A SCALABLE WEB TILED MAP MANAGEMENT SYSTEM

MENELAOS KOTSOLLARIS

May 2017

TECHNICAL REPORT
NO. 309

A SCALABLE WEB TILED MAP MANAGEMENT SYSTEM

Menelaos Kotsollaris

Department of Geodesy and Geomatics Engineering
University of New Brunswick
P.O. Box 4400
Fredericton, N.B.
Canada
E3B 5A3

May 2017

PREFACE

This technical report is a reproduction of a thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering in the Department of Geodesy and Geomatics Engineering, May 2017. The research was co-supervised by Dr. Emmanuel Stefanakis and Dr. Yun Zhang, and funding was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Kotsollaris, Menelaos (2017). *A Scalable Web Tiled Map Management System.* M.Sc.E. thesis, Department of Geodesy and Geomatics Engineering, Technical Report No. 309, University of New Brunswick, Fredericton, New Brunswick, Canada, 146 pp.

ABSTRACT

Modern map visualizations are built using data structures for storing tile images, while their main concerns are to maximize efficiency and usability. The core functionality of a web tiled map management system is to provide tile images to the end user; several tiles combined construe the web map. This thesis presents a comprehensive end-to-end analysis for developing and testing scalable web tiled map management systems. To achieve this, several data structures are showcased and analyzed. Specifically, this thesis focuses on the SimpleFormat, which stores the tiles directly on the file system; the ImageBlock, which divides each tile folder (a folder where the tile images are stored) into subfolders that contain multiple tiles prior to storing the tiles on the file system; the LevelFilesSet, a data structure that creates dedicated Random-Access files, wherein the tile dataset is first stored and then parsed in files to retrieve the tile images; and, finally, the LevelFilesBlock, a hybrid data structure which combines ImageBlock and LevelFilesSet data structures. This work signifies the first time this hybrid approach has been implemented and applied in a web tiled map context. Specifically, each data structure was implemented in Java. The JDBC API was used for integrating with the PostgreSQL database. This database was then used to conduct cross-testing amongst the data structures. Subsequently, several benchmark tests on local and cloud environments are developed anew and assessed under different system configurations to compare the data structures and provide a thorough analysis of their efficiency. These benchmarks showcased the efficiency of LevelFilesSet, which retrieved tiles up to 3.3 times faster than the other data structures. Peripheral features and principles of implementing scalable web tiled map management systems among different software architectures and system configurations are analyzed and discussed.

ACKNOWLEDGMENTS

I wish to express my deepest gratitude to Dr. Stefanakis for all his guidance and support throughout my Master's degree.

My special thanks to Dr. Zhang for all his insights and advices during this research project.

I would like to express my sincere gratitude to William Liu, for his important contributions and recommendations during the design and implementation of the data structures mentioned in this research project.

Thanks to David Fraser, for his help and coordination during this research project.

I am thankful to Dr. Spacek, from the English Department in UNB, for his useful recommendations on writing this thesis. Also, thanks to Vasilis Pallikaras and Rachel Clark, for their helpful insights on reviewing this thesis.

Special thanks to Vasilis Drimilis and Vasilis Kontopoulos, for the motivation and mentorship they have provided me up until this day.

I am grateful to my family, for their constant support throughout my life.

Table of Contents

List of Tables

List of Figures

List of Symbols, Nomenclature or Abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BLOB	Binary Large Object
CRUD	Create, read, update and delete
CSV	Comma-separated values
GGE	Geodesy and Geomatics Engineering
GIS	Geographic Information System
GIT	Version Control System
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDD	Hard Disk Drive
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IT	Information Technology
JAR	Java Archive
J2EE	Java 2 Platform, Enterprise Edition
JDBC	Java Database Connectivity
JMH	Java Microbenchmarking Harness
JPG	Joint Photographic Experts Group
JVM	Java Virtual Machine
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MB	Megabyte
OGC	Open Geospatial Consortium
OOP	Object Oriented Programming
ORM	Object-Relational Mapping
OS	Operating System
OWS	OGC web services
PB	Petabyte
PNG	Portable Network Graphics
REST	Representational state transfer
TB	Terabyte
UML	Unified Model Language
UNB	University of New Brunswick
URL	Uniform Resource Locator
VCS	Version Control System
SQL	Structured Query Language
SSD	Solid State Drive
SVN	Apache Subversion
WMS	Web Map Service
WMTS	Web Map Tile Service
WTMMS	Web Tiled Map Management System
XML	Extensible Markup Language

1. INTRODUCTION

Web tiled map management systems have been developed and used for more than a decade. Popular vendors, such as Google, Microsoft, and ESRI, have been developing large scale mapping systems (e.g., Google Maps, Bing Maps and ESRI Maps) to visualize the world. However, despite public knowledge of the scale of these companies, very little is known about the architecture underlining these projects. Consequently, researching the efficiency and applicability of storing and retrieving tiles remains open to investigation. Anchoring the work on the most widely used techniques, the three most common solutions for web tiled map management system were implemented. These solutions include techniques which store the files directly to the File System or use the databases (Sample et al., 2013). This thesis presents an analysis of two file system solutions, the SimpleFormat and the ImageBlock, along with a data structure, named LevelFilesSet, which stores the tiles into Random-Access files (Housel, 2003).

File systems are complex in nature and have significant differences (e.g., NTFS and ExFAT; Microsoft, 2013). In implementing a system using a file system solution, there are decisions that will decrease its scalability. As a prime example, the system will be compatible with a specific operating system and its accompanying file system. This inflexibility denies the freedom of cross-platform applicability when using a single solution. In addition, file system-based solutions are complex and time consuming, and they significantly decrease the system's maintainability. In contrast, the LevelFilesSet data structure is not tied to any operating or file system. Instead, the LevelFilesSet improves performance and accessibility for the developer adopting the solution.

1.1. Research Questions

This thesis predominantly attempts to determine which data structure provides the best results for accessing and handling a tile dataset. On a second level of analysis, the thesis investigates whether the data structure is flexible and scalable across several systems as well as easy-to-use for the developers who adopt the solution.

1.2. Research Objectives

The objectives of this research are fivefold:

- To adopt an existing data structure for managing a tile dataset, named LevelFilesSet.
- To compare the LevelFilesSet with two file system based solutions, the SimpleFormat and the ImageBlock.
- To design and implement the LevelFilesSet data structure.
- To render the LevelFilesSet data structure easy-to-use for other developers.
- To present the systems engineering of the web tiled map management system.

1.3. Thesis Organization

The remainder of the thesis is outlined as follows: Chapter 2 discusses the literature review which shows the relevant research that was helpful for this thesis. Chapter 3 provides an analysis of the existing data structures for storing and retrieving a tile dataset. Chapter 4 describes the LevelFilesSet and its unique functionalities. Chapter 5 showcases the benchmarks ran locally and provides additional features that will enrich the capabilities and efficiency of the system. Chapter 6 showcases the benchmarks ran on Google Cloud. Chapter 7 highlights the software architecture of the system and the reasoning behind the decisions. Finally, Chapter 8 highlights the conclusions of the thesis.

1.4. Methodology

The methodology followed is shown in Figure 1.1. Each of the steps listed will be described in detail in subsequent chapters of this thesis.

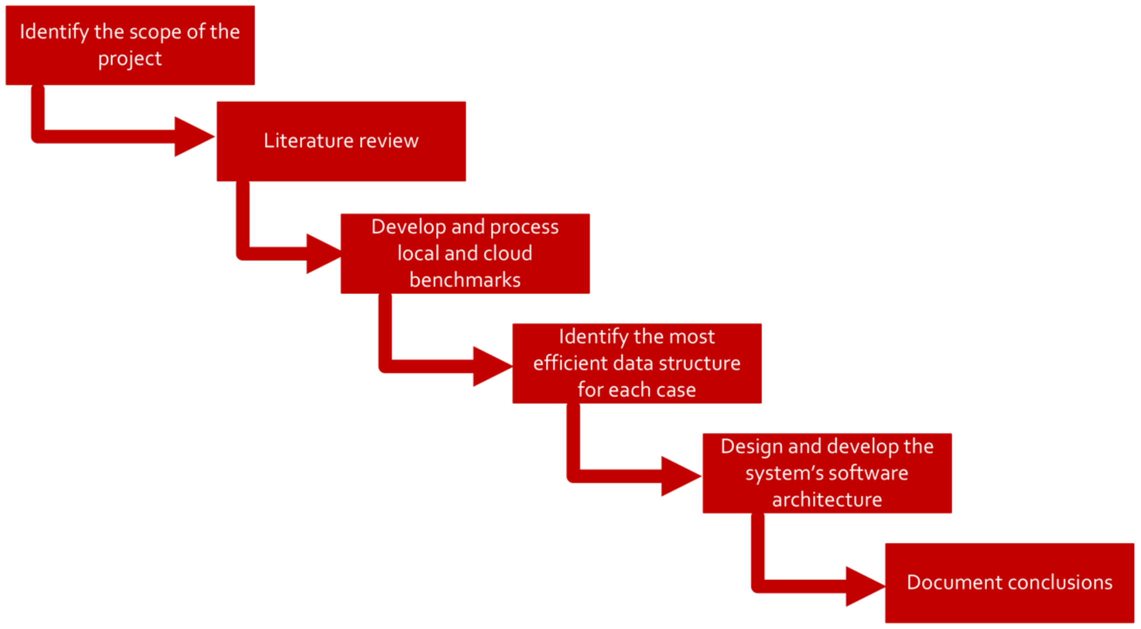

Identify the scope of the project
Literature review
Develop and process local and cloud benchmarks
Identify the most efficient data structure for each case
Design and develop the system's software architecture
Document conclusions

Figure 1.1 The workflow

- Identify the scope of the project: Select the areas and use cases in which this research will be focused.
- Literature Review: Investigate research on the topics.
- Develop and process local and cloud benchmarks: Perform benchmark tests, on both local and cloud environments, in order to examine the proposed data structures' efficiency.
- Identify the optimal data structure for each case: Based on the benchmarks results, identify which is the best data structure for each case and make it easy-to-use for other developers.
- Design and develop the system's software architecture: Implement the system's data structure for storing and retrieving tile images.

- Document conclusions: Document the conclusions derived from the analysis of this thesis and identify other topics of interest and potential future research.

Initially, the scope and the objectives of the project will be identified. These objectives are based on the goals of the project and on other factors such as time constraints and project orientation. After the scope of the project is defined, the literature review will be examined to analyze the work of the other researchers in the field. Several topics, including web tiled map management system (WTMMS) development, scalable systems, efficient data structures, web and software architecture, backend and frontend caching and so on are examined to determine the orientation of the project. The next step is to run several benchmark tests for the proposed data structures to examine their efficiency on both storing and retrieving tile images. Several algorithms must be designed, analyzed and implemented and the end goal must be considered, which is to determine which data structure can be used in a real WTMMS. It is important to highlight that benchmark efficiency does not necessarily make the data structure optimal. In this research, scalability, which is the ability of the system to adapt to changes as the project evolves and to be easily modifiable across different systems, is considered. The data structure must be able to maintain large volumes of data regardless of the environment configurations of the system. The cloud component must also be taken into account and analyzed too. Google Cloud is among the vendors which offers tile storage and retrieval functionalities and thus it is important to examine its efficiency. On a further level of analysis, the software engineering principles under which the data structures are implemented must be highlighted and explained in extensive detail; a system is a summation of coding principles and thus the design patterns applied in the low level of the implementation have a significant impact on the system’s performance and scalability. System engineering is an ongoing process that progresses over time. Important concerns and future suggestions are crucial for the lifecycle of the project (Rajlich et al., 2000). In the end, potential features and further suggestions are proposed and discussed for future research.

2. LITERATURE REVIEW

Initially, Sample and Ioup (2010) mention in their book which core patterns must be followed in order to implement a web tiled map management system. Their research is focused on many areas, such as how a tile image dataset can be created by having only a high resolution image, what kind of data structures should be used for the tile storage, and a comparison analysis between the more famous techniques of storing files. They examine different solutions, such as storing tile images on a database and storing them on the file system, and how the tiles can be served on the client side in order to make it possible for the user to have an interactive map and run specific queries. This is the most relevant recent research in the existing literature and this paper will be heavily based on this source.

Another important topic is web caching, which is the summary of rules that we apply in order to categorize which files are more important than others. After categorizing whether a file is important or not, we can save it accordingly to the proxy server. Thereafter, if another user requests the same file that exists on the proxy server, then the proxy server can provide the result without having to forward the request to the origin web server and that saves a lot of time.

