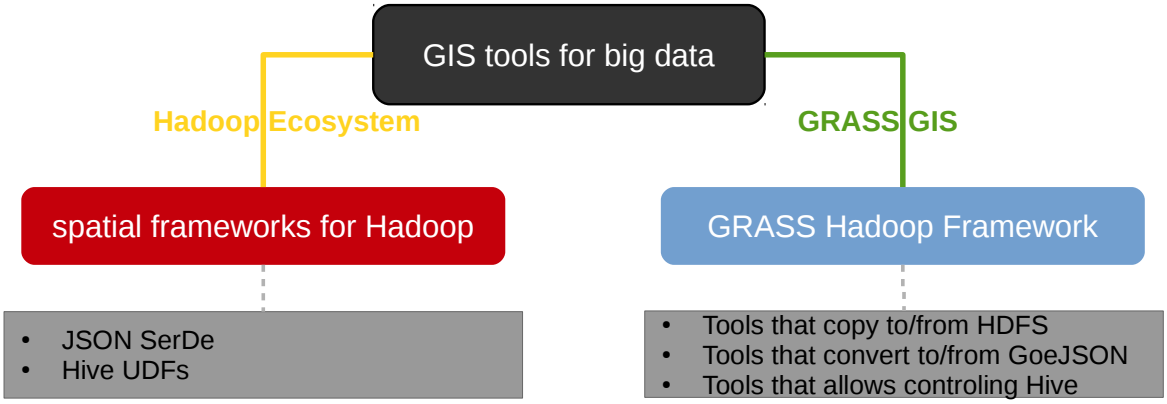


Figure 2.1: Resources for processing big data within desktop GIS

2.1 Functionality

The GRASS Hadoop Framework (GHF) is Add-ons package of modules for GRASS GIS. It provides interaction between GIS desktop environment and Hadoop/Hive. Although GHF can be used just for transferring data between filesystem and HDFS, the main purpose is to bring capability of Hadoop spatial frameworks to the GIS desktop application.

Hadoop-based spatial libraries can be used externally from the command line console

e.g. *hiveserver2* or *hadoop fs* command line tool. This way of usage can be more powerful for experienced users with administration access to the cluster and knowledge of scripting, especially in bash. If we assume that average user prefers friendly interface, than GHF is suitable choice.

The package includes several modules 2.5 for managing connections, exporting GRASS maps to customized GeoJSON (suitable for serialization), loading them to HDFS, creating tables in Hive, loading data into table and after processing, importing result back to GRASS.

hd.hdfs.* modules	description	hd.hive.* modules	description
hd.hdfs.db.connect	Management of connections	hd.hive.json.table	Create table for GeoJSON
hd.hdfs.in.fs	Put data to HDFS from local	hd.hive.csv.table	Create table for csv data
hd.hdfs.in.vector	Put GRASS vectors to HDFS	hd.hive.execute	Execute Hive command
hd.hdfs.out.vector	Put vectors from HDFS to GRASS	hd.hive.select	Execute Hive query
hd.hdfs.info	HDFS metadata	hd.hive.info	Hive metadata

Table 2.5: Modules of GRASS Hadoop Framework

2.1.1 Connection management

The handler of connection drivers for Hadoop/Hive is covered under *hd.hdfs.db.connect*. The module provides storing of connection profiles in default GRASS GIS database backend which is SQLite by default. The usage of the database manager is derived from current GRASS db.* modules. Thus, based on set up primary connection which is use for all involved modules. In contrast, database manager for HDFS allows setting connection id and its driver. So for each type of database (driver) can be stored several user connections distinctive by user defined id (*conn_ide* parameter) meanwhile each driver can have only one primary connection.

Defining connection Parameter *driver* and *conn_id* are mandatory for each connection profile. Parameter *driver* defines the protocol for communication with database and *conn_id* is a free unique string of connection profile. Other parameters as *host*, *port*, *login*, *passwd*, *schema*, *authmechanism* depends on a configuration of database server. After a new connection is added, the module automatically set the new one as active.

parameters	description	flags	description
driver	Type of database driver. options: hiveserver2, hdfs, webhdfs	-c	Print table of connection
conn_id	identificator of connection	-p	Print active connection
host	connection host	-r	Remove all connections
port	port of database	-t	Test connection by conn_type
login	user login	-a	Set active connection by conn_id and driver
passwd	password to database	–h	Print usage summary
schema	set active schema in db.	–v	Verbose module output
authmechanism	support PLAIN mechanism	–q	Quiet module output
connectionuri	connection uri of database	–ui	Force launching GUI dialog
rmi	Remove connection by id		

Table 2.6: Flags and parameters of *hd.db.connection* module.

Connections management Module offers several flags for handling management of connections. In table 2.6 there are described flags and theirs functions. In case of controlling several Hadoop clusters it is suitable to define its connection profiles and switching between by flag -a with parameter *conn_id* and *driver*.

2.1.2 GRASS HDFS interface

The modules starts with prefixes *hd.hdfs.**(exclude *hd.db.connection*) provide tools for transferring data between GRASS maps or filesystem and Hadoop distributed filesystem. In addition the module *hd.hdfs.info* provide basic metadata of files stored in HDFS.

HDFS metadata Module *hd.hdfs.info* currently supports several the basic operation for check if path exists, recursive listing of directories and creating HDFS directory.

GRASS Map to HDFS Vector maps in native GRASS format are not suitable for serialization which is needed to exploit the potential of spatial frameworks for Hadoop. The effective way and in the most cases the only possible is to store spatial data in JSON,

especially GeoJSON. This format suits well for serialization and library for reading is available in catalog of Hive.

Module *hd.hdfs.in.vector* supports transformation of GRASS map to1 GeoJSON format and transfer to HDFS. Behind the module there are two main steps. Firstly, the map is converted to GeoJSON using *v.out.ogr* and edited to format which is suitable for parsing by widely used SerDe functions for Hive. After that, custom GeoJSON format is uploaded to the destination on HDFS. By default, the HDFS path is set to *hdfs://grass_data_hdfs/<LOCATION_NAME >/<MAPSET >/vector*.

In addition, hd.hdfs.* package also includes module *hd.hdfs.in.fs* which allows transfer of external files to HDFS. Usage of this module becomes important for uploading CSV or GeoJSON files outside of GRASS. For uploading external GoeJSON files to HDFS it is necessary to modify its standardized format. The serialization for JSON has several formatting requirements.

- Each vector feature must be on separated/new line.
- Brackets must be at the same line.

```
// this will work
{ "key" : 10 }

// this will not work
{
"key" : 10
}
```

- Header of GeoJSON must be erased. For GeoJSON exported by *v.out.ogr* it means to remove first 5 lines and last 3.

```
//GeoJSON format. Not possible to serialize
{
"type": "FeatureCollection",
"crs": { "type": "name", "properties": { "name": "urn:ogc:def...

"features": [
{ "type": "Feature", "properties": { "cat": 546 }, "geometry":...
{ "type": "Feature", "properties": { "cat": 539 }, "geometry":...
]
}

// possible to serialize
{ "type": "Feature", "properties": { "cat": 546 }, "geometry":...
{ "type": "Feature", "properties": { "cat": 539 }, "geometry":...
```

HDFS GeoJSON to GRASS map Module *hd.hdfs.out.vector* handles results to get back from HDFS. The module download de serializes GeoJSON file from HDFS and in the second step modifies file to standard GeoJSON format. The second step is important to make file consistent according to GeoJSON standard. In that time, format is readable by *v.in.ogr* which is based on GDAL library and can be transformed to the native vector map format.

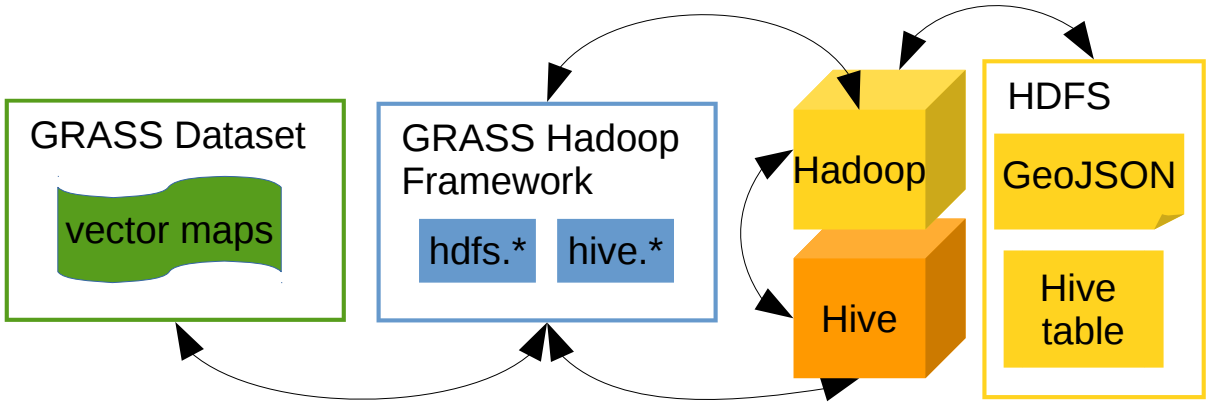


Figure 2.2: Components interaction

2.1.3 GRASS Hive interface

Two spatial frameworks, from ESRI and HadoopGIS use Hive as a top layer for usage MapReduce-based library in more expressive way. The GRASS Hadoop Framework consist several modules which start with prefix hd.hive.* analogically to mentioned hd.hdfs.* modules. Module for connection to Hive is covered by module *hd.hdfs.db.connect* which is an exception in naming nomenclature. The "hive" modules provide functions for creating tables and loading data into. Furthermore, module for select queries and execute commands is available. Main two modules *hd.hive.csv.table* and *hd.hive.json.table* support user-friendly creation of tables and loading data into. The process of creation table and loading data can be split. The hd.hive.*.table modules support creation table without loading data. This step can be provided by module *hd.hive.table.load* or simply by Hive command using *hd.hive.execute*.