Sample and Ioup (2010) conclude that while a page request does indeed reveal short-term correlations and other structures, a simple model for an independent request stream following the Zipf-like distribution is sufficient to capture certain asymptotic properties observed at web proxies. Karger et al. (1999), propose a new consistent hashing technique to provide significant performance improvements on web caching systems. In their conclusion, they demonstrate how their system handles locality issues, balances load among caches and possesses a high level of fault tolerance that is absent from other Web caching systems. In addition, Fan et al. (2000) first presented a scalable Wide-Area Web Cache Sharing Protocol based on measuring the overhead of the existing protocols using trace-driven simulations and a prototype implementation. They later compared it to existing protocols such as the internet cache protocol (ICP). Their study was based on an existing proposal called directory server which was implemented by Anderson et al. (1997). Barish and Broaczka (2002), present the taxonomy of the World Wide Web Caching Architectures and describe some of their design techniques.

Furthermore, they highlight the need of a survey paper in the area, and they present several types of Caching proxies (Reverse Proxy, Transparent and Forward). They also mention the importance of optimizing the Disk input and output and they do so by measuring the performance of each system (NTFS, FAT and so on) in correlation with the CPU workload. Based on the research conducted by Barish et al. (2002), Li and Zhu 2008, propose a server side caching system that supports efficient storage and retrieval of cached map image tiles. This efficiency is achieved by organizing the map image tiles of the same layer in the exact same file and saving the relationships between corresponding map image tiles on different zoom levels using memory-resident data structures and hard disc indices. Finally, they evaluate the performance by concluding that the new caching system can significantly reduce the hard disc access times needed to process a web map service request.

Quinn (2008) proposes a predictive model for frequently viewed tiles in a Web map. Quinn’s project aimed to explain what server side caching is and how it has become a popular technique for developing web maps; present a predictive model for determining popular areas anticipated to be in high demand and describe ways the model could be used, evaluated, and updated. Quinn concludes that maps with many scales might require multiple iterations of the model to be most effective and that the server administrator can choose how many tiles to create at each scale level of the cache, and may use several runs of the model when moving from small to large scale.

In their work, Nagappan et al. (2008) assess the effects of Test-Driven Development (TDD). More specifically, they built a non-trivial software system based on a stable standard specification using a disciplined, rigorous unit testing and build approach based on the TDD practice. TDD takes a different approach to engineering a system. TDD aims to develop scalable and reusable code that makes systems scalable and easily deployable across different environments. In their research, they conclude that the introduction of TDD led to 50% reduction of total written code when compared with an experienced team who used a different approach for system engineering. They achieved these results with minimal impact on developer productivity. This test is the basis for quality checks and serves as a quality contract between all members of the team. It is highly recommended that more teams adopt TDD in their environments.

Cloud computing usage has recently increased and more developers are adopting such solutions, according to Armbrust et al. (2010). The biggest benefit of adapting to new solutions such as cloud computing comes from the fact that developers with large enterprise systems are no longer required to invest large capital for hardware and other resources, but instead, vendors such as Google Cloud and Amazon S3 can be used. By switching to cloud computing, developers have to be concerned only with the deployment and component configurations, and not with other issues such as hardware repairing, building servers from scratch, and continuous monitoring of the servers.

Rajkumar et al. (2008), describes the vision of 21st century cloud computing and identifies various computing paradigms that promise to deliver the future of computing utilities. The definition of the high-level market-oriented cloud architecture can be seen in Figure 2.1.

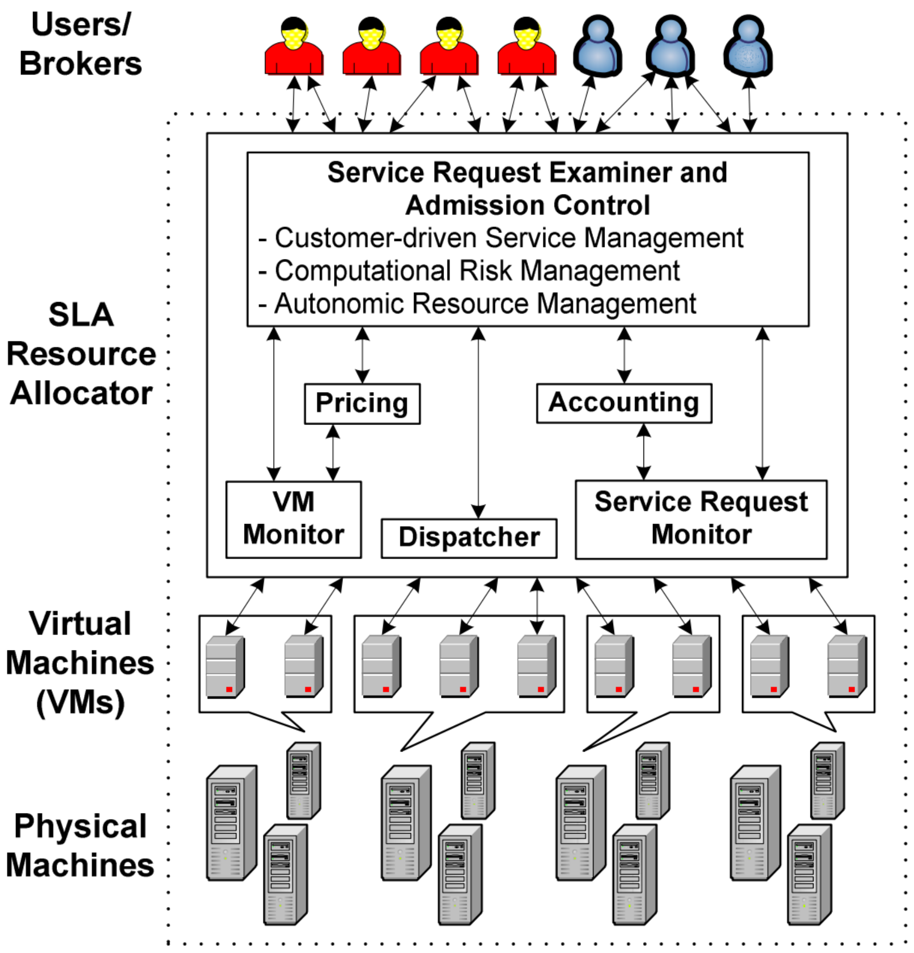

Users/ Brokers
SLA Resource Allocator
Service Request Examiner and Admission Control
- Customer-driven Service Management
- Computational Risk Management
- Autonomic Resource Management
Pricing
Accounting
VM Monitor
Dispatcher
Service Request Monitor
Virtual Machines (VMs)
Physical Machines

Figure 2.1 High-level market-oriented cloud architecture; taken from Rajkumar et al. (2008)

As can be observed, users use the cloud engines for different purposes (e.g., dispatching messages, service-request monitoring) and the messages are dispatched to further Virtual Machines (VM) where the computations take place. They define cloud computing as a new and promising paradigm delivering IT services for computing utilities. The same architecture could be extended to support web tiled map management systems and efficient tile retrieval.

In the subject matter of this thesis, the choice of file system has an important effect on the results and, particularly on low level computation.
Consequently, understanding how the file system operates is at the core of this project. Early on, Douceur and Bolosky (1999) analyzed a snapshot of data from 10,568 file systems of 4801 personal computers running Windows OS in a commercial environment, containing 140 million files totaling 10.5 TB of data. The purpose of their research was mainly to approximate the distributions of file size, file age, directory size, and directory depth They performed a series of benchmark tests and concluded that file sizes fit a log-normal distribution and that directory sizes fit an offset inverse-polynomial distribution. However, they did not address the performance of the New Technology File System (NTFS) over the others.

In other research, Resenblum et al. (1992) presented a new technique for a disk storage management system, called a log-structured file system. This type of file system writes all modifications to disk sequentially, and contains indexing information so that files can be read back with logarithmic efficiency. These authors presented a series of simulations that demonstrate the efficiency of this file system. The basic principle behind this structure is to collect large amounts of new data in a file cache stored in main memory, and then write the data to disk in a single large input/output (I/O) that can use all the disk’s bandwidth. Although complicated to implement, its performance is cost-effective.

Shvachko et al. (2010) describe the Hadoop distributed file system (HDFS), which is designed to reliably store very large datasets, and stream these data sets at high bandwidth to user applications. By distributing storage and computation across many servers, the resources can grow with demand while remaining economical at the same time. They concluded that their current clusters were less than 4000 cloud machines and they believe that with the architecture proposed in their paper, they can scale to a much larger volume of clusters with no problems.

In relevant work, Ghemawat et al. (2003) designed and implemented the Google File System (GFS). The purpose of their work was to render a scalable, well-distributed file system for large data-intensive applications; along with other applications, WTMMS falls under this category. In their research, they concluded that the GFS is a scalable and highly optimized file system capable of handling an extremely large number of datasets across different cluster-servers. The GFS is used in the Google Cloud, so this file system is most appropriate for storing and retrieving the type of tile dataset described in this thesis. In their research, they concluded that GFS is an important tool that enables Google to continue to innovate and tackle problems on the massive scale of the web. This, in turn, supports the fact that the GFS is able to store the dataset of the entire web.

Chung et al. (2003) gave emphasis to the ability of systems to evolve and maintain their interoperability features. Interoperability refers to the ability of the system to make use of the available information. The authors mention that web service computing poses significant challenges as developers determine how to leverage emerging technologies to automate individual applications, based on multiple software components. The Web has become the user interface of every entity around the world, and for this reason, Web services must offer interoperability through open standards, such as XML, SOAP, and REST.

While working through the parameters responsible for implementing WTMMS it is important to consider the relevant applications. Henning's (2000) research aimed to define the principles and guidelines under which algorithms should measure CPU performance, leading to the establishment of the Standard Performance Evaluation Corporation (SPEC CPU2000). The goal of SPEC is to develop technically credible and objective benchmarks to allow individuals to make decisions based on realistic and unbiased workloads. Henning also mentions that C++ applications are the biggest challenge to performance measurements because of their continuous version updates (Figure 2.2).

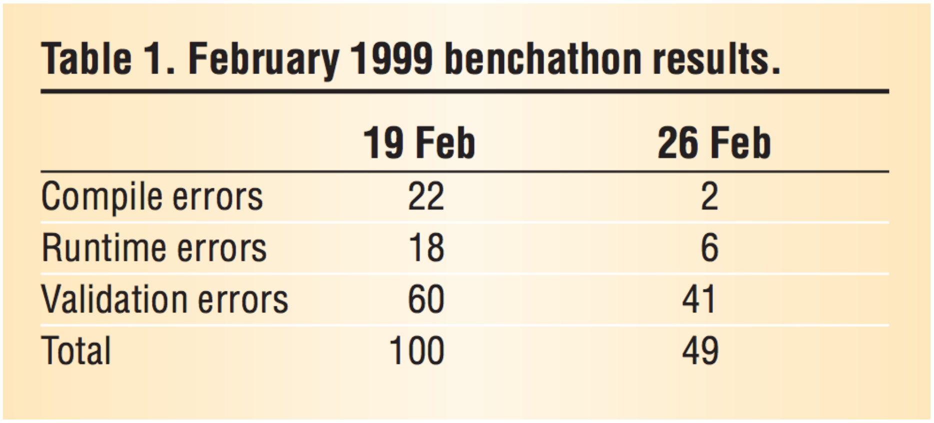


Table 1. February 1999 benchathon results.

	19 Feb	26 Feb
Compile errors	22	2
Runtime errors	18	6
Validation errors	60	41
Total	100	49

Figure 2.2 At that time, the benchathon solved just over half the outstanding problems. The point of a benchathon is to gather as many e project leaders as possible, platforms, and benchmarks in one place and have them work collectively to resolve technical issues

In this work, Henning admits that there is a lot more to discuss and define in the CPU benchmarking, and that the SPEC is the initial step for further investigation.

Based on Henning's research, Phansalkar et al. (2005), analyze the benchmark dependency on the system configuration. Although past efforts to identify subsets have primarily relied on using microarchitecture-dependent metrics of program performance, it follows that the results could be biased by the idiosyncrasies of the chosen configurations. Thus, the objective of their study was to present a methodology to measure the similarity of programs based on their inherent microarchitecture-dependent characteristics, which would make the results applicable to any microarchitecture. Based on a series of benchmarking scenarios, they concluded that the proposed methodology performed similarly across different platforms, in spite of their differences in configurations and environments.

Henning (2006) continued his research on CPU2006, working for Sun Microsystems, detailing a more advanced benchmark description than in his earlier research (CPU2000). The main examined programming languages were C++ and Fortran. In his more recent research, Henning performed again a series of unbiased benchmarking. The purpose of his paper was to present the most current curriculum in the field of CPU benchmarking; he achieved this by performing 30 benchmark tests on both integer and floating point benchmarks. He mentions that, although the benchmarks could have behaved better than the initial CPU2000 version, more research should be conducted to thoroughly investigate the behavior of the performed benchmarks.

Features other than CPU performance are critical to WTMMS. Wieczorek et al. (2004) mentioned a point-radius method for georeferencing tile images. This is useful for web tiled map management systems in that it supports features, such as searching for certain geographical areas. In their research, they described a method for georeferencing locality descriptions that accounts for the idiosyncrasies, sources of instability, and practical maintenance requirements encountered frequently. Their method minimizes the subjectivity involved in the georeferencing process and the results are consistent, reproducible, and allow for the use of uncertainty in analyses that use data.

The systems used to implement the ideas and complex data structures mentioned above are important factors when building a scalable web tiled map management system. Thus, when choosing among systems, it is important to identify the framework and the programming language that will provide such flexibility in implementation. Hundt (2011) compared the scalability of C++, Java, GO, and Scala. Although C++ performance was by far the best among the others, it also required the most extensive turning efforts, making it impossible for the average developer to realistically benefit from it. On the contrary, the Java version was the easiest to implement and the results-although inferior to C++ - satisfactory when taking into consideration Java's ease of use and accessibility.

To the best of my knowledge, apart from the Tile-Based Geospatial Information Systems book written by Sample and Ioup (2010), there is no other scientific research that is focused on Tile-Based Information Systems. However, the techniques applied in the peer-to-peer and web caching applications as well as the frameworks used, could be combined to develop an efficient web tiled map management system that can handle large volumes of data in an optimal and efficient way. In this light, the combination of the techniques mentioned in this literature review could be used to develop such a scalable web information system.

3. DATA STRUCTURES FOR TILE STORAGE

As the Figure 3.1 indicates, there are several ways of storing and retrieving image tiles:

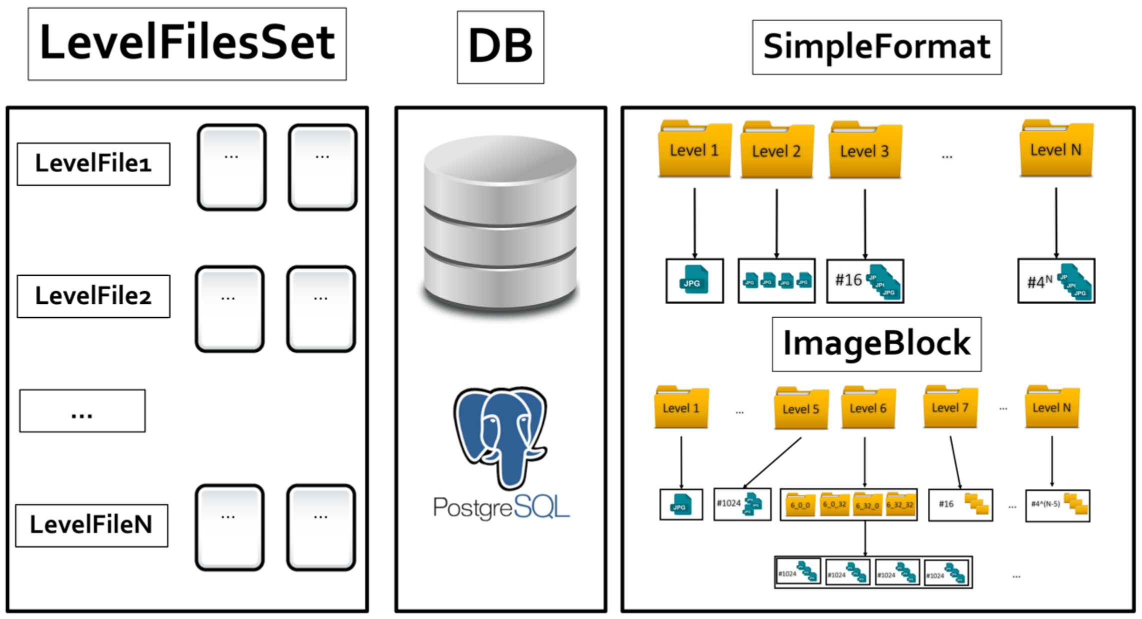


Figure 3.1 The LevelFilesSet, Database, SimpleFormat and ImageBlock data structures

The options for storing and retrieving are the SimpleFormat, the ImageBlock, Databases and the LevelFilesSet. This chapter analyzes the benefits and drawbacks of these options.

3.1. Tile scheme and zoom levels

The tiles are stored in folders, referred to as zoom levels. Each zoom level is expected to contain 4^k tile images, where k represents the folder number. For instance, the first zoom level will contain 4 images. As the zoom level increases, each tile is divided into 4 sub-tiles. The maximum number of columns and rows for each zoom level is $2^n - 1$, where n represents the number of zoom levels. For instance, in zoom level 5, the number of columns and rows will range between 0 and 31 (e.g., 5_0_0.jpg to 5_31_31.jpg; Z_X_Y.jpg, where Z represents the zoom level, X the column, and Y the row). In this research, tiles have the size of 256 pixels and are classified by their zoom level, column and row of the 2-dimensional map grid (Figure 3.2).

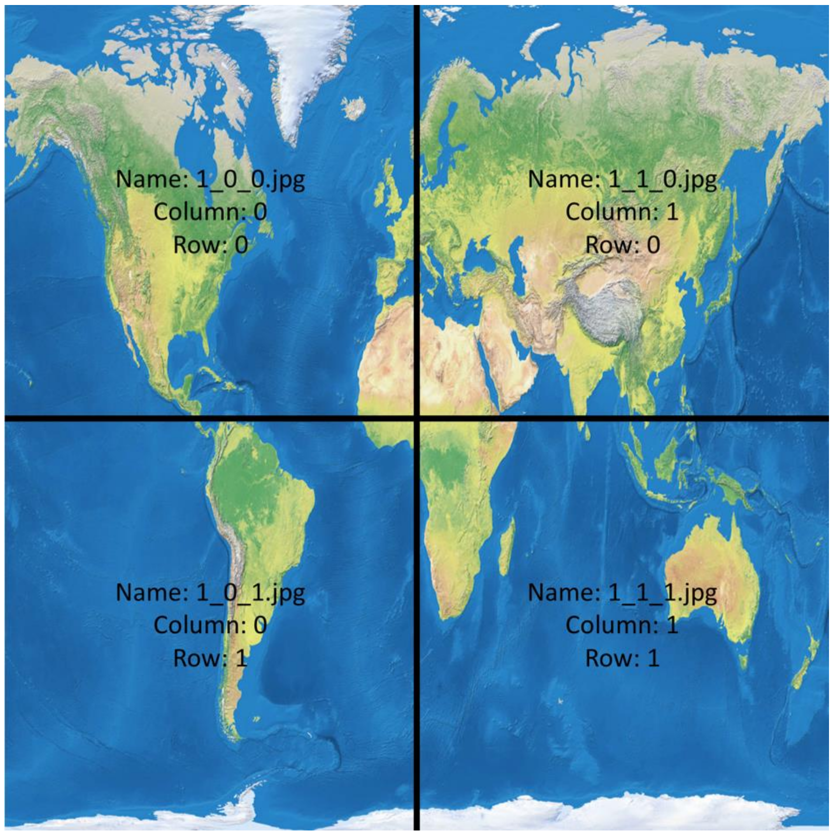


Figure 3.2 Tiling scheme for zoom level 1

The logical tile scheme is the foundational element of a web tiled map management system. In this research, tiles have the size of 256 pixels and are classified by their zoom level, column, and row of the 2-dimensional map grid. The logical scheme consists of mapping the address of the tile images to geospatial coordinates of the geographical area that the image covers. Google, Bing and Yahoo! Maps all use the tiling scheme mentioned in this thesis (Sample et al., 2010). This tile scheme renders computing the addresses of the tiles a trivial procedure and it is preferred over the others because of its simplicity and easiness to implement. As explained in MicroImages (2010), the Google Maps tile dataset structure is stored in each subdirectory as described below. Every tile is aligned on a fixed grid of Spherical Web Mercator projection (Stefanakis, 2014). This way, Google Maps can quickly and efficiently load millions of tiles (Figure 3.3). This hierarchical tile structure ensures that the tile dataset with the maximum possible resolution will never be able to exceed the maximum number of tiles or directories that the WTMMS can store, thus rendering it efficient and scalable across different systems.

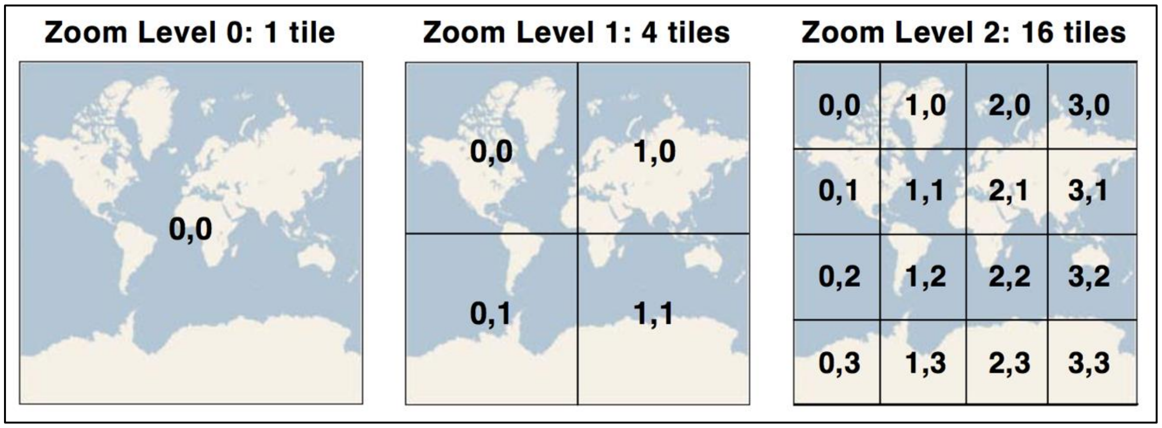


Figure 3.3 Global tile grids at the three Lowest Google Maps zoom levels, using the Spherical web Mercator projection, taken from MicroImages (2010)

3.2. Tile Dataset & dealing with missing tiles

In thoroughly validating the performance of the proposed solutions, the greater the number of the zoom levels, the higher the accuracy of the results. However, in reality, there are multiple missing datasets; thus, a dataset fulfillment algorithm has to be developed to address this problem. There are several different algorithms that can supplement the dataset with the expected tiles. The core consideration is the average size of the expected missing tile. The algorithm has to insert a tile that approximates the expected tile size, which is difficult to be predicted with high accuracy. An algorithm that can provide all the missing tiles without taking into consideration the tile size is illustrated in Figure 3.4.

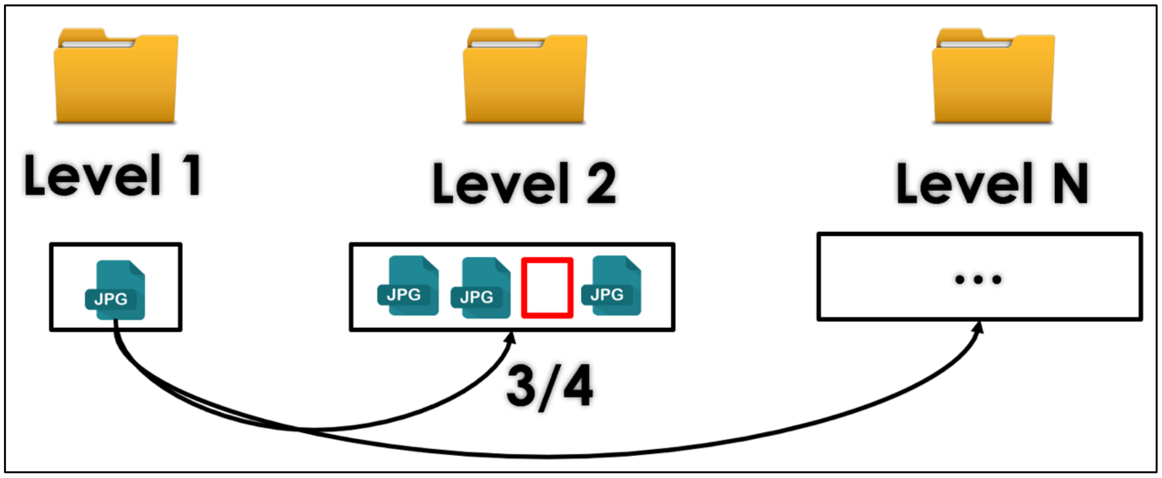


Figure 3.4 Imputing with the same tile

Since testing the performance is the primary concern and since the difference between reading 4KB and 16KB (or more) is negligible, by adding the same tile wherever a tile is missing can efficiently add all the expected tiles in the dataset.