Creation of table Spatial Framework from ESRI supports reading from several file format. The Framework allows creating geometric data type from WKB, JSON and Geo-

JSON. Table can be created from *hiveserver2* command line or with using developed module *hd.hive.json.table* within GHF. Defining feature of table is provided using parameters and flags of module. It helps to user make table with GeoJSON table without advanced knowledge of Hive syntax. In table 2.7 there are described main features of each parameter. The examples of usage are shown in the next section 3. The second, slightly modified module, supports creation of table suited for loading CSV data. In hd.hive.* package there is also module for loading data to table. This module is optional because is built-in for hd.hdfs.*.table modules.

parameter	description	flag	description
driver	Type of database driver options: hive_cli,hiveserver2,	-e	create external table
table	name of table	-o	overwrite data in table
attributes	python dictionary {attribute:datatype}	-d	firstly drop table if exists
struct	structure of json		
stored	output format		
serde	java class for serialization of json default: org.openx.data.jsonserde.JsonSerDe		
outformat	java class for handling output format		
jsonpath	hdfs path specifying input data		

Table 2.7: Description of *hd.hive.json.table* parameters and flags

Select and execute In the package there are two additional modules for advanced commands. The module *hd.hive.select* is intended for building custom query and produce the result according to standard output or to the file. The second one, module for execution can be used for any command. In contrast, that works properly only with non-optimised query type, when creating table, loading data, dropping databases etc. Consequently, it queries without complex output.

2.2 Implementation

Package GRASS Hadoop Framework has been developed in Python language. The background of the choice of Python language is clear. GRASS GIS natively supports Python and GRASS libraries are fully covered by API. As GRASS GIS modular design, the user interface of package GRASS Hadoop Framework is modular as well. The user interface of GHF is provided by command line modules. The design of framework is based on three layers: (1) the core library, (2) the middle interface and (3)the top level interface. The core library of presented project represent wrappers for HDFS, Hive drivers and its management of connection. The middle interface consist of factories for the top level interface, thus modules accessible from GRASS GIS user interface.

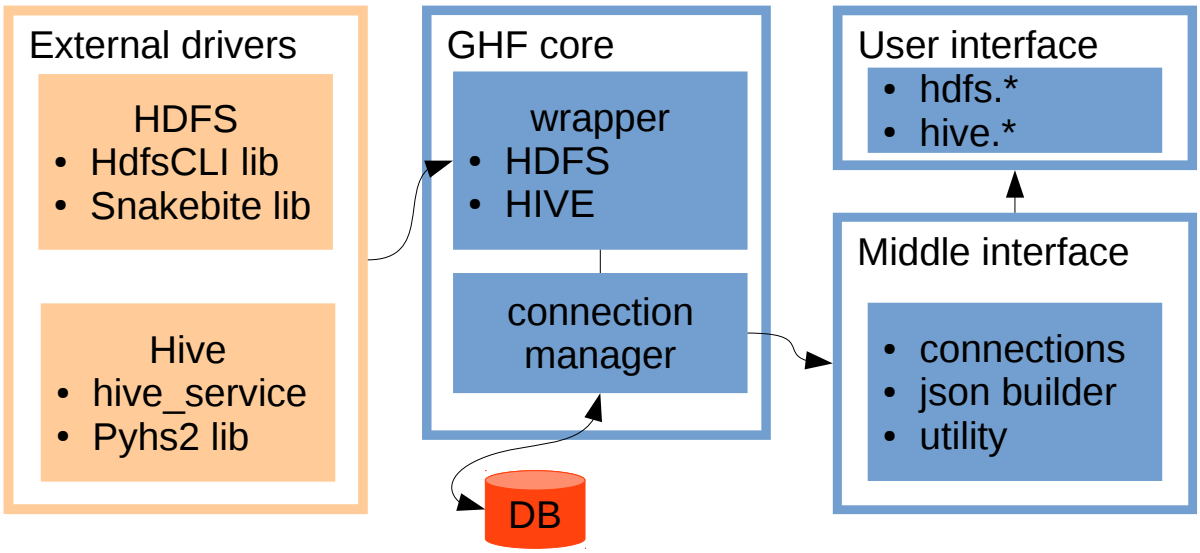


Figure 2.3: Architecture of GRASS Hadoop Framework

2.2.1 External libraries (drivers)

In this section introduction to the external libraries for interaction with HDFS and Hive, which are used in GHF, is provided.

HDFS libraries For communication with HDFS two libraries are used. The first one, wrapper of *HdfsCLI* command line shell which is primary developed to make interaction with HDFS more intuitive than standard *hadoop fs* command line shell. It has features as

completion and command history. The commandline tools is wrapped by python library which is used within GHF. The controlling of remote HDFS is based on WebHDFS REST API and HttpFS supporting both secure and insecure clusters.

The second library which has been developed within Spotify project is called Snakebite. This library is shared under Apache licence, Version 2.0. The library is written as pure HDFS client to eliminate expensive calling *hadoop* from Python. The HDFS client instead of calling hadoop uses Protocol Buffers, which are a language-neutral, platform-neutral extensible mechanism for serializing structured data.[32]. Snakebite relies on protobuf messages and implements the Hadoop RPC protocol description for talking to the NameNode.[33]

Hive libraries Similarly to HDFS interaction, Hive uses also two libraries. The first called *pyhs2* is for interaction with *hiveserver2*. It includes all required packages such as SASL and Thrift wrappers. The second used library is *hive-thrift-py*. From this Thrift based library there is used only module for accessing meta-store of Hive.

2.2.2 GHF core

The core provides API for the middle interface and consists of functions for interaction with HDFS and Hive database. More specifically, it allows to put/get data to/from HDFS and interacts with HDFS using essential filesystem commands. The part focused on Hive interaction allows to create specific table for loading CSV and JSON files, fetch metadata from meta store of Hive and execute non-optimized queries or fetch result of select query. In the core library there is used part of the code from *Airflow* project, which is currently in incubator of Apache [35]. The core library consists mainly from wrappers on the mentioned external libraries. The classes of core library can be parted to the wrappers (hooks) and to the connection manager.

Connection management on the lower level is handled by class *Connection*. The class is a place holder to store information about different database instances. The class inherits the base class for declarative class definitions of SQLAlchemy library. It allows to use ORM SQL in expressive/programmatically way.

```
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()
```

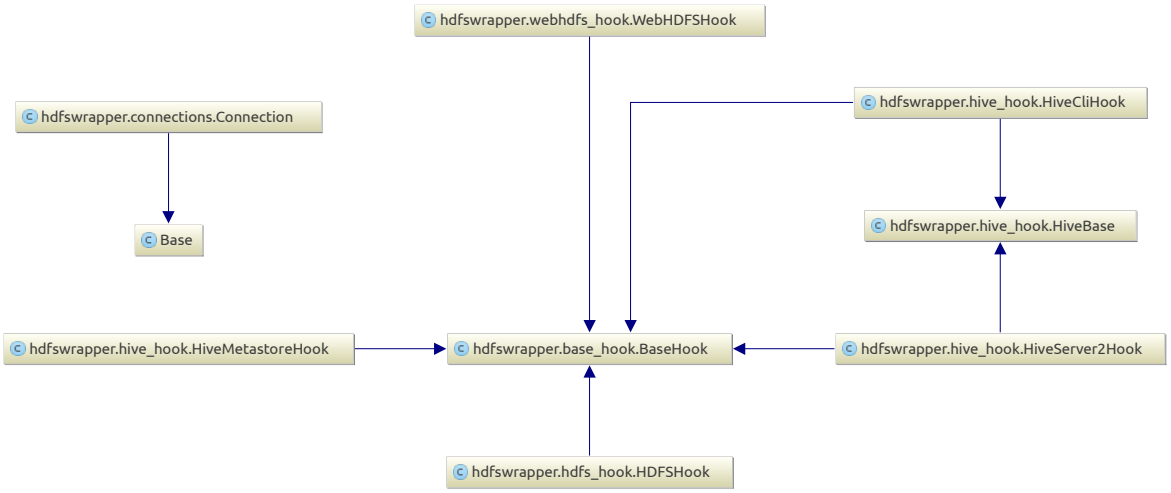


Figure 2.4: Design of the core library

```
class Connection(Base):
__tablename__ = "connection"
id = Column(Integer(), primary_key=True)
conn_id = Column(String(ID_LEN),unique=True)
conn_type = Column(String(500))
```

The class ensures integrity of initialization of *connection* table in database and the main access point for getting hooks of each connection type, such a hdfs hook, hive hook etc.

```
def get_hook(self):
"""End point for accessing hooks"""
if self.conn_type == 'hiveserver2':
return hive_hook.HiveServer2Hook(HiveServer2Hook=self.conn_id)
elif self.conn_type == 'webhdfs':
return webhdfs_hook.WebHDFSHook(webhdfs_conn_id=self.conn_id)
```

After initialization of instance for the *Connection* class method *get_hook* returns connection hook according class variable *conn_id*, which is an unique identifier for each group of connection type, such as *hiveserver2*, *webhdfs* etc.

Connection hooks Python modules called **_hook.py* consist of classes which inherits from the class *BaseHook* stored in module *base_hook.py*. *BaseHook* is an abstract class for hooks and return of object that can handle the connection and interaction to specific instances of these systems and expose consistent methods to interact with them.

Modules **_hook.py* and their hook classes provide wrappers on the external libraries. Module *hdfs_hook.py* and *webhdfs_hook.py* provide essential functions as put data to HDFS, get data from HDFS, make directory, check if path exists etc. The module *hive_hook.py* consist three classes. The class *HiveServer2Hook* allows to interact with *hiveserver2* CLI. The second class is a simple wrapper around the hive CLI. It also supports beeline Hive CLI, which is lighter due to independence on JDBC. Third class of this module - *HiveMetastoreHook* provide access to meta store of Hive.

2.2.3 Middle Interface and GHF Modules

The middle interface consists of two Python modules. The first one, called *hdfs_grass_lib*, which includes classes for Connection management and GeoJSON conversion. The second module called *hdfs_grass_util* is its helper. This interface allows to developed command lines modules. These modules mainly call API according to parsed parameters and flags.

Class *JSONBuilder* and class *GrassMapBuilder* are API for conversion between GRASS native vector map and GeoJSON. *JSONBuilder* convert GRASS map to GeoJSON and modify the text file to the form, which is suitable for serialization. In contrast, the class *GrassMapBuilder* convert exported map from HDFS to the format, witch is readable by *v.in.ogr* module. It means transformation of the result from processed analysis back to the GRASS native format.

GHF modules Already introduced modules hd.hdfs.* and hd.hive.* as a user interface and endpoint for accessing GHF from GRASS GIS user environment are developed accordingly to GRASS "standard", where the GRASS parser read input flags and parameters. GRASS parser is a function, which parse the header of Python module. Rules of syntax allow to comfortably interact with GRASS environment variables. Moreover, GRASS parser works with any programming language. In addition, *g.parser* can display an auto-generated GUI interface, help page template, and command line option checking. Below is a part of parsed lines from the module *hd.hive.json.table*.

```
# %option
# % key: driver
# % type: string
# % required: yes
# % answer: hiveserver2
# % description: Type of database driver
# % options: hiveserver2
# %end
# %flag
```

```
# % key: d
# % description: Firstly drop table if exists
# % guisection: table
# %end
```

The first option is a parameter for defining values of driver. This parameter is mandatory and currently has only one possible value. The second is a flag for dropping table, before creation the new on. This flag is in the *guisection* table, which is after generating GUI form the notebook page. Below there is example of initialization and usage of *g.parser*.

```
import grass.script as grass

def main():
grass.message(options['driver'])
grass.message(flags['-d'])

if __name__ == "__main__":
options, flags = grass.parser()
main()
```

See below the executed script from GRASS environment.

```
$ hd.hive.json.table hiveserver2 -d
out>
hiveserver2
true
```

GRASS Add-ons package The GHF has been published to GRASS Add-ons repository. GRASS module *g.extension* allows to fetch and deploy packages from Add-ons to the GRASS environment. Each package in Add-ons must include HTML manual site and Makefile for proper deploy.

3 Usage: Process Workflow

This section shows developed GRASS Hadoop Framework in use. Two main groups of particular tasks covers the full spatial processing workflow.

1. Setup working environment: cluster and client computer 3.1
2. Spatial analysis: processing spatial data using Hadoop within GRASS GIS 3.2

The second group is partly focused on description of Hive usage, such as serialization/deserialization of GeoJSON and its limitation with different libraries.

3.1 Prerequisites

To fulfil prerequisites for usage of computational power of Google Cloud for spatial analysis several steps must be done.

1. Set up project 1.1 in Google Cloud and configure storage 1.2;
2. Configure networking 1.3.1;
3. Configure and deploy Hadoop cluster with Hive 1.4;
4. Install all prerequisites on cluster B;
5. Install all prerequisites on client computer C;
6. Configure hosts on local computer 1.3.1

After finalising of each particular step the status should be: Hadoop cluster is configured and running on Google Cloud, all dependencies on cluster are installed, Goggle network allows to connect on Hadoop from local computer and GRASS GIS and GRASS Hadoop Framework is installed. If all works well the processing of data can start.

3.2 Test case: Spatial Binning

The section *Test case: Spatial Binning* will show GRASS Hadoop framework in use, especially the interaction with ESRI Spatial library for Hadoop. At the first part introduction of work aims on service management that allows the second of part of workflow to be started and proceeded. This part includes preparation of data and querying Hive database. In addition it includes usage of functions from ESRI Spatial Framework for Hadoop.

The main goal suppose to create time series dataset, which show partial contribution to OpenStreetMap for each year. The motivation behind this task is to recognize which areas of Europe are more touched by contributions and which year OpenStreetMap project registered boom. Europe data extraction of Planet dataset from OpenStreetMap repository covers area of Europe[2] and includes all history of changes within the project.

[2]source of dataset `http://osm.personalwerk.de/full-history-extracts/latest/continents/`

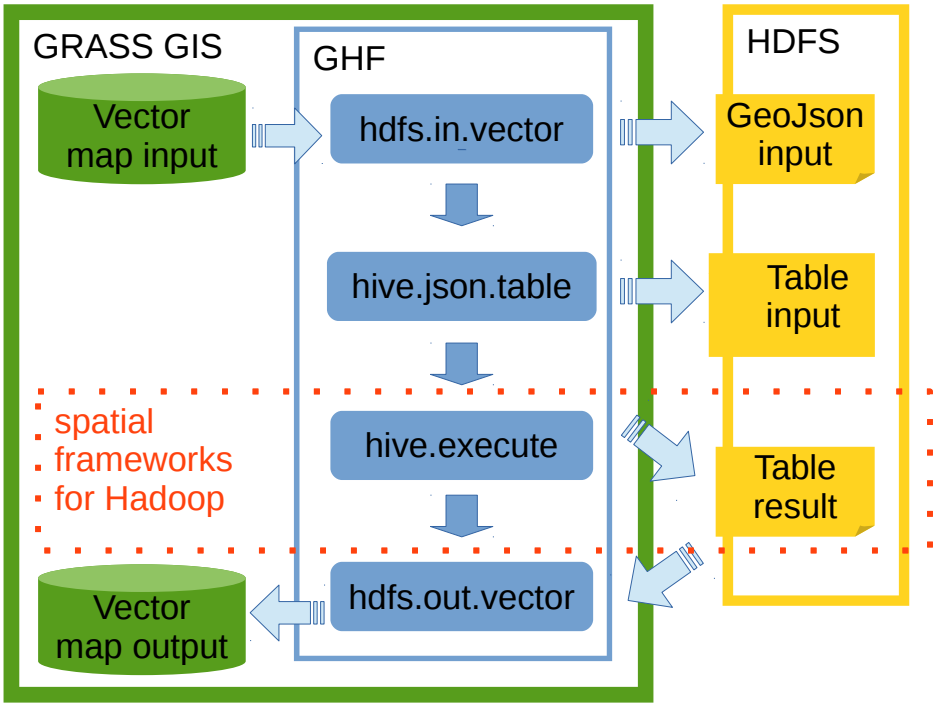


Figure 2.5: Process workflow of GRASS Hadoop Framework

The size of compressed dataset (pbf compression) of Europe is 23 Gb and consists approximately 1,3 billions of points. The goal of this case study is to create density map of partial contributions in time resolution of one year. For this kind of analyses it is suitable to use spatial binning. Data binning is a data pre-processing technique used to reduce the effects of minor observation errors [36]. The original data values which fall in a given small interval, a bin, are replaced by a value representative of that interval or just add to. It is a form of aggregation used for generalization and for creation of density maps.

3.2.1 Preparation of data

Data to HDFS The tool Osmconvert[3] allows to transform binary file to CSV, where each @ after *–csv* parameter defines column of data.

[3]Osmcomverter web page, `http://wiki.openstreetmap.org/wiki/Osmconvert`

```
$ osmconvert italy.osh.pbf  -o=europe.csv --csv-separator=, \
                            --csv="@id  @lon  @lat  @timestamp"
```

After conversion, the size of file is 116Gb. Without writing special Hive UDF (User Defined Function) the timestamps of exported file must edited to proper *datetime* format. Using *sed* command line tool it is easy even for big data.

```
$ sed "s/T/ /g"
```

Available are two ways how to put file to distributed filesystem. If GS filesystem is used and cluster is deployed using bdutil, than the best way to load data is to bucket which cluster is used. In more common way, using Hadoop and its *hadoop fs* tool can be used to put data to HDFS.

1. HDFS - loading data to the default distributed filesystem of Hadoop.

   ```
   $ hadoop fs -put europe_latest_fix.csv /data/
   ```

2. Google Storage - loading data to bucket of GS, below is shown used slicing of file which speed up uploading process.

   ```
   $ gsutil -m -o GSUtil:parallel_composite_upload_threshold=150M cp /
                   europe_latest_fix.csv gs://data/
   ```

This ensures that file is distributed over cluster to blocks. In addition according configured number of replication, data is replicated over DataNodes. By default replication is set to 2. Above is shown approach where data are downloaded directly to the server. However, in some cases user cannot control the cluster directly, only over WebHDFS access. For this case it is suitable to use module *hd.hdfs.in.fs* which support that.

```
# copy local file to HDFS on server
$ hd.hdfs.in.fs  driver=webhdfs hdfs=/data local=europe_latest_fix.csv
```

The HDFS folders and files can be check by module *hd.hdfs.info*

```
# print path recursively
$ hd.hdfs.info  driver=webhdfs path=/data -r
```

Below continue analyses focused on visualization of Europe contribution to OpenStreetMap. In attachment F is shown several steps which make further binning only for Czech Republic area. Usage of *ST_Contain* is shown.