An alternative approach would be to estimate the average size for each zoom level and then complete the dataset with the expected average tile. Inarguably, this is an extremely time consuming technique, considering the fact that that zoom levels increase exponentially with the number of the tiles (i.e., in higher zoom levels there are billions of expected tiles that have to be read). The algorithm's pseudocode is illustrated in Figure 3.5.

```
For each Level:
Representative_Tile repr_Tile; //the representative tile
    For each Column & Row:
        If(tile is Missing):
            Fulfil with repr_Tile;
        End If;
    End For each;
End For each;
```

Figure 3.5 Pseudocode for the accurate tile fulfilment algorithm

A graphical representation of how this algorithm operates is showcased in Figure 3.6.

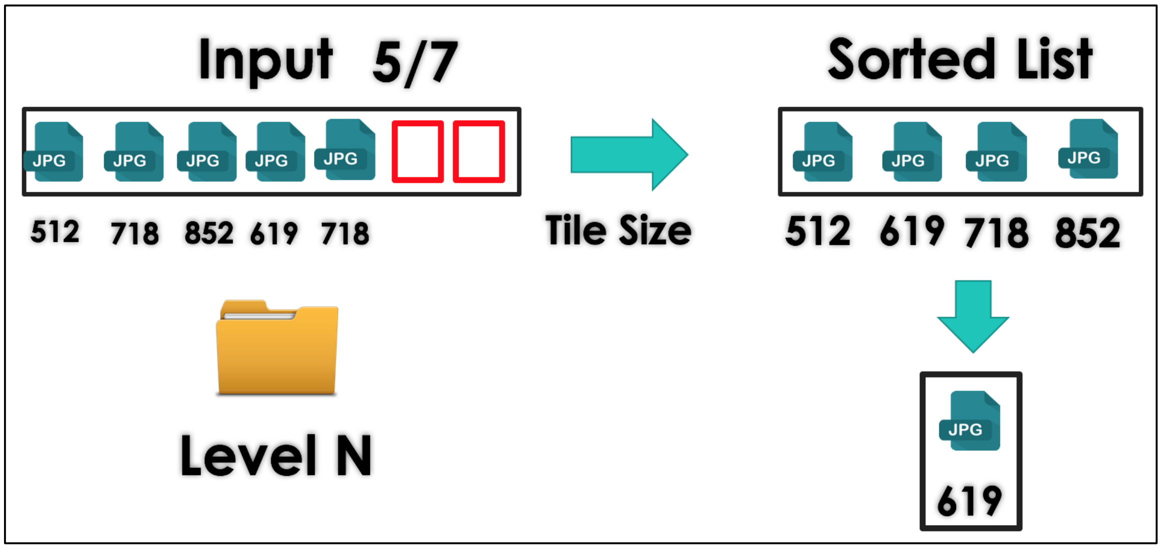


Figure 3.6 Accurately imputing the missing tiles based on the tile's size

For demonstration purposes, the first algorithm (Figure 3.5) is used for the tile dataset completion. More sophisticated algorithms can be used based on the normal distribution which depends on the size of the tiles (Dutilleul, P., 1999). These algorithms may better approximate the expected tile size but, as previously noted, are very time consuming, especially in the upper zoom levels (14 and higher).

3.3. SimpleFormat – File System

By following this data structure, the tiles are directly stored into folders segregated by zoom level. One of the advantages of this solution is the ease of its implementation and usage. However, this comes with drawbacks, including deciding which operating system and file system should host the LevelFilesSet. Consequently, a benchmarking across all the available file systems (e.g., NTFS, ExFAT, FAT32) must be performed to verify the best performance. This would certainly be a time-consuming practice. Moreover, this solution restricts the system regarding which operating and file systems can be used. For instance, if NTFS is proven to be the best performing file system, then this system will not be supported by any operating systems other than Windows. The ideal system would operate optimally regardless of the underlying operating system (Zhang et al, 2008). The main problem with using file system-based solutions is that each file system is operating system-specific. Thus, the very nature of a file system renders it inapplicable in the context of a multi-modal solution scaling across different operating systems.

3.4. ImageBlock – File System

Similarly to the SimpleFormat, ImageBlock structure operates in the File System where the tiles are stored in folders. In this case, however, each folder has an upper limit on the number of tiled images that can be accommodated. The maximum number of tiles that each folder can contain is 1024, so after the 6th zoom level, the folders will contain child-folders. For example, the 6th folder will contain 4 folders as depicted in Figure in Figure 3.7:

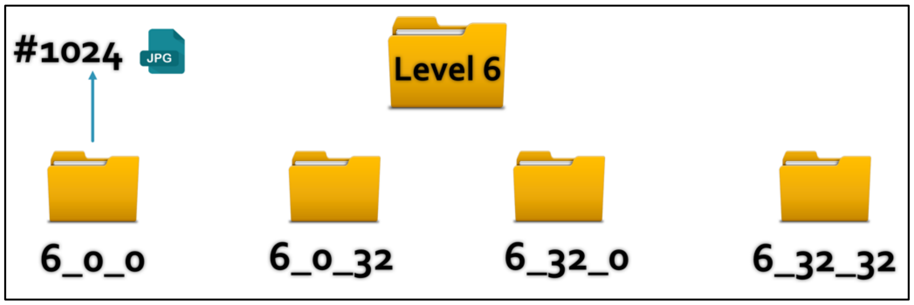
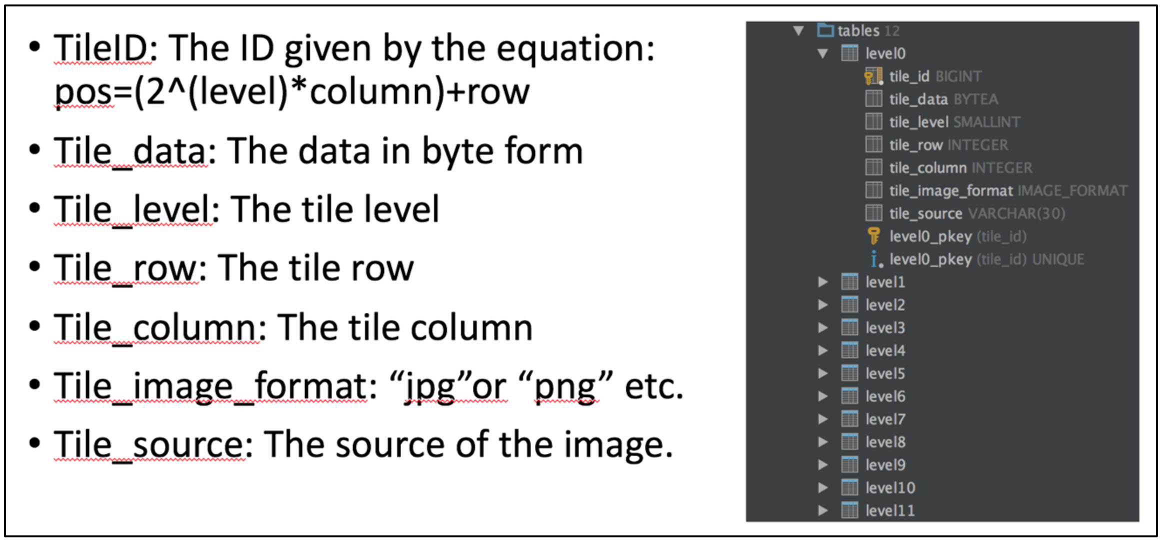


Figure 3.7 The ImageBlock format

For instance, tile 6_0_34 will be classified into 6_0_32 folder because it contains tiles with columns ranging from 1 and 31 and rows 32 and 63. As in the tile naming, the first number of the folder name represents the level, the second number represents the column, and the third represents the row of the tiles. This solution has the advantage of distributing the tiles in a more elegant way rather than lumping them under 1 folder, like SimpleFormat.

3.5. Databases

In the database solution, all the tiles are stored in tables according to each zoom level. The structure in which the tiles are stored is similar to either SimpleFormat or ImageBlock. Although databases offer multiple APIs that have been tested and used by several developers, as presented in Chapter 4, their performance for storing and retrieving tiles is the worst out of all the alternative solutions. The benefit of using the database solution is that they are very robust and scalable; however, when performance is the core factor of decision making, as in this case, this solution proves to be insufficient.

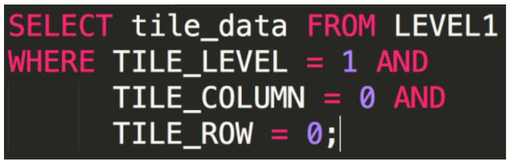


- TileID: The ID given by the equation: pos=(2^(level)*column)+row
- Tile_data: The data in byte form
- Tile_level: The tile level
- Tile_row: The tile row
- Tile_column: The tile column
- Tile_image_format: “jpg”or “png” etc.
- Tile_source: The source of the image.

```
▼ tables 12
  ▼ level0
      tile_id BIGINT
      tile_data BYTEA
      tile_level SMALLINT
      tile_row INTEGER
      tile_column INTEGER
      tile_image_format IMAGE_FORMAT
      tile_source VARCHAR(30)
      level0_pkey (tile_id)
      level0_pkey (tile_id) UNIQUE
  ▶ level1
  ▶ level2
  ▶ level3
  ▶ level4
  ▶ level5
  ▶ level6
  ▶ level7
  ▶ level8
  ▶ level9
  ▶ level10
  ▶ level11
```

Figure 3.8 The database schema following the SimpleFormat structure

The database schema showcased in Figure 3.8 follows the SimpleFormat structure. Each zoom level is stored as a table and each tile contains several attributes (e.g., tile_id, tile_source and so on). The tiles are saved as BLOB data type (Sears et al. 2007). The dataset can be queried by using a SQL query as described in Figure 3.9.



```
SELECT tile_data FROM LEVEL1
WHERE TILE_LEVEL = 1 AND
      TILE_COLUMN = 0 AND
      TILE_ROW = 0;
```

Figure 3.9 Requesting tile 1_0_0.jpg using SQL query in Postgres 9.3

The databases can inherit different schemas. For instance, the ImageBlock structure could be used for the structure of the tables. In this research, the SimpleFormat’s schema is used for testing purposes. Recently Mapbox, a GIS company, has came up with a format similar to those of the one described above, named MBTile format. The MBTile format uses a SQLite database to store tiles in one single table (Table Tiles) and has attributes similar to the database described above (column, row, BLOB, and so on). Similarly to the file system based solutions, databases bring a lot of unnecessary features that introduce significant overhead to the system. A tile storage system will not require the rich number of APIs featured in the database; instead only a handful of the APIs are necessary and thus databases are not able to efficiently retrieve tile images.

Databases are designed to manipulate a structured volume of data, such as characters and numbers. A tile storage system has little need for queries on structured data. However commercial systems, such as such as MapBox, use databases instead of other solutions. If the tile application required frequent update of tile images (from the user side), then the database would offer straightforward functionalities that would render tile storage and retrieval a lot more straightforward and less time consuming. Hybrid approaches, such as file system with database solutions would also be an option (e.g., LevelFilesBlock stored in the database, as mentioned in Chapter 4.3). In this research, Postgres 9.3, an open-source database which provides multiple APIs and further geospatial tools, is chosen for benchmark purposes. The Java framework provides the JDBC API (see following section), which allows data access from any relational database. The code can be found in the Appendix IV.

3.5.1 The JDBC API

The JDBC API is a Java API that can access any kind of tabular data and particularly data stored in the Relational Databases. It works as a “connector” between the application and the software (Hamilton, G et. al, 1997). Most of the IDEs provide several plugins that allow them to run database queries by their own interface (e.g. Intellij IDEA, Oracle etc.). This API is easy to use and it provides the following core interfaces and classes:

- DriverManager: This class is responsible for the list of database drivers.
- Driver: This is an interface which handles the communication with the database server.
- Connection: This interfaces with all methods for contacting a database.

- Statement: Used to access objects created from this interface to submit the SQL statements to the database.
- ResultSet: These objects hold data retrieved from a database after a SQL query is executed.
- SQLException: This class handles any errors that occur in a database application.