3.2.2 Loading data to Hive

Below, in next section is described usage of Hive data warehouse. Mainly creating tables and inserting data into. In examples are shown HQL query together with examples of GHF equivalents.

Similarly to process of establishing connection to HDFS, the connection must be set connection for Hive.

```
#connection using wrapper around hiveserver2
$ hd.db.connect driver=hiveserver2 \
                conn_id=hiveserver2_id \
                host=cluster-4-m.c.hadoop-jakub.internal \
                port=10000 \
                login=matt \
                schema=default

...
      Test connection (show databases;)
              [['default']]
```

By executing query *'show databases;'* module tests if the connection passed. If the test fails, it returns connection error message.

Module *hd.hive.json.table* allows to load exported GeoJSON data to table. ESRI implements Java libraries for serialization ESRI GeoJSON which is supported by GDAL only for reading. Thus for serialization of GeoJSON another library must be used. There are several libraries available. In attachment E several notes is added about serialization JSON.

Table europe The 116 Gb CSV file which consists all points over the history of OpenStreetMap on the area covered by Europe is separated by comma and without header. For this CSV format without extra features it is possible to use built-in function for serialization which is initialized by *ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';* expression.

Some benchmarks discussed on web site of *HortonWorks*[4] point that the fastest serialization of CSV is reached using native approach. Benchmark shows that library *org.apache.hadoop.hive.serde2.OpenCSVSerde* is 3x slower than native CSV reader. On the other hand, OpenCSVSerde allows to pares CSV with strange delimiters, such blanks.

[4] `http://tinyurl.com/hvvr5nt`

Table *euro*pe according to CSV of OpenStreetMap data has columns: *id, lat, lon and time.*

```
#cretion of table for Europe dataset withinhive command-line
CREATE EXTERNAL TABLE europe(id BIGINT,
                             lat DOUBLE,lon DOUBLE,
                             time TIMESTAMP)
                             ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```

Pure HQL command is shown above. Similarly, creation of table can by done using GHFs module *hd.hive.csv.table*. Additional options as serialization, inserting data etc. are described in section *Functionality* 2.1.3.

```
#cretion of table for Europe dataset
$ hd.hive.csv.table driver=hiveserver2 table=europe \
                    attributes="id BIGINT, lat DOUBLE, lon DOUBLE, time TIMESTAMP" \
                   -e -d
```

This command of GHFs module analogically to HQL example creates external table *europe*. In addition flag *-d* firstly drops table if exists. Furthermore, the module allows to load data directly to table. In this case the command parameter is *csvpath* e.g. *csvpath=/data/europe_latest_fix.csv*.

Module *hd.hive.load* provides another option to load data to the table. Availability of this module ensures more space within building of the workflow especially using scripting or graphical modeler of GRASS[5].

The loading data into prepared table within command line tool of Hive is by means:

```
# Loading data stored in HDFS to the table
LOAD DATA INPATH '/data/europe_latest_fix.csv' OVERWRITE INTO TABLE  europe;
```

After that command is executed and data are not any more stored in location */data/europe_latest_fix.csv* but moved to the Hive data warehouse which is by default */user/hive/warehouse/*.

3.2.3 Initialization of Hive UDFs

Hive supports user defined functions. Function can be defined permanently or temporary. Thus, to avoid repetition of defining functions for each hive session it is possible to create them permanently. In addition, for executing Hive commands/queries within GHF is that step mandatory.

[5] https://grass.osgeo.org/grass70/manuals/wxGUI.gmodeler.html

Within described configuration of cluster has been defined directory where Java libraries are stored. If configuration is different, the path to jar is initialized by execution:

```
# Initialize path to libraries for using its UDF's
add jar ${env:HIVE_HOME}/lib/ESRI-geometry-api.jar;
add jar ${env:HIVE_HOME}/lib/spatial-sdk-hadoop.jar;
add jar ${env:HIVE_HOME}/lib/json-serde-1.3.8-SNAPSHOT-jar-with-dependencies.jar;
```

However, this step can be skipped due to configuration of the cluster.

User defined functions(UDF) serves for access of the libraries within Hive like-SQL. As already mentioned initialisation of UDF function is expressed by:

```
# Cration of UDF functions
create function ST_Bin as 'com.ESRI.hadoop.hive.ST_Bin';
create function ST_Point as 'com.ESRI.hadoop.hive.ST_Point';
create function ST_BinEnvelope as 'com.ESRI.hadoop.hive.ST_BinEnvelope';

# Temporary funciotns
create temporary function ST_Bin as 'com.ESRI.hadoop.hive.ST_Bin';
create temporary function ST_Point as 'com.ESRI.hadoop.hive.ST_Point';
create temporary function ST_BinEnvelope as 'com.ESRI.hadoop.hive.ST_BinEnvelope';
```

For usage of GHF it is mandatory to create permanent functions. The execution of commands can be done using GHF's module *hd.hive.execute*.

In example below three function for spatial operation have been added. The constructor of bin - *ST_Bin* and constructor of point *ST_Point* and *ST_BinEnvelope* for returning bin envelope for given point.

3.2.4 Spatial Query

If all prerequisites for spatial query are satisfied next step is to create table where results of analyses will be stored.

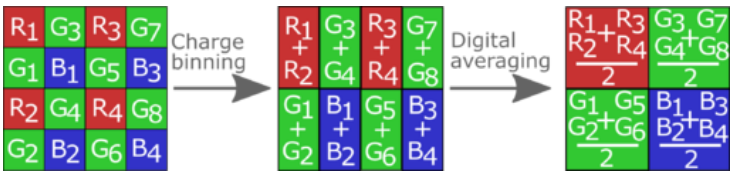


Figure 2.6: Use case of spatial binning for generalization of raster. Source: <https://web.stanford.edu

Spatial binning Spatial aggregation is useful in summarizing big data to gain a meaningful view on map patterns. Spatial aggregation works by creating square bins of a user specified size.

From shown 2.7 visualisation of OpenStreetMap it is hard to recognize any specific information. For classification and visualisation this amount of data no desktop GIS tools is available. The possible approach use aggregation of data into bins, and visualises map of reasonable amount of data using desktop GIS.

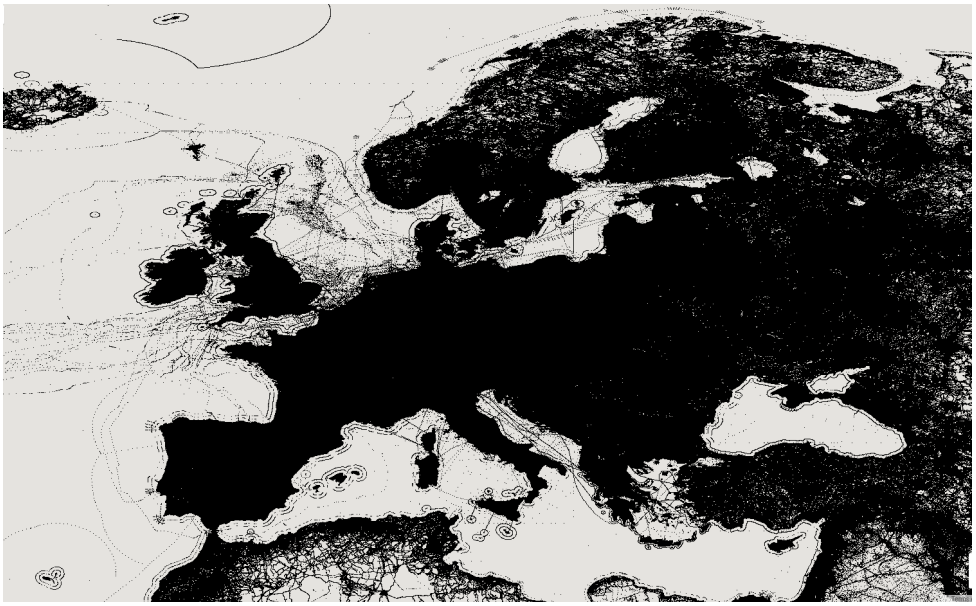


Figure 2.7: Snap shot of 1.3 billions of points from OpenStreetMap Planet dataset
Source: <http://spatialhadoop.cs.umn.edu/datasets/osm2/all_nodes.pyramid/

Let's create table for output of spatial binning. Notice, that for serialization of data is used library of ESRI framework. The last line consists from class for replacing key by null before feeding the writer of output format.

```
# Creation of table for results of binning
CREATE TABLE europe_agg2014(area BINARY, count DOUBLE, bin_id BIGINT)
ROW FORMAT SERDE 'com.esri.hadoop.hive.serde.JsonSerde'
STORED AS INPUTFORMAT 'com.seri.json.hadoop.UnenclosedJsonInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat';
```

In table named europe_agg2011 there will be stored result of spatial binning of points contribution between June 2010 and June 2011. Analogically creation of table within GHF is developed module *hd.hive.json.table*.

```
# Creation of table using  GHF
hive.json.table driver=hiveserver2
                    table=europe_agg2014
                    attributes="area BINARY, count DOUBLE, bin_id BIGINT"
                    serde=com.ESRI.hadoop.hive.serde.JsonSerde
                    outformat=org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
                    stored=com.ESRI.json.hadoop.UnenclosedJsonInputFormat
                    -e -d
```

As has been created table for results, similarly have been created tables for each year between the beginning of project till 2014, so 2006, 2007, ..., 2014.

The next and last stem is execution of spatial query for binning. The command firstly create bin (ST_Bin) for each point. Bin is not special datatype but is represented as polygon simple feature which is in ESRI word called ring or *esriGeometryRing*. Than, function *ST_BinEnvelope* create uniform cells-based polygon of defined size. Actually, the function returning bin envelope for given bin ID or point. The *GROUP BY* statements provide aggregation.

```
#Spatial binning for given time interval
FROM (SELECT ST_Bin(0.2, ST_Point(lon,lat)) bin_id, time
FROM europe
WHERE time > '2013-06-00 00:00:00' AND time < '2014-06-00 00:00:00') bins
INSERT OVERWRITE TABLE europe_agg2014
SELECT ST_BinEnvelope(0.2, bin_id) shape, COUNT(*) count, bin_id
GROUP BY bin_id;
```

Firstly, are created bins which are within the time interval and than are aggregate. The size of bin was defined as 0.2 degrees which is approximately 22km.

The execution of query can be made from *hiveserver2* command line or using *hd.hive.execute*.

Transfer data to GRASS If Map and Reduce processing finish. Data are stored in table *europe_agg2014*, which is located in hdfs://user/hive/warehouse/europe_agg2014. The folder consists several files with 000000_0, 000001_0 etc. Each file represents data block of HDFS.

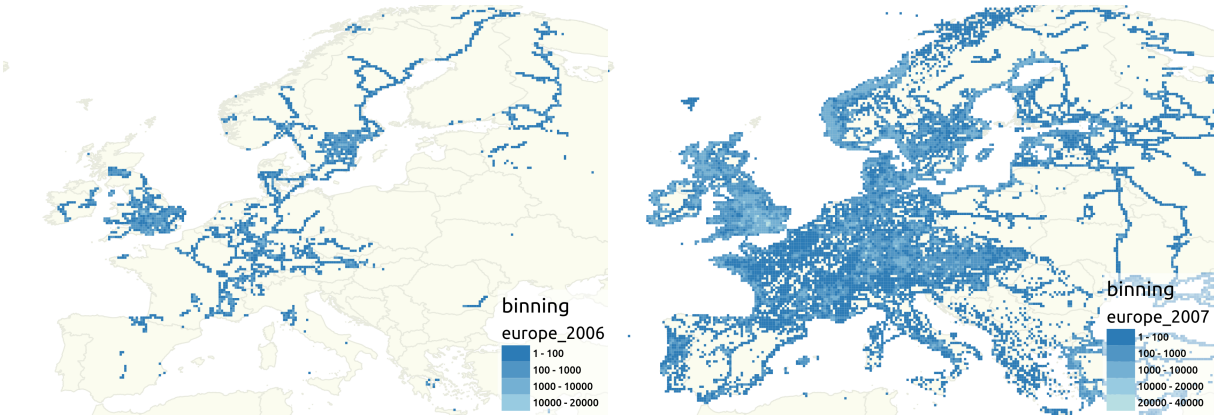
```
#Spatial binning for given time interval
hd.hdfs.out.vector.py driver=webhdfs
                      out=europe_agg2013
                      attributes='count int,bin_id int'
                      table=europe_agg2013
```

Using module *hd.hdfs.to.vector* is provided transfer of data from DataNodes to local system. Data are temporary stored in temporal directory of GRASS. After transfer of data, each map block is converted to ESRI GeoJSON and exported as native GRASS vector map. In addition is build and printed GRASS command for merging split blocks to a contiguous map. This task is not automatic due to size block variability and further processing variation. Module allows two ways how locate data on HDFS. For automate way is parameter *table*. For this approach, user define name of table and hive driver[6] using Hql query 'describe formatted' recognize location of table in HDFS. Second way is to define HDFS path manually using parameter *hdfs*.

Now, the spatial analyses using GHF and ESRI Spatial Framework for Hadoop is done and result can be processed within GRASS GIS. In figures 2.8, 2.9 is shown visualization aggregated points into bins.

In addition, using the same precess patter was aggregate all points for whole history of OpenStreetMap project on the area of Europe. Thus, approximately 1.3 billions of points to bin resolution 0.1 degree. The map us in attachment F.

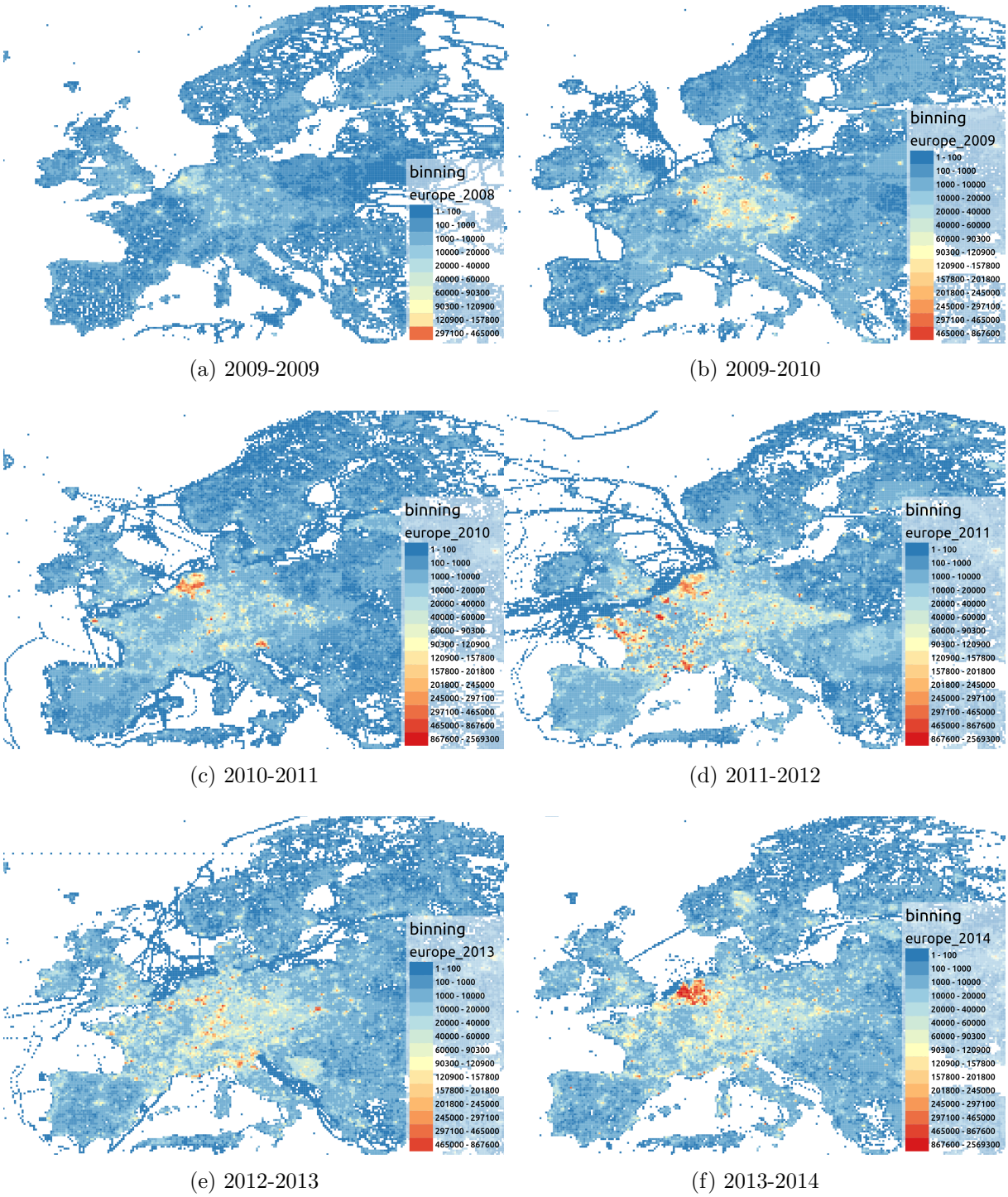
Summary of GHF usage



(a) 2006-2007 (b) 2007-2008

Figure 2.8: Aggregation of OSM points for particular 1 year interval. The resolution of spatial binning is 0.2 degree (approx 20km). The source OSM map consists of 1.3 billions of points.

[6]using hd.connect must be set connection to Hive



(a) 2009-2009 (b) 2009-2010

(c) 2010-2011 (d) 2011-2012

(e) 2012-2013 (f) 2013-2014

Figure 2.9: Aggregation of OSM points for particulars 1 year interval. The resolution of spatial binning is 0.2 degree (approx 20km). The source OSM map consists of 1.3 billions of points.

4 Conclusion

The first part of work introduces essentials and features of Hadoop design and its components. The following discussion aims on problems of spatial libraries coupled with Hadoop. The second chapter expands the theoretical principles by the application in practice and describes functionality, implementation and usage of developed GRASS Hadoop Framework in detail.

The first chapter briefly introduced Hadoop ecosystem and its main components: distributed filesystem, layer for parallel processing - MapReduce and architecture of Hadoop server components. Thereafter, detailed analysis of the functionality and implementation strategy of spatial libraries for Hadoop is provided. Following discussion deals with the challenges connected to spatial processing using MapReduce/HDFS and walk through particulars solutions for each library. The rest of the first part overviews Google Cloud Platform, especially services for cluster hosting.

Experiment framework consists of spatial processing workflow, description of implemented tools in detail and the usage. As the first step in the workflow tasks nascent within the setup of working environment and Hadoop configuration is described. Therefore, the configuration of cluster for developing software and performance of spatial processing is also described. After that, the functionality of developed framework for GRASS and implementation design is overview. The usage of developed framework is shown on the end.

Contribution

The main goal of the work was design and implement processing workflow for interaction between Hadoop and GRASS GIS, especially bring native access to distributed filesystem and data warehouse Hive within GRASS. This combination allows to use external libraries for parallel spatial processing.

As experiment within the work GRASS Hadoop Framework (GHF) has been developed. The package includes several GRASS command line modules for interaction between Hadoop and Hive. The full pack of modules simplifies and automates the processing workflow. Therefore provides conversion data to serialized format, transaction of data to distributed filesystem and load data to Hive data warehouse which supports two spatial

libraries. The result of highly demanding performance of spatial analysis by the GRASS Hadoop Framework provides transfer of results to local filesystem and conversion to the native GRASS vector map.

The selected case shown example of GRASS Hadoop Framework use. The Europe extraction of OSM Planet dataset consisting 1.3 billions of point has been processed on Hadoop from GRASS native interface. The developed framework meets expectations to be user friendly and effective tool.

Further work

Package GRASS Hadoop Framework compared with ESRI Hadoop GP Toolbox provides in many aspects same functionality, even more, like Hive support or manager for connections. On the other hand, ESRI supports executing Apache Oozie workflow for Hadoop even it is with several limitation. The further enhancement of GHF is supposed to be implementation of Oozie workflow support.

The design of GHF respects design layers based on interfaces hierarchy. The fact that core API is implemented independently on GRASS and middle interface is dependent only partly it is allows to bring developed code into another GIS project. In field of open source free GIS makes room for developing of similar framework for Quantum GIS(QGis). Due to design this task will consists only of implementation of GUI interface(QtGUI) and minor refracting of part of code which is connected to GRASS map conversion tools.

List of Figures

List of Tables

Acronyms

ACLs Access Control Lists

API Application Programming Interface

bdutil Command line script designed for managing

BOS Boundary optimized strip partitioning

BSP Binary split partitioning

IAM Identity and Access Management

CPU Central processing unit

CRC32C Cyclic redundancy check

CSV Comma-separated values

DNS Domain Name System

EMR Elastic Map Reduce

FG Fixed grid partitioning

GRASS Geographical Resources Analysis Support System

GCS Google Cloud Storage

GNU GPL GNU General Public License

GHF GRASS Hadoop Framework

GIS Geographic information system

GIT Version control system that is widely used software development

GPL General Public License

GUI Graphical user interface

HC Hilbert curve partitioning

HDFS Hadoop Distributed File system

CLI Comand line interface

HttpFS is a server that provides a REST HTTP gateway

IAM Identity and Access Management

IP Internet Protocol address

JDBC Java Database Connectivity

JSON JavaScript Object Notation

LISP Locator/Identifier Separation Protocol

MBB Minimal Bounding Box

MBR Minimal Bounding Rectangle

MPI Message Passing Interface

MWL Microwave links

NoSQL COM Structured Storage

OGC Open Geospatial Consorcium

ORM SQL Object-relational mapping

PD Extra persistent disk