The architecture can be seen below (Figure 3.10):

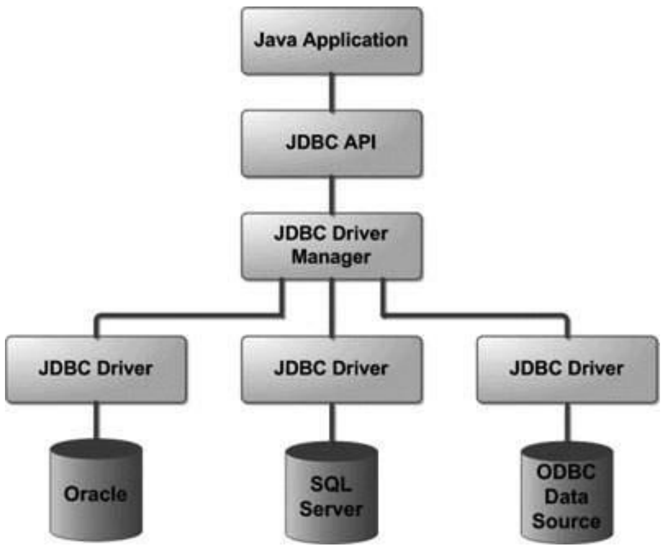


Figure 3.10 The JDBC Architecture (Java Oracle Documentation, 1995)

3.6. Parsing tiles instead of reading them from the file system

Instead of storing the tiles in the file system, a file which contains all the image information can be created and parsed accordingly whenever needed. In this solution, the complexity of the retrieval algorithm is not dependent on the file system. By using

Random-Access files, the tiles can be retrieved in constant time. The main drawback of the LevelFilesSet is that the memory complexity is larger than the previously mentioned solutions; this is because of the LookupFile which is needed to store the extra pointers to the TileDataset file. However, the overall performance of the LevelFilesSet, as discussed in the Benchmarks Results section, is better than the other solutions (Figure 3.11).

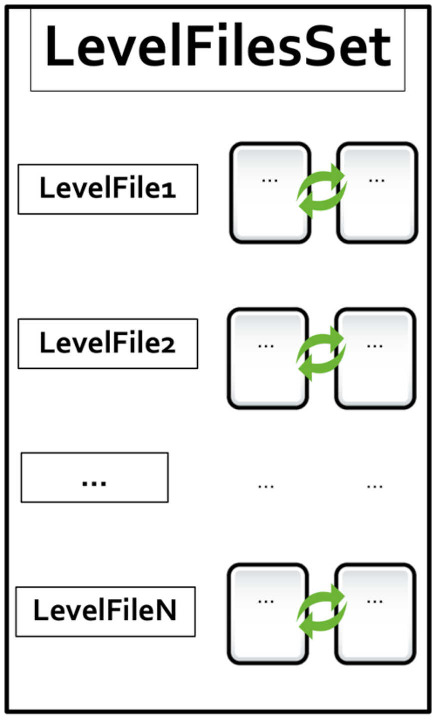


Figure 3.11 The LevelFiles for each zoom level

4. THE LEVELFILESSET

Instead of accessing the tile files by using the File System's inner functions, two files are created: The Lookup file and the TileDataset file. Each time a tile is requested, the Lookup file provides pointers that point to locations in the TileData file (Barish, et al, 2000). The TileData file holds the information about all the tiles in each zoom level (Figure 4.1).

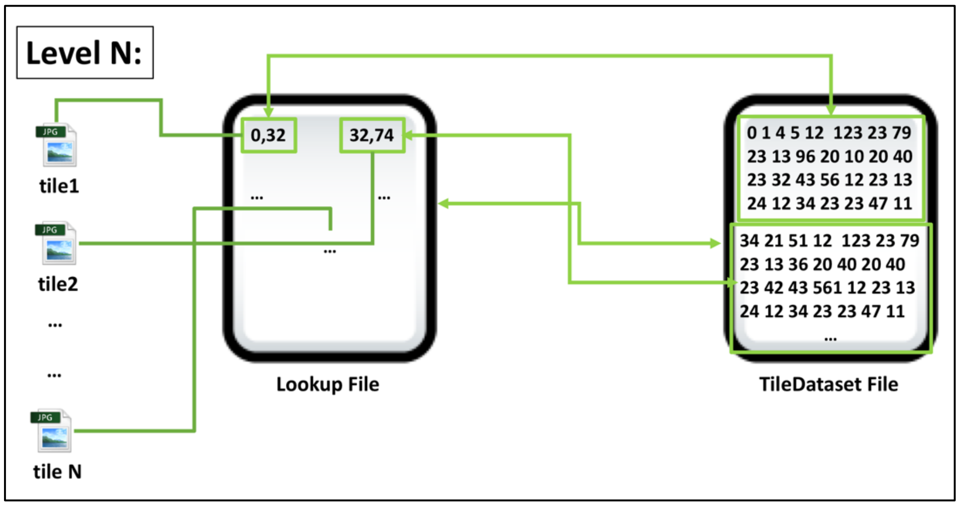


Figure 4.1 The communication of the Lookup File with the TileDataset File

4.1. The functionality of LevelFilesSet

The lookup file contains 2 variables:

- CursorPointer (8 bytes): Represents a pointer to the Data File.

- SizePointer (4 bytes): Represents the size of the tile.

Initially, the LevelFilesSet Dataset is generated by copying the bytes of the tile files on the LevelFilesSet per zoom level. The way the tiles are stored is based on the following formula:

$$position = ((2^{level} * column) + row) * byteLength \qquad (1)$$

where:

- position: the position at the Lookup file
- level: the zoom level of the tile
- column: the column of the tile
- row: the row of the tile
- byteLength: the length of the CursorPointer plus the SizePointer.

The Lookup file can be extremely large on the high zoom levels (e.g., zoom level 14 or higher). The maximum size of the TileData file can be 2^{64}-1 bytes; this number ensures that the data structure will be able to support higher zoom levels that contain billions of images. For example, the user wants to retrieve the tile on the 3rd zoom level, 2nd column and 1st row, based on equation (1), where position=204. This means that to retrieve the tile, the LookupFile must be parsed on the 204th byte of the *LookupFile*. The first 8 bytes of that position (bytes 204 until 211) will contain the pointer of the *TileData* file and the next 4 bytes (bytes 212 until 215) will contain the size of the tile. After retrieving the *CursorPointer* and *SizePointer*, the *TileData* file is parsed, starting at the *CursorPointer* and reading SizePointer bytes.

Although the implementation of the LevelFilesSet data structure is complex, the benefits are the following:

a) the system does not rely on the file system for reading a file

b) retrieval of any tile is achieved in constant time.

In other words, a file system on top of the existing file system is created and used for tile retrieval. This data structure offers the feature of tile generation, which is based on the pre-existing tiles structured that depends on the SimpleFormat structure. There are several other features that could be implemented in the future and could enrich the functionality of the LevelFilesSet to support other use cases and scenarios. For instance, one of these features could be the functionality of adding and deleting tiles dynamically. Tile-based management systems usually have to update their tiles within a specific time range. The current state of the data structure allows tile generation, which means that the LevelFiles have to re-generate every time a tile is updated. For supporting this feature, the implementation of the LevelFilesSet should be upgraded to a more sophisticated version, as proposed by Sample and Ioup (2010). By storing an extra file which keeps the pointers of each tile, the LevelFilesSet can modify (i.e., add or delete) each tile separately. However, this feature is expected to increase the memory needs of the systems since additional pointers must be stored apart from the tile dataset. Moreover, the performance is expected to decrease since an additional Seek and Read within the newly added file will be needed. This upgrade can be seen in the Figure 4.2.

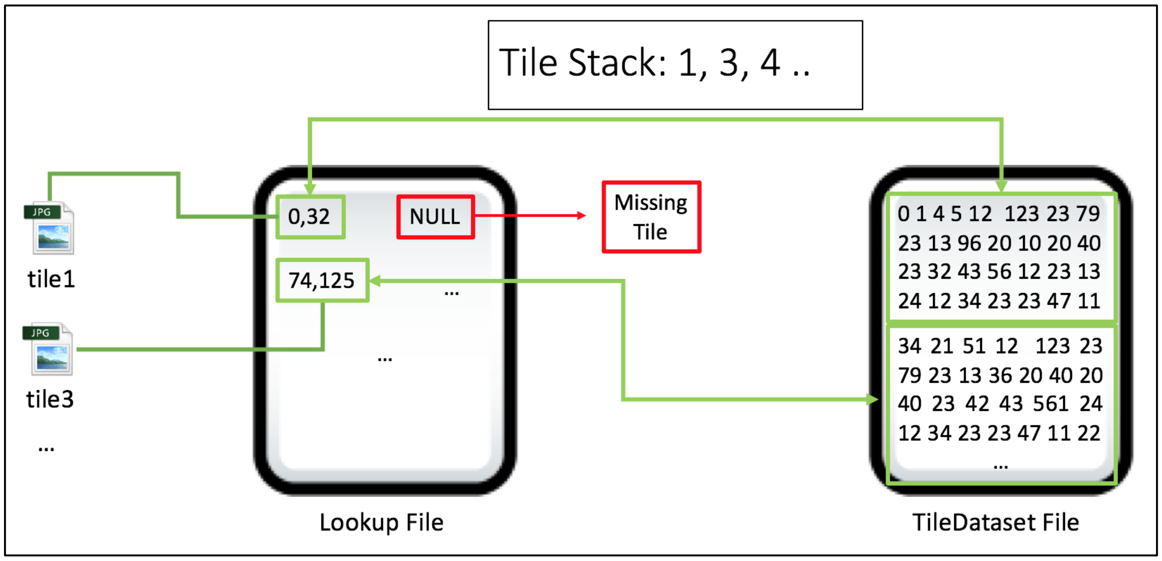


Figure 4.2 Version of the LevelFilesSet that allows missing tile indexes by using the tile stack which indicates the existing tiles within the system

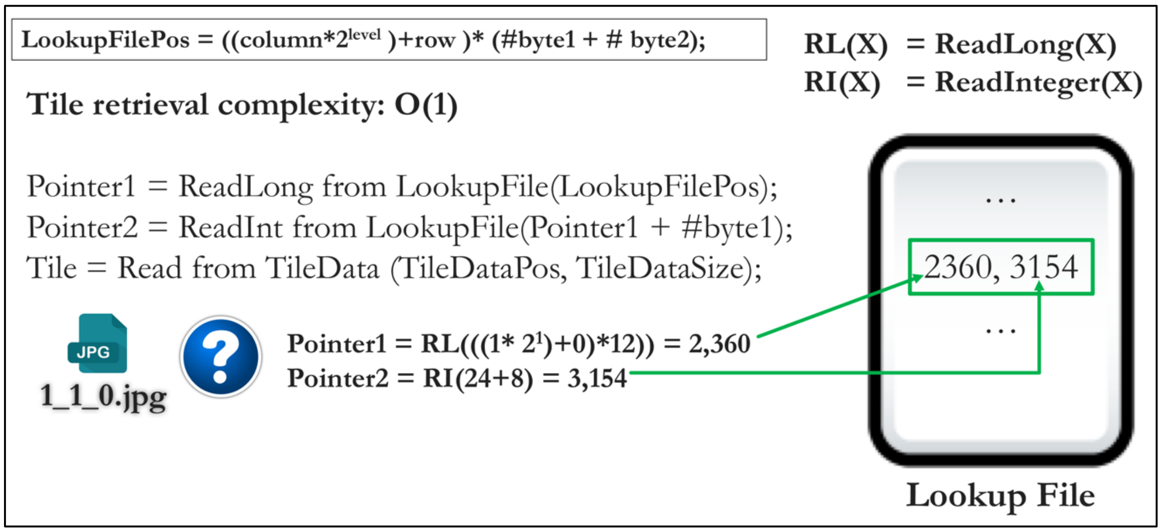

LookupFilePos = ((column*2^level)+row)* (#byte1 + # byte2);
RL(X) = ReadLong(X)
RI(X) = ReadInteger(X)
Tile retrieval complexity: O(1)
Pointer1 = ReadLong from LookupFile(LookupFilePos);
Pointer2 = ReadInt from LookupFile(Pointer1 + #byte1);
Tile = Read from TileData (TileDataPos, TileDataSize);
JPG
1_1_0.jpg
Pointer1 = RL(((1* 2^1)+0)*12)) = 2,360
Pointer2 = RI(24+8) = 3,154
...
2360, 3154
...
Lookup File

Figure 4.3 An example of computing the pointers within the Lookup File based on the equation (1) to retrieve the 1_1_0.jpg tile

Overall, the tile retrieval is done in real time; thus, the complexity of searching the tile is also done in real time. As showcased in Figure 4.3, since the pointers are stored in the Lookup file, by reading the tile starting point in the TileDataset file and the expected size of the tile, the tile can be retrieved without linearly examining other tiles. The Random-Access files allow such functionality since the information is stored in raw binary data. However, this functionality comes with two drawbacks:

1. The implementation of the LevelFilesSet is complicated and requires careful design patterns while developing. Similarly to all data structures, if the design patterns are not optimal, the results will follow suit.
2. The memory complexity increases since the Lookup file requires space for storing the pointers. The alternative proposed techniques do not require this additional space.

If implemented correctly, the LevelFilesSet will be optimal and, as presented in Chapter 4, optimal results are expected for its performance.

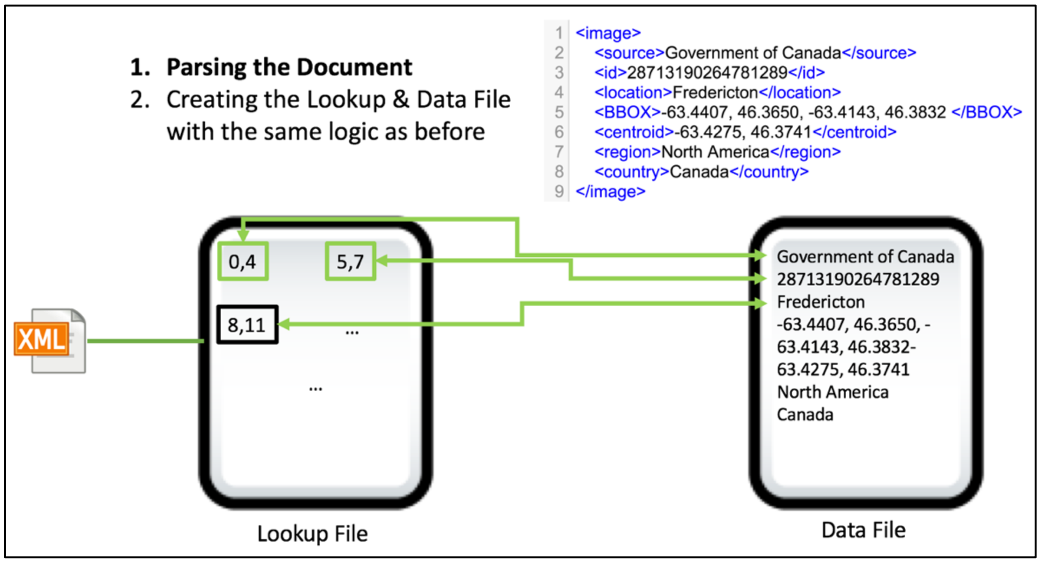
4.2. Metadata storage

The system is expected to support several other types of data storage, such as Metadata. Metadata are used to store information relevant to the dataset (Yee et al., 2003). For instance, if the system uses data from Open Government Canada Data, then the organization has to be credited accordingly by being stored in the Metadata record. The Metadata are usually stored in XML files and, in this example, a relevant tag could be the following: *<source> Open Government Canada Data</source>*. This raises the question of where the metadata have to be stored, and whether the LevelFilesBlock (see section 5.6) could be used for Metadata storage as well.

Multiple systems need to store information as Metadata. In the web tiled map management server, a tile can be georeferenced (Wieczorek et al., 2004) so that it can store further information relevant to its purpose.

4.2.1. LevelFiles storing Metadata

With minimal alteration, the implementation of the data structure, the LevelFilesSet can support Metadata manipulation. Supporting Metadata storage requires knowledge of the structure of the XML file beforehand since the tags have to be assigned specific bytes, as with the case of storing the tile images.



```
<image>
  <source>Government of Canada</source>
  <id>28713190264781289</id>
  <location>Fredericton</location>
  <BBOX>-63.4407, 46.3650, -63.4143, 46.3832 </BBOX>
  <centroid>-63.4275, 46.3741</centroid>
  <region>North America</region>
  <country>Canada</country>
</image>
```

Figure 4.4 Using LevelFilesSet to store Metadata

For instance, as showcased in Figure 4.4, each item of data information can be stored in the Lookup file and retrieved via the Data file, just as with the tile images. If the developer wants to access the “location” field from the Metadata, then the lookup file will be parsed to retrieve the pointers pointing at the Data File and then retrieve the data.

Since the LevelFilesSet is more efficient than any other data structure, Metadata storage is expected to be useful for tiled map management systems that use Metadata to store additional information for the tile dataset. Furthermore, this is an example of how the LevelFilesSet can be expanded to support any type of data, including Metadata.

4.3. The LevelFilesBlock data structure

The core ability of storing and retrieving tiles makes LevelFilesSet the most suitable and efficient solution. However, as previously stressed, when faced with large zoom levels an architecture must be designed to support the LevelFilesSet across multiple storage disks. The Table 4.1 and Figure 4.5 provide a conservative estimation, if the average tile size is 4KB, of the expected size for each zoom level.

Table 4.1 Expected size for each Zoom Level

Tile Zoom Level	Number of Tiles	Needed Storage Memory (TB)
14	4^{14}	1

15	4^{15}	4
16	4^{16}	16
17	4^{17}	64
18	4^{18}	264
19	4^{19}	1,024
20	4^{20}	4,096
21	4^{21}	16,384
22	4^{22}	65,536

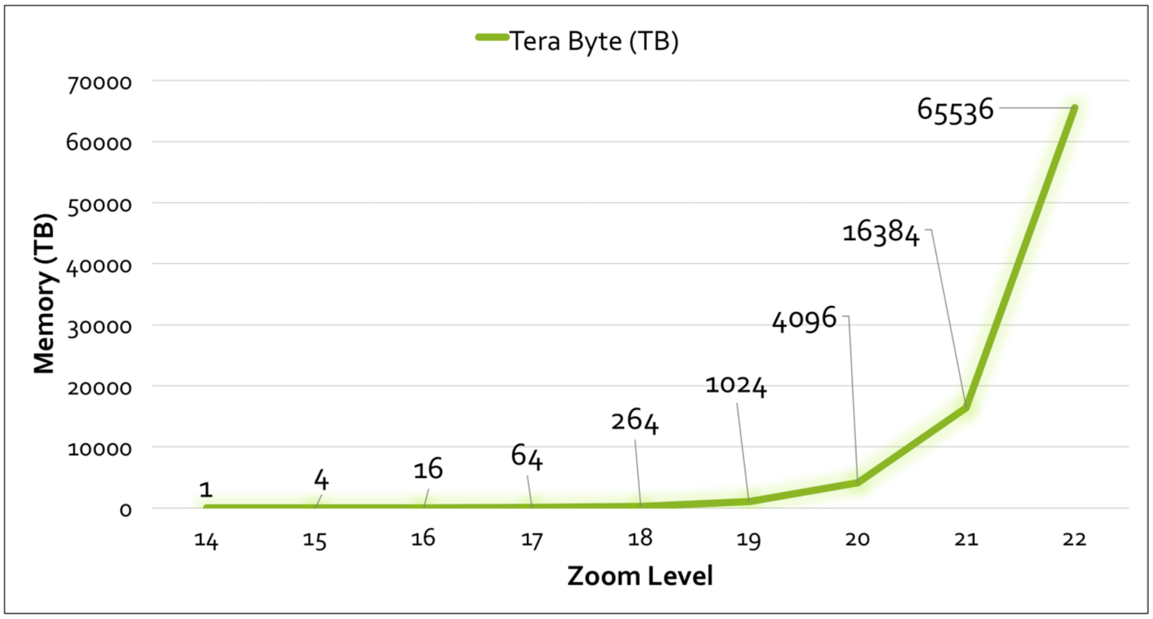


Figure 4.5 Total expected size of tiles for each zoom level

For instance, for the 22nd zoom level, if the average size of an image is 4KB, then the total size of the tiles is expected to be 65.6 PB; which means that the TileDataset file will have an equal size. Inarguably, storing this large number of information into one single file proves to be impossible and thus non-scalable. The need to distribute tiles across multiple storage systems becomes vital and the LevelFilesSet, in its simple form, does not fulfil that requirement. Instead, a hybrid approach, based on the benchmarking scenarios and scalability concerns, can be created, as illustrated in Figure 4.6.

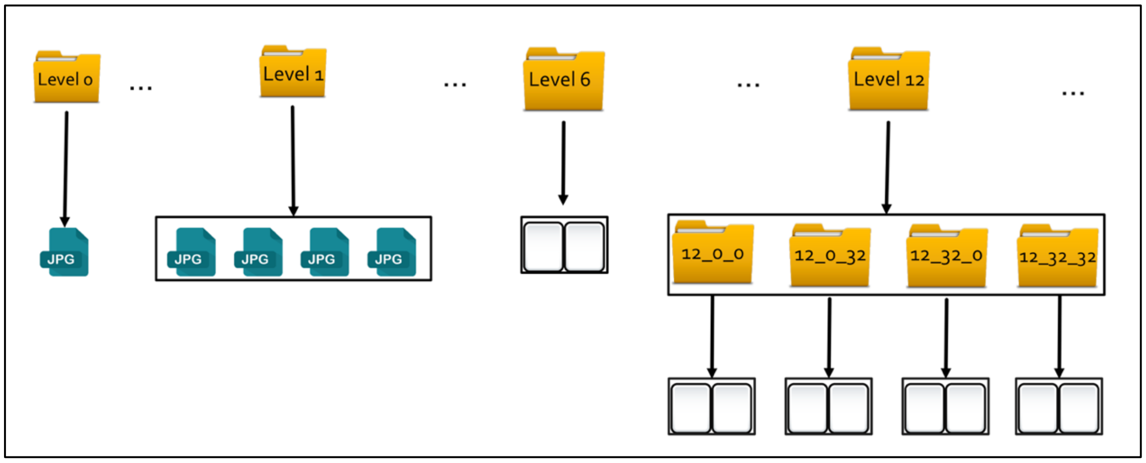


Figure 4.6 The LevelFilesBlock - A hybrid approach based on the LevelFilesSet and the ImageBlock

LevelFilesBlock combines LevelFilesSet with ImageBlock. It takes advantage of the speed superiority of LevelFilesSet and the elegance of ImageBlock, which together can distribute the tiles across different storage systems. For zoom levels 0 to 5, this structure follows SimpleFormat's (and ImageBlock's) format. For the zoom levels 6 to 11, the LevelFilesSet format is applied. For the upper zoom levels, ImageBlock's structure is applied with the major difference being that instead of tile images, LevelFilesSet is used per subdirectory. The logic behind this approach is to take advantage of LevelFilesSet performance superiority and ImageBlock's sophistication in regard to scalability. A major factor of this approach is limiting the average size of the image. The goal of each LevelFilesSet within each subdirectory is not to surpass 64GB.

This number (64GB) is empirically extracted for each application and is depended on the average system storage capacity. The goal of selecting a fair constant is to re-ensure that the system will be able to handle any insertion or deletion within a sub-directory. For instance, if the system has multiple storage systems of 1TB capacity, then the constant can be set on 64GB (16% of the system’s total size). On the other hand, increasing the sub-directories leads to more demands on the system (as derived from Chapter 5.5). Based on the latest concern, ImageBlock’s tile-limit number can be increased. For example, if the limit is set to 1 million, then, since the average size of each image is 4KB, the expected size of the total images will be approximately 1GB. The average image size may be larger than 4KB. That is, the image size depends on the category of the tiled map management system and its purpose (e.g., high-resolution tiled map management system, and so on). In the end, the developer must take inconsideration the following

a) the average tile size within each zoom level
b) the number of the tiles within each sub-directory
c) the maximum capacity of the storage system

By using LevelFilesBlock, the tiles can be distributed across different storage systems in an efficient and modifiable way and thus both performance and scalability concerns can be fulfilled. The developer is expected to adjust the maximum number of subdirectories based on the average size of the images.