pip Python Package Index

POSIX Portable Operating System Interface

RDBMS Relational database management system

REST API Representational state transfer Application Programming Interface

SerDe Serializer and Deserializer

SLC Strip partitioning

SQL Structured Query Language

SSH Secure Shell

STR Sort-tile-recursive partitioning

TBB Threading Building Blocks

UDF Universal Disk Format

URI Uniform Resource Identifier

URL Uniform Resource Locator

VM Virtual machine

WKB Well known binary

XML Extensible Markup Language

YARN yet another resource negotiator

Bibliography

[1] John Gantz, David Reinsel. *THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadow and Biggest Growth in the Far East*, December 2012 <http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf

[2] WHITE, Tom. *Hadoop: the definitive guide. Fourth edition.* Beijing: O'Reilly, 2015. ISBN 14-919-0163-2.

[3] NEUMAN, B.C. a T. TS'O. *Kerberos. IEEE Communications Magazine*. 1994, 32(9). DOI: 10.1109/35.312841. ISSN 0163-6804. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=312841

[4] JACOX, Edwin H. a Hanan SAMET. *Spatial join techniques*. ACM Transactions on Database Systems. 2007, 32(1), 1-5. DOI: 10.1145/1206049.1206056. ISSN 03625915. URL: <http://portal.acm.org/citation.cfm?doid=1206049.1206056

[5] Ablimit Aji, Vo Hoang, Fusheng Wang *Effective Spatial Data Partitioning for Scalable Query Processing* 2015/9/3, Journal: arXiv preprint arXiv:1509.00910 <http://arxiv.org/pdf/1509.00910

[6] Ray, Suprio, et al. *Skew-resistant parallel in-memory spatial join.* Proceedings of the 26th International Conference on Scientific and Statistical Database Management. ACM, 2014.

[7] YOU, Simin, Jianting ZHANG a Le GRUENWALD. *Large-scale spatial join query processing in Cloud.* 2015 31st IEEE International Conference on Data Engineering Workshops. IEEE, 2015, , 34-41. DOI: 10.1109/ICDEW.2015.7129541. ISBN 978-1-4799-8442-8. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7129541

[8] GOMES, Abel J.P. A *Total Order Heuristic-Based Convex Hull Algorithm for Points in the Plane. Computer-Aided Design.* 2016, 70, 153-160. DOI: 10.1016/j.cad.2015.07.013. ISSN 00104485. URL: <http://linkinghub.elsevier.com/retrieve/pii/S001044851500113X

[9] Wirz, Alexander, Björn Knafla, and Claudia Leopold. *Comparison of Spatial Data Structures in OpenMP-Parallelized Steering.* HIGH PERFORMANCE COMPUTING SIMULATION (HPCS 2008) (2008): 31.

[10] ZHANG, Jianting, Simin YOU a Le GRUENWALD. *Parallel online spatial and temporal aggregations on multi-core CPUs and many-core GPUs.* Information Systems. 2014, 44, 134-154. DOI: 10.1016/j.is.2014.01.005. ISSN 03064379.URL: <http://linkinghub.elsevier.com/retrieve/pii/S0306437914000234

[11] AJI, Ablimit, Fusheng WANG, Hoang VO, Rubao LEE, Qiaoling LIU, Xiaodong ZHANG a Joel SALTZ. *Hadoop GIS:A High Performance Spatial Data Warehousing System over MapReduce.* 2013, 6(11), 1009-1020. DOI: 10.14778/2536222.2536227. ISSN 21508097. URL: <http://dl.acm.org/citation.cfm?doid=2536222.2536227

[12] ELDAWY, Ahmed a Mohamed F. MOKBEL. *SpatialHadoop: A MapReduce framework for spatial data.* DOI: 10.1109/ICDE.2015.7113382. ISBN 10.1109/ICDE.2015.7113382. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7113382

[13] WHITMAN, Randall T., Michael B. PARK, Sarah M. AMBROSE a Erik G. HOEL. *Spatial indexing and analytics on Hadoop.* Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - SIGSPATIAL '14. New York, New York, USA: ACM Press, 2014, , 73-82. DOI: 10.1145/2666310.2666387. ISBN 9781450331319. URL: <http://dl.acm.org/citation.cfm?doid=2666310.2666387

[14] FOX, Anthony, Chris EICHELBERGER, James HUGHES a Skylar LYON. *Spatio-temporal indexing in non-relational distributed databases.* 2013 IEEE International Conference on Big Data. IEEE, 2013, , 291-299. DOI: 10.1109/BigData.2013.6691586. ISBN 978-1-4799-1293-3. URL:

<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6691586

[15] GHEMAWAT, Sanjay, Howard GOBIOFF a Shun-Tak LEUNG. *The Google file system. ACM SIGOPS Operating Systems Review.* 2003, 37(5), 29-. DOI: 10.1145/1165389.945450. ISSN 01635980. URL: <http://portal.acm.org/citation.cfm?doid=1165389.945450

[16] DEAN, Jeffrey a Sanjay GHEMAWAT. *MapReduce: Simplified Data Processing on Large Clusters* 2008, 51(1), 107-. DOI: 10.1145/1327452.1327492. ISSN 00010782. URL:

<http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

[17] Matej Krejci, Bachelor theses *Analysis and vizualization of rainfall data from microwave links using GIS* <https://github.com/ctu-osgeorel-proj/bp-krejci-2014/blob/master/text/matej-krejci-bp-2014.pdf

[18] HDFS Permissions Guide, Apache Hadoop [online]. [2016-05-06]. URL: <https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsPermissionsGuide.html

[19] GIS Tools for Hadoop. ESRI GitHub [online]. [2016-03-18]. URL: <http://esri.github.io/gis-tools-for-hadoop/

[20] WebHDFS REST API, Apache Hadoop [online]. [2016-05-06]. URL: <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/WebHDFS.html#Append_to_a_File

[21] Amazon EMR. Amazon [online]. [2016-03-18]. URL: <https://aws.amazon.com/elasticmapreduce/

[22] Apache Hadoop main. Apache Hadoop [online]. [2016-03-18]. URL: <http://hadoop.apache.org/

[23] Apache Nutch. Apache Hadoop [online]. [2016-03-18]. URL: <http://nutch.apache.org/#News

[24] Apache Hadoop news. Apache Hadoop [online]. [2016-03-18]. URL: <http://hadoop.apache.org/index.html#News

[25] Apache Hadoop news. Apache Hadoop [online]. [2016-03-18]. URL: <https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[26] Hadoop Rack Awareness. Apache Hadoop [online]. [2016-03-18]. URL: <https://hadoop.apache.org/docs/r1.2.1/cluster_setup.html#Hadoop+Rack+Awareness

[27] Google Cloud, Home Page [online]. [2016-04-18]. URL: <https://cloud.google.com

[28] Google Cloud, Hadoop Configuration [online]. [2016-04-18]. URL: <https://cloud.google.com/hadoop/setting-up-a-hadoop-cluster#setupscripts

[29] Apache Hadoop docs. Apache Hadoop [online]. [2016-05-18]. URL: <https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/YARN.html

[30] Apache Hadoop setting up Single cluster node, Apache Hadoop [online]. [2016-05-18]. URL: <https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/YARN.html

[31] Geoprocessing Tools for Hadoop, GitHub. [online]. [2016-04-18]. URL: <https://github.com/esri/geoprocessing-tools-for-hadoopl

[32] Protocol Buffers, Google Developers [online]. [2016-0-29]. URL: <https://developers.google.com/protocol-buffers/

[33] Snakebite documentation, Snakebite readthedocs [online]. [2016-04-29]. URL: <http://snakebite.readthedocs.io/en/latest/index.html

[34] Cluster properties, Google Cloud Platform [online]. [2016-04-29]. URL: <https://cloud.google.com/dataproc/concepts/cluster-properties

[35] Airflow Apache incubator project. Diff file is included in the attachmentH

[36] Applications of Parallel Computers (CS 5220), *Spatial binning and hashing*, Bindel, [2016-04-29]. <http://www.cs.cornell.edu/~bindel/class/cs5220-f11/notes/spatial.pdf

[37] JSON Formats, GitHub, [2016-04-29]. <https://github.com/esri/spatial-framework-for-hadoop/wiki/JSON-Formats

Attachment

A Attachment: Transformation of SQL to BigQuery

SQL data migration Within project there have been created several buckets for storing exported data from SQL. Data captured by microwave operator T-Mobile has been described in my bachelor theses[17]. Date are stored in relation database PostgreSQL and includes over $3e9$ rows. Due to big data characteristic there has been developed pythonG script for exporting data in defined time interval. It protects a database from overloading by queries for all rows in one query.

VM instance from Google Compute engine was used as a machine for exporting, merging and uploading data to bucket.