5. LOCAL BENCHMARKS

In this section, several benchmarks that measure the performance of the proposed data structures under different system specifications are developed and analyzed. The specifications of the computer that ran the tests described in this research are shown in Table 5.1:

Table 5.1 Computer Configurations

System	*CPU*	*Memory*
Macintosh Sierra	Intel® i7, 2.2 GHz	Kingston SSD 256GB, 16GB RAM
Windows 10	Intel® Xeon® CPU E5-2620 V2 @ 2.10GHz	Samsung SSD 512GB, 8 GB RAM
Windows 10	Intel® Xeon® CPU E5-2620 V2 @ 2.10GHz	Western Digital 3TB HDD 8GB RAM
Database (Macintosh)	Postgres 9.3 PgAdmin®	Kingston SSD 256GB, 16GB RAM

5.1. Tile Generation Benchmark

Figure 5.1 and Figure 5.2 show how much time it takes to generate the dataset for the LevelFilesSet and the database. The database takes approximately 32 times longer than the LevelFilesSet to copy the dataset and store the tiles as BLOBs. This benchmark is an indicator of the performance problem and significant delays that arise by using the database as a storage option

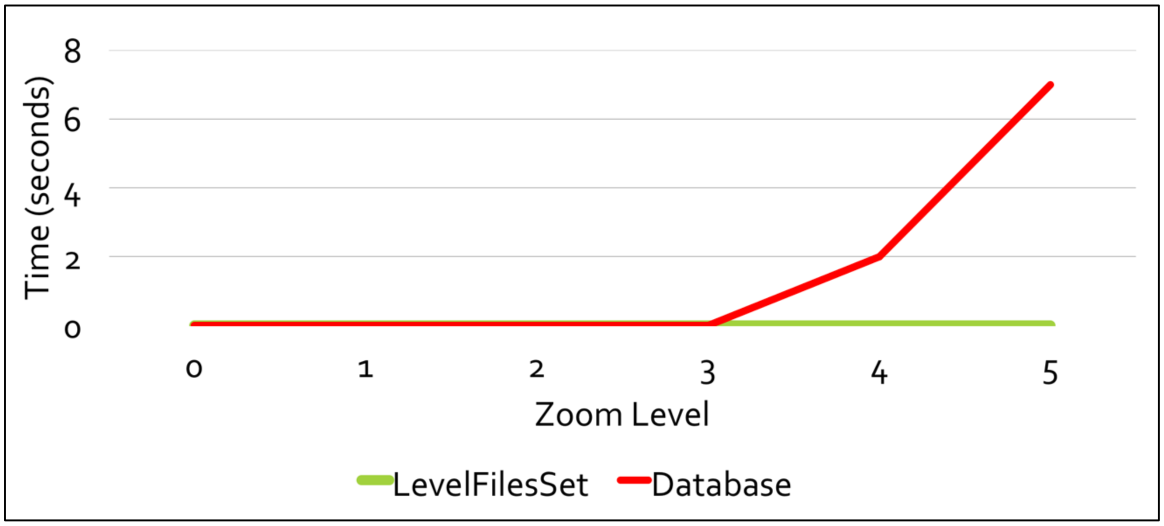


Figure 5.1 Tile generation performance for zoom levels 0 until 5 (Macintosh, Postgres 9.3)

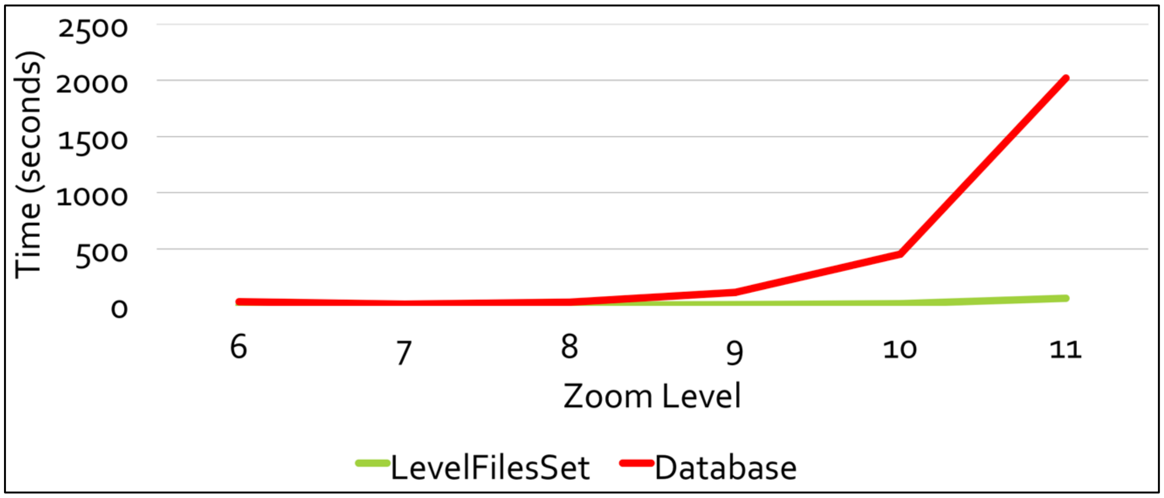


Figure 5.2 Tile generation performance for zoom levels 6 until 11 (Macintosh, Postgres 9.3)

In the next sections, several benchmarks are developed that address the performance constraints of each data structure under different system specifications when a tile is requested for retrieval. The benchmark phases were not predefined but, instead, they were developed based on the results of each phase. Each phase is based on the results of the previous phase and is developed to extract new goals and achieve new finding(s). The logic followed in the benchmarks as follows:

The first benchmark examines the average time of reading a tile image in each zoom level. The dataset in this scenario will have missing tiles. This benchmark works as a first prototype and indicates the performance of each data structure. The purpose of this benchmark is to extract initial results that will lead to further investigation of what is needed to be examined in the future.

To validate that the network performance is similar to the performance examined in the localhost environment from the first benchmark, the second benchmark examines the relationship between the localhost and network results. This benchmark ensures that the first results are applicable to a real-case scenario. As in the first benchmark, this benchmark is expected to lead into further results and indicate further steps to be taken.

The third benchmarking phase will address the issues discovered from the two previous phases. In this benchmark, the dataset will deal with the issue of the missing tile dataset, as described in Chapter 4, and I will propose a benchmarking algorithm that will allow to fairly measure the performance of each proposed data structure and, finally, will run it on different system environments to ensure the validity of the performance. The logic is described in Figure 5.3.

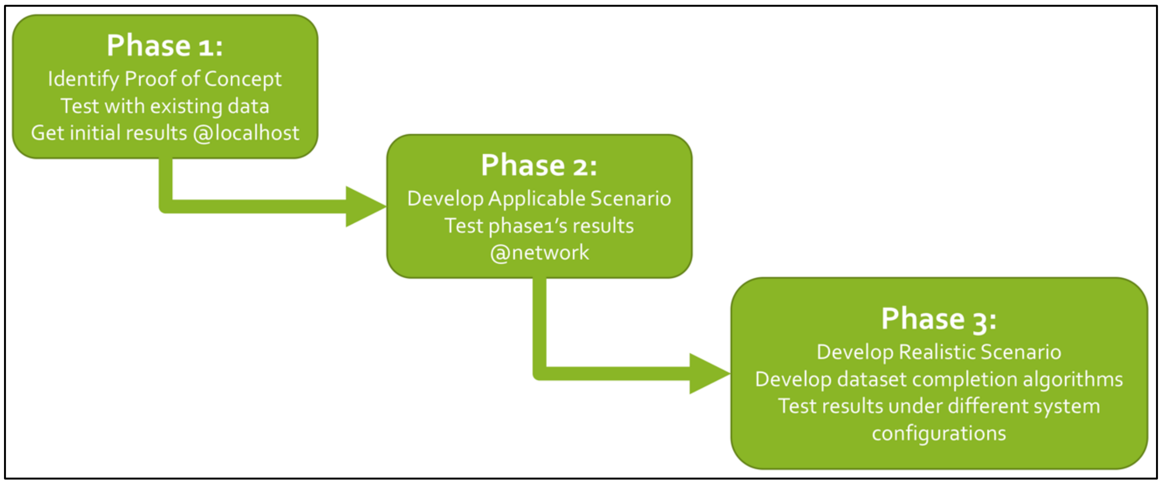

Phase 1:
Identify Proof of Concept
Test with existing data
Get initial results @localhost
Phase 2:
Develop Applicable Scenario
Test phase1's results
@network
Phase 3:
Develop Realistic Scenario
Develop dataset completion algorithms
Test results under different system configurations

Figure 5.3 The three benchmarking phases

5.2. The first benchmarking phase

There were three core phases during which the data structures were compared. In the first phase, a dataset with missing tiles was tested to present an initial idea on how the various solutions perform; a comparison between the File System (SimpleFormat and ImageBlock), Databases, and LevelFilesSet. The algorithm's pseudocode can be seen in Figure 5.4.

```
For Each Level:
Total Avg_Time_Duration = 0;
        For Each Tile:
                Start_Time = Get_Current_Time();
                Read Tile; // SimpleFormat, Database, LevelFilesSet
                End_Time += Get_Current_Time() – Start_Time;
        End For Each;
Avg_Time_Duration = End_Time / #tiles;
End For Each;
```

Figure 5.4 Pseudo-Code for the retrieval algorithm

This algorithm measures the average time of tile retrieval for each zoom level and for each solution (SimpleFormat, ImageBlock and LevelFilesSet). It reads through every tile image within each zoom level. Then, the needed time for reading the tile is added to the variable which holds the total time duration. In the end, the computed time is divided by the total number of the tile images in the zoom level. The output number will indicate how much time, on average, is needed to read a tile image from each zoom level. The resulting number is useful will showcase how the zoom levels (with different tile numbers stored) perform and how this number effects the performance of the system.

The first phase's benchmark runs under the Macintosh OS with the Database used being Postgres®. The results and can be seen in Table 5.2 (Figure 5.5).

Table 5.2 First benchmarking results (Macintosh, ExFAT, SSD, Postgres®)

Tile Zoom Level	Number of Tiles	Database (ms)	SimpleFormat (ms)	LevelFilesSet (ms)
0	1	0.05	0.01	0.001
1	4	0.1	0.16	0.003
2	16	0.19	0.29	0.024
3	64	0.41	0.3	0.03
4	256	0.48	0.36	0.27
5	1024	1.54	0.38	0.3
6	4096	2.28	0.46	0.32
7	892	3.24	0.59	0.46
8	3565	3.49	0.61	0.47
9	14257	3.51	0.62	0.5
10	57025	3.69	0.67	0.51
11	228097	3.73	0.69	0.52

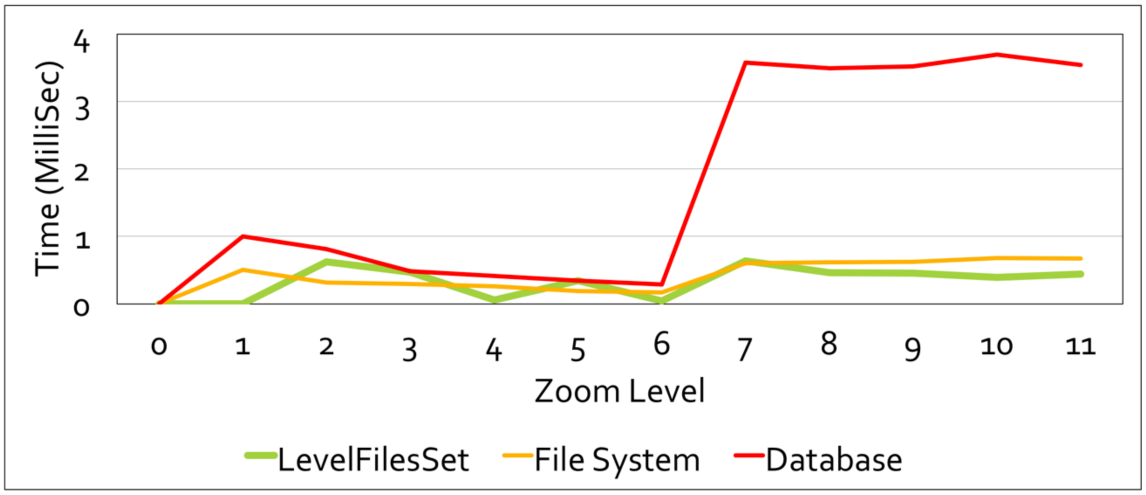


Figure 5.5 Graphical Representation of the first benchmark results

Figure 5.6 illustrates the total time that is required for all the tiles to be retrieved.

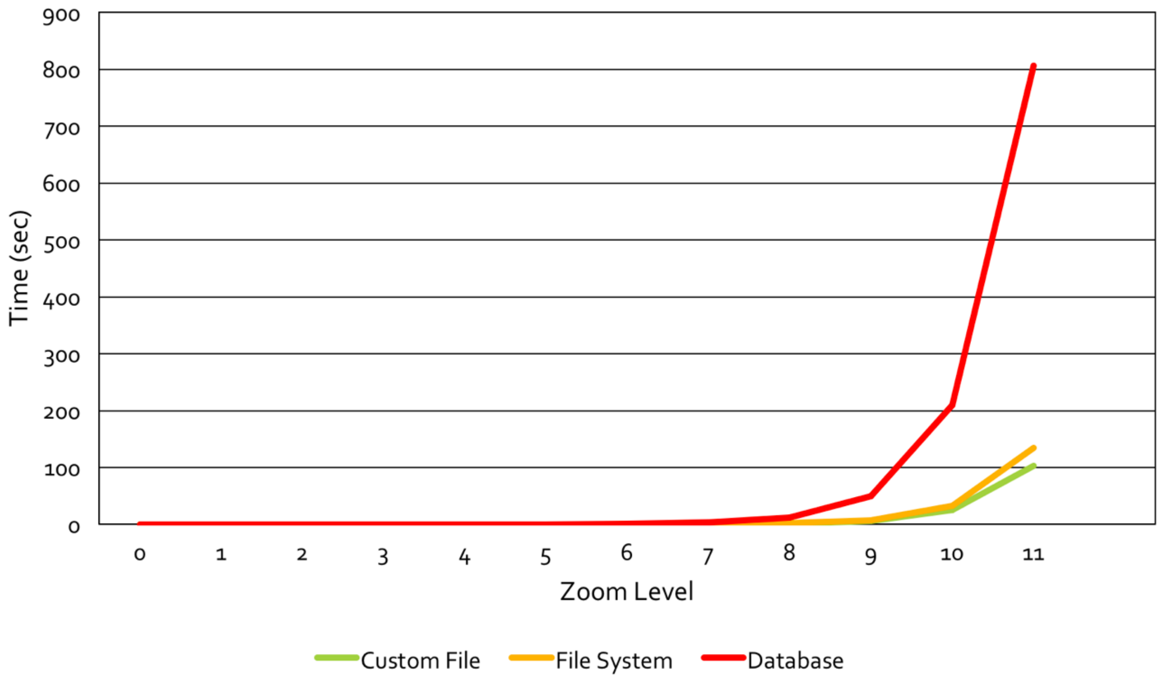


Figure 5.6 Total needed time for each tile to be loaded

The Database, in the 11th zoom level, is 6.2 times slower than the LevelFilesSet, rendering it insufficient to store large volumes of tile datasets. Furthermore, the

LevelFilesSet is approximately 33% faster than the SimpleFormat. While this benchmark provides an initial proof of concept regarding each solution's performance, it does not take in consideration the fact that developers will request certain parts of the web map each time and that there will be multiple HTTP requests made on the server. The second benchmark analyzes the network performance and monitors certain latencies that would not be added while testing locally on the system.