```
#create compute engine with 500 GB persistent disk
$ gcloud compute --project "spatial-hadoop" disks create "vmexport" --size "500"
\
--zone "europe-west1-b" --type "pd-standard" --image \
"/debian-cloud/backports-debian-7-wheezy-v20160418"
```

Table public.record from mwdb database has been exported in time step one week. For merge of each exported csv files Unix *cat* command has been applied. The size of exported table record is around 98 GB or 17 GB tarball.

```
#create bucket
$ gsutil mb -l EU gs://mwdata_export
#copy big file with slicing
$ gsutil -m -o GSUtil:parallel_composite_upload_threshold=100M cp \
mwdump.csv gs://mwdata_export/
```

BigQuery is a top performance service based on distributed database. The main characteristics are petabyte scale, low cost and unnecessary maintenance. BigQuery interface is similar to SQL databases, where queries are alike. BigQuery is accessible by API in

column	datatype	mode
linkid	INTEGER	REQUIRED
time	TIMESTAMP	NULLABLE
rx	FLOAT	NULLABLE
tx	FLOAT	NULLABLE

Table 1: Table record

several languages an allows programmatic way for control. The pricing model is based on pay-as-you-go.

By use of BigQuery table1 accordingly to columns in CSV file has been created.

Table has been queried by several queries, which had impact on slowing down the performance of GRASS GIS module *g.gui.mwprecip*, which is developed within my bachelor theses. The queries was massively speed up. Below are few comparison of queries on PosgreSQL database on *geo102.fsv.cvut.cz* and on BigQuery scalable engine.

```
SELECT linkid, avg(rxpower) FROM record WHERE time >='2014-07-07 10:53:00'\
AND  time<='2014-07-27 10:53:00% \
group by linkid order by 1
```

- PostgreSQL: 206.8 second
- BigQuery: 2.3 second

```
SELECT min(time) FROM mw.record
```

- PostgreSQL: 2.3 millisecond (index on time)
- BigQuery: 2.4 second

```
select linkid,avg(rx-tx) as m from record group by linkid
```

- PostgreSQL: 91.5 minutes
- BigQuery: 4.2 second

B Attachment: Compilation Spatial Libraries

Compilation Below there are shown some important notes for compilation of used libraries, especially for compilation of libraries with Apache Maven [2] profiles.

Usage of ESRI Spatial Framework for Hadoop is dependent on package Geometry API Java. After cloning GIT repository with ESRI Spatial Framework for Hadoop the maven test after packaging can failed. According to issue tracker of the repository there is suggested to skip tests.

```
sudo mvn clean package -DskipTests
```

For packaging JsonSerde package profile must be choosen. According documentation it is clear that the profiles are set to Hadoop distribution project such a *Cloudera* and *HortonWorks*. Thus its not completely clear which profile should be used for different Hadoop versions.

For using Hadoop 1.x on Google Cloud works profile for Cloudera 4 (CDH4)

```
mvn -Pcdh4 clean package
```

For building package for Hadoop 2.x profile for Hortonworks Data Platform 2.3 was tested.

```
mvn -Phdp23 clean package
```

In attachment of this work there is bash script for automated deployment of packages. Above mentioned *maven* profile settings must be change according the used Hadoop version.

[2]Apache packaging system for Java `https://maven.apache.org/`

[2]Json SerDe for Hive <`https://github.com/rcongiu/Hive-JSON-Serde`

C Attachment: Installation of GRASS and GHF

Below there are listed dependency on external packages. In addition, in second paragraph there is described deployment of GHF in GRASS GIS.

Python dependency Project is written and tested in Python 2.7, with emphasis to 3.x compatibility. The GHF package uses several external Python libraries which must be deployed. For installation of Python libraries has been used Python Package Index (pip) which is command line tool, and provide installation from the *pip* repository.

```
pip install pyhs2 hdfs thrift urlparse jaydebeapi cryptography
pip install snakebite protobuf pydoop dagpype future
```

GRASS GIS installation Installation of GRASS GIS is well described on official website. The developed modules within the thesis are available in the GRASS GIS Addons public repository. GRASS allows by using module g.extension to fetch and install modules to the system.

```
grass$ g.extension hd.client
...
Fetching <hdfs.client> from GRASS GIS Addons repository (be patient)...
Compiling...
Installing...
Updating addons metadata file...
Installation of <hd.hdfs.client successfully finished
```

Above is shown GRASS console with successful installation of the package.

D Attachment: Serialization ESRI GeoJSON

To hold integrity with GHF modules, as library for storing data and serialization must be use *com.esri.json.hadoop.Unenclosed JsonInputFormat* or *com.esri.json.hadoop.EnclosedJsonInputFormat*. This serialize store data in format ESRI GeoJSON. The differences between enclosed and unenclosed JSON is that Enclosed format includes header.[37]

- Enclosed JSON include header of structure. This format not suits to serialization.

```
                {
                "geometryType": "ESRIGeometryPoint",
                        "spatialReference": {
                        "wkid": 4326
                },
                "fields": [
                {
                "name": "Id",
                        "type": "esriFieldTypeOID",
                        "alias": "Id"
                },
                {
                "name": "Name",
                        "type": "esriFieldTypeString",
                        "alias": "Name"
                }
                ],
                "features": [
                {
                "geometry": {
                        "x": -104.44,
                        "y": 34.83
                },
                "attributes": {
                        "Id": 43,
                        "Name": "Feature 1"
                }
                },
                {
                "geometry": {
                        "x": -100.65,
                        "y": 33.69
                },
                "attributes": {
                        "Id": 67,
                        "Name": "Feature 2"
                }
            }
        ]
}
```

- Unenclosed JSON is represented by dictionaries where each line consist one dictionary - vector feature.

```
{
"geometry": {
        "x": -104.44,
        "y": 34.83
},
"attributes": {
        "Id": 43,
        "Name": "Feature 1"
}
}
```

GDAL driver allows to exporting and read only Unenclosed GeoJSON, so editing of format before conversion must be made.

E Attachment: Serialization JSON

In this section are described some findings around JSON and serialization. In last paragraph is introduced generator of Hive schema from JSON source file.

SerDe from version Hive 1.2 is included in hcatalog core. Below is example of usage.

```
add jar ${env:HIVE_HOME}/hcatalog/share/hcatalog/hcatalog-core-0.12.0.jar;

CREATE EXTERNAL TABLE links1 (
type string,
properties map<string,string>,
geometry string
)ROW FORMAT SERDE 'org.apache.hcatalog.data.JsonSerDe'
STORED AS TEXTFILE;
```

This SerDe from *hcatalog* does't work properly with GeoJSON. Hcatalog SerDe not allows to parse array in array structures as string which can be useful for conversion to geometry datatype(for ESRI Spatial Framework)

```
#GeoJson input- cant pares array in array
{ "geometry": { "type": "LineString", "coordinates": [ [ 15.7912, 50.2392 ],\
[ 15.7451, 50.23 ] ] } }

#usage
select ST_GeomFromGeoJSON(geometry) from links1;
```

Solution is to use SerDe [3] . Compiling SerDe for Google Cloud must be with CDH4 mvn profile for Hadoop1.x. For Hadoop 2.x with profile for HortonwWorks Sandbox.

```
sudo apt-get install -y maven openjdk-7-jdk git  && \
git clone https://github.com/rcongiu/Hive-JSON-Serde.git && \
cd Hive-JSON-Serde && \
mvn -Pcdh4 clean package
Hive SerDe schema generator
```

The SerDe from openx library works well.

```
add jar ${env:HIVE_HOME}/hcatalog/share/hcatalog/hcatalog-core-0.12.0.jar;

CREATE EXTERNAL TABLE links1 (
type string,
properties map<string,string>,
geometry string
)ROW FORMAT SERDE 'org.apache.hcatalog.data.JsonSerDe'
STORED AS TEXTFILE;
```

and geometry constructor from ESRI *ST_GeomFromGeoJSON* is able to parse it.

[3] https://github.com/rcongiu/Hive-JSON-Serde

Hive schema generator from JSON For complex JSON structures is useful to use generator of scheme from JSON source.

1. install Scala [4]
2. Clone repository of hive schema generator

```
git clone https://github.com/strelec/hive-serde-schema-gen.git
```

3. Conversion

```
$ cd hive-serde-schema-gen
$ touch test.json && echo '"{ "type": "Feature", "properties":\
{ "cat": 546 }, "geometry": { "type": "LineString", "coordinates":\
[ [ 14.4881, 50.1503 ], [ 14.5033, 50.1508 ] ] } }"' > test.json
$ sbt "run test.json"
```

output:

```
CREATE TABLE data (
type VARCHAR(7),
properties STRUCT<cat: SMALLINT >,
geometry STRUCT<coordinates:
ARRAY<ARRAY<FLOAT>>
type: VARCHAR(10)
>
) ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.JsonSerde';
LOAD DATA LOCAL INPATH 'test.json' INTO TABLE data;
```

[4] http://www.scala-sbt.org/download.html

F Attachment: Transformation map from GRASS to HDFS

Although, the goal of the main analyses is to visualize heat map Europe, someone can be interested only to data within Czech Republic boundaries. In GIS, clip function makes this task trivial. Assume that we have in GRASS database polygon of Czech boundaries. Before we can load GRASS vector map to Hadoop, firstly connection must be set to the server. Module *hd.db.connect* allows it. Firstly we set connection to HDFS.

```
#connection using WebHDFS driver
$ hdfs.db.connect  driver=webhdfs \
conn_id=google \
login=matt \
host=cluster-4-m.c.hadoop-jakub.internal \
port=50070

...
Test <webhdfs> connection (is path exists: ls /)
True
```

If *True* is printed, the connection pass. If connection is established we can continue next steps. In cr-wgs84 GRASS dataset map *cr* covers Czech Republic 2. Goal of this step is to transform map from naive GRASS vector format to serialized GeoJSON. Module *hd.hdfs.in.vector* allow convert file and put it into HDFS hosted on server.

```
#copy vector map cr to hdfs://data
$ hd.hdfs.in.vector driver=webhdfs  hdfs=/data map=cr layer=1

...
File has been copied to:
path :
/data
```

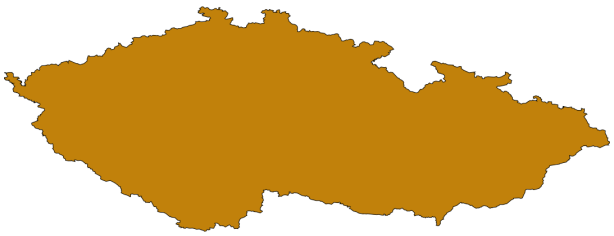


Figure 1: Czech Republic map for clipping the Europedataset.

Above, in this section, we copied all mandatory data to HDFS. The Europe dataset was prepared in cloud using performance VMs and loaded to HDFS of cluster. In contrast, Czech boundary map has been converted to serialized GeoJSON and copied directly from GRASS GIS DataNodes of cluster.

The next step is to prepare table which will store only data within Czech boundary.

```
CREATE EXTERNAL TABLE cr_europe(id BIGINT, lat DOUBLE, lon DOUBLE, time TIMESTAMP)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```

For analyzing relation between vector features, concretely if polygon A contains feature B, thus Czech Republic contains points is used user defined function *ST_Contains*

```
create temporary function ST_Contains as 'com.esri.hadoop.hive.ST_Contains';
create temporary function ST_GeomFromGeoJson as 'com.esri.hadoop.hive.ST_GeomFromGeoJson';
```

The exported GeoJSON file and its geometry can be constructed as binary geometry datatype using *ST_GeomFromGeoJson* UDF.

```
INSERT OVERWRITE TABLE cr_europe
SELECT europe.id, europe.lat,europe.lon,europe.time
FROM europe
JOIN cr
WHERE ST_Contains(ST_GeomFromGeoJSON(cr.geometry), ST_Point(europe.lon, europe.lat)) ;
```

And than form hiveserver2 console using hd.hive.execute can be executed query for clipping points.

Attachment: Europe map aggregation

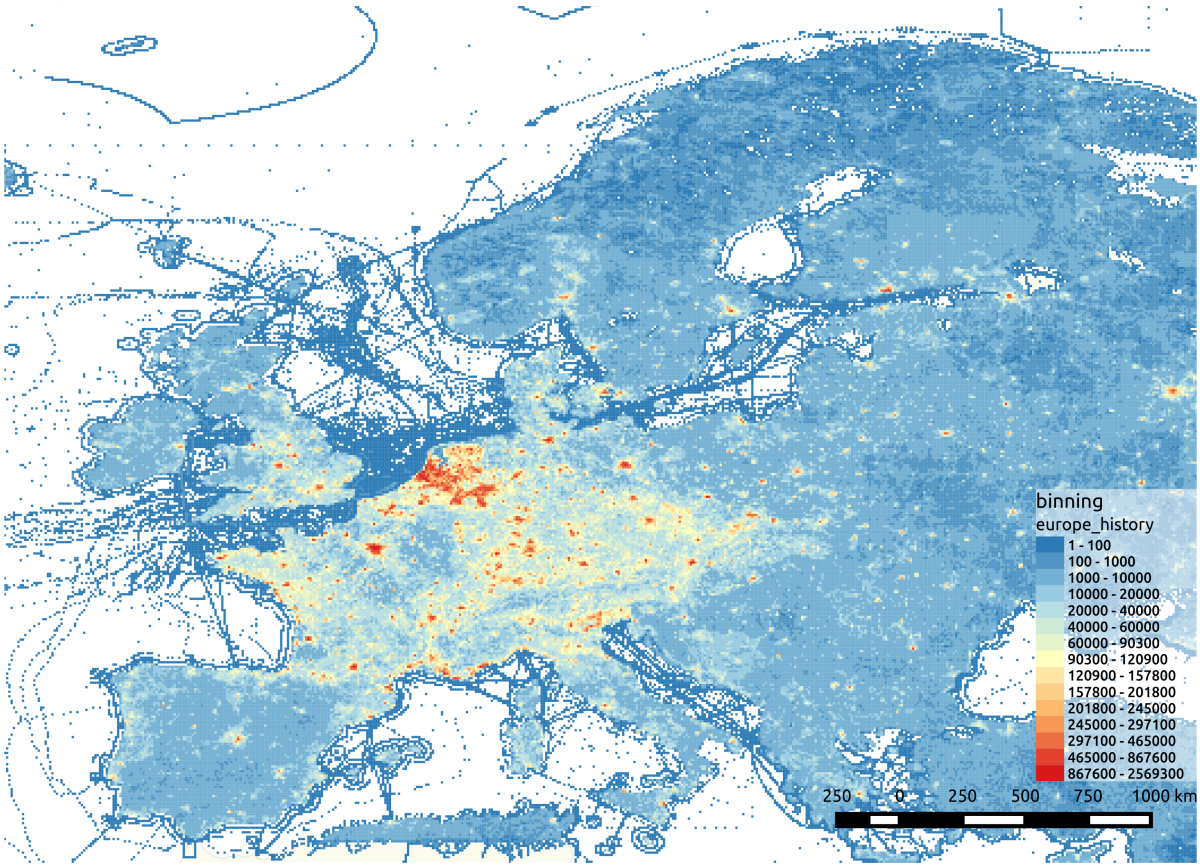


Figure 2: Aggregation of OSM points created between 2006 and end of 2014 The resolution of spatial binning is 0.1 degree (approx 10km). The source OSM map consists of 1.3 billions of points.

G Attachment: Configuration of bdutil and BigQuery migration

```
google_cloud/
|-- bdutil-1.3.4
|   |-- spatial1.sh
|   |-- spatial.sh
|-- bigquery
`-- mw_schema.json
```

H Attachment: Airflow diff

The difference file between Airflow and developed GHF.

```
.
|-- src
->      |   -- airflow.dif
|   `-- grass_hdfs
|       |-- hdfsgrass
|       `-- hdfswrapper
|-- text
`-- zadani
```

I Attachment: Source code of GHF

The CD with source code is attached to the printed version of the thesis.

```
.
|-- hdfsgrass
|   |-- grass_map.py
|   |-- hd.db.connect.py
|   |-- hd.esri2map.py
|   |-- hdfs_grass_lib.py
|   |-- hdfs_grass_util.py
|   |-- hd.hdfs.info.py
|   |-- hd.hdfs.in.fs.py
|   |-- hd.hdfs.in.vector.py
|   |-- hd.hdfs.out.vector.py
|   |-- hd.hive.csv.table.py
|   |-- hd.hive.execute.py
|   |-- hd.hive.info.py
|   |-- hd.hive.json.table.py
|   |-- hd.hive.load.py
|   `-- hd.hive.select.py
`-- hdfswrapper
        |-- base_hook.py
        |-- connections.py
        |-- hdfs_hook.py
        |-- hive_hook.py
        |-- hive_table.py
        |-- __init__.py
        |-- security_utils.py
        |-- settings.py
        |-- utils.py
        `-- webhdfs_hook.py
```