5.3. The second benchmarking phase based on the Network Performance

To make the benchmark scenarios more applicable to real cases, this benchmark examines whether the HTTP request for a tile will have the same average performance as the first benchmark that ran locally. For this benchmark, the Apache JMeter networking tool was used. The JMeter is a Java open source software, designed to examine functional behavior and measure the performance of the application (Halili, 2008). It is used to test both static and dynamic resources of the application, regardless its backend technologies and languages. It can be used to simulate a heavy load on a server, group of servers, network, or object to test its strength or to analyze overall performance under different load types. JMeter provides a user-friendly interface. This benchmark simulates 600 requests per second that run repeatedly 50 times. After that, an HTTP GET request is made to all the tiles on the server. For that purpose, a CSV Data Set Config Element is created to load the CSV file containing all the tile names of the system. The logic of this algorithm is the same as the one in the algorithm used for the first benchmark (Figure 5.5); the difference is that in this benchmark the network performance is analyzed to test whether it is identical to the performance derived from the first benchmark. The average results can be seen in Table 5.3.

Table 5.3 JMeter Benchmark Results, where

Data structure: the tested data structure

No. of samples: the number of successful responds of the server

Average time: the average time of response for each request

Data Structure	No. of samples	Average time (ms)
LevelFilesSet	21480	29
SimpleFormat	13060	41
Database	1567	233

The samples are different by approximately 65%, a result almost identical with the performance of the first benchmark that ran locally. This outcome proves that the behavior between a locally-run server and an online web server is very similar. On the other hand, the average time of a response has a difference of approximately 41%. This indicates that the response of the server is lower than it is on the localhost, and that is because of the HTTP overhead (Heidemann et al., 1997). For each HTTP request, an additional overhead is added for establishing the connection between the client and the web server, resulting in an additional latency of 24%. A way to reduce this overhead is to request more than one tile at a time. For instance, if 10 tiles are requested, then the total expected latency added by the overhead will be lower than the initial one. However, since the server will be responding back with 10 tiles, the required bandwidth of the network will be much higher; thus, there is a tradeoff between the server's performance and the available network's bandwidth.

In the third benchmark phase, an algorithm which takes all these aspects into consideration is developed and run to test the performance of the proposed data structures.

5.4. The third benchmarking phase

In the third phase, there are 3 individual benchmarks under which the data structures are compared. It is important to mention that the purpose of these benchmarks is to test with all the expected tiles on the system (Chapter of Data Structures). The pseudo-code for the algorithm used for the benchmark tests can be seen in Figure 5.7.

```
Randomly Select Area (10 tiles);
For Each Level:
    For (threshold 1: 10): //threshold = requesting tiles number
        Compute LevelFilesSet Performance;
        Compute ImageBlock Performance;
        Compute SimpleFormat Performance;
    End For;
    Compute Average Time();
End For Each;
```

Figure 5.7 Pseudocode for the benchmarking scenarios

The above algorithm selects randomly an area of 10 tiles for each zoom level. Then, an average time variable is used to estimate the required time that every data structure. The number of requested tiles increases sequentially each time (e.g., 1, 2, 3 and so on). In the end, the average time is computed. To simulate a realistic benchmarking scenario, 1 to 10 tiles are requested for each zoom level. The threshold indicates the number of the tiles that are requested each time. Finally, the average time is computed and reported. It is significant that, in the testing dataset, a tile has an approximate size of 4 Kilobytes (KB). Depending on the network traffic, the developer might want to retrieve more than one tile per request. This algorithm takes that feature into account and represents a fair comparison with the number of the requested tiles varying from 1 to 10. The first test ran on Macintosh under the ExFAT file system and provided the following results (Table 5.4 and Figure 5.8):

Table 5.4 Benchmarking results (Macintosh, SSD)

Tile Zoom Level	Number of Tiles	SimpleFormat (ms)	ImageBlock (ms)	LevelFilesSet (ms)
0	1	1.3	1.2	1.1
1	4	1.3	1.2	1.1
2	16	1.4	1.3	1.2
3	64	1.4	1.3	1.2
4	256	2.5	2.3	2.1
5	1,024	4.2	4.7	2.7
6	4,096	8.3	9.3	6.3
7	16,384	11.4	12.2	9.1
8	65,536	13.1	12.9	9.4
9	262,144	16.4	16.1	10.8
10	1,048,576	19.8	19.5	11.9

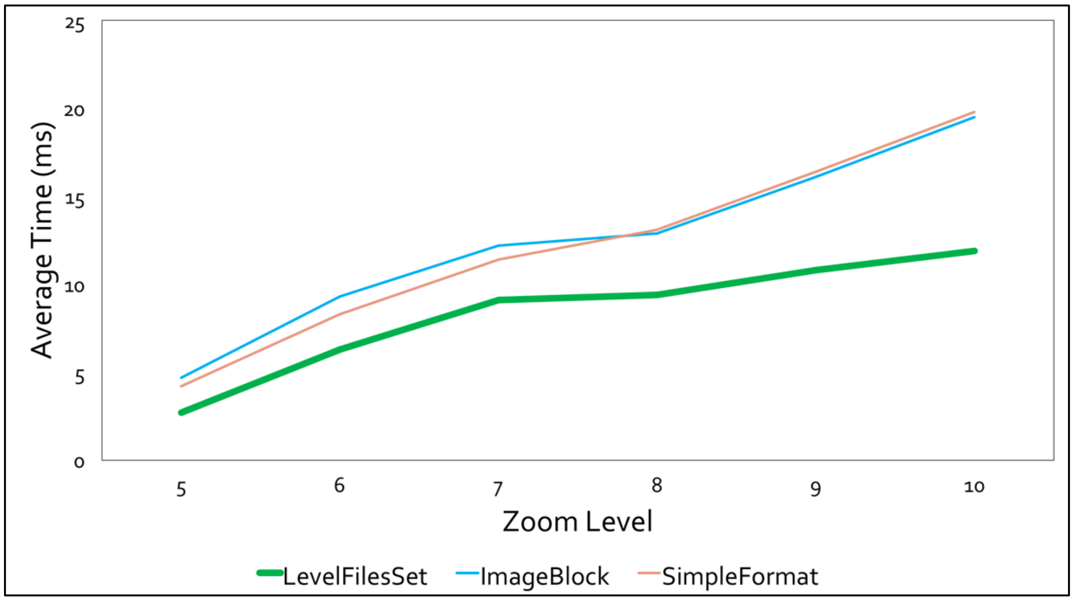


Figure 5.8 Graphical Representation of the first benchmark results for the zoom levels 5 until 10 (Macintosh, ExFat and SSD)

As observed, for zoom level 10, it takes an average of 11.9 milliseconds (ms) to retrieve the tiles by using LevelFilesSet, 19.5 ms for ImageBlock and 19.8 ms for SimpleFormat. The performance improvement of LevelFilesSet over SimpleFormat and ImageBlock, in zoom level 10, is 66%. SimpleFormat and the ImageBlock have almost identical results.

The second benchmark ran on Windows (NTFS) using a Solid-State Drive (SSD). The Windows SSD memory capacity allowed the test to run until zoom level 11, rather than the 10 levels in the case of the first benchmark that ran on Macintosh. The results are reported in the Table 5.5 and Figure 5.9.

Table 5.5 Benchmarking results on Windows (SSD)

Tile Zoom	Number of	SimpleFormat	ImageBlock	LevelFilesSet

Level	Tiles	(ms)	(ms)	(ms)
0	1	5.4	5.4	7.4
1	4	5.5	5.5	7.6
2	16	5.5	5.6	7.8
3	64	5.5	5.6	7.9
4	256	10.8	10.7	7.9
5	1024	21.7	22.3	9.9
6	4096	44.3	41.2	13.8
7	16384	54.9	52.0	16.6
8	65536	67.3	64.3	18.4
9	262,144	68.7	65.8	18.6
10	1,048,576	69.8	72.1	19.9
11	4,194,304	75.1	89.7l	21.8

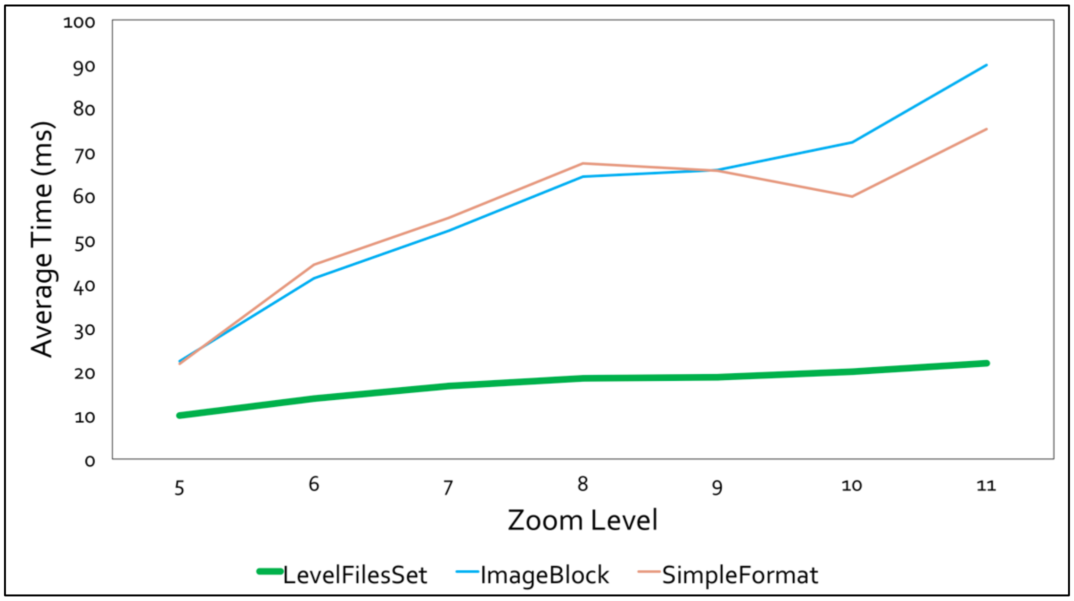


Figure 5.9 Graphical Representation of the first benchmark results for the zoom levels 5 until 11 (Windows, NTFS and SSD)

The performance improvement of LevelFilesSet over the SimpleFormat and ImageBlock, in the 11th zoom level, is approximately 323%.

For the third benchmark test, the same Windows machine is used, with the difference being that the files are stored in the Hard Disk Drive (HDD) instead of the SSD. The HDD capacity allowed the test to run until the 13th zoom level. The results are as follows (Table 5.6 and Figure 5.10):

Table 5.6 Benchmarking results on Windows (HDD)

Tile Zoom Level	Number of Tiles	SimpleFormat (ms)	ImageBlock (ms)	LevelFilesSet (ms)
0	1	10.3	10.3	10.7
1	4	10.4	10.4	10.8

2	16	10.4	10.4	10.8
3	64	18.9	18.8	11.1
4	256	35.5	35.4	15.6
5	1,024	58.2	58.2	23.7
6	4,096	92.9	99.9	41.6
7	16,384	226.9	260.5	90.8
8	65,536	352.7	393.8	127.8
9	262,144	444.7	573.9	171.5
10	1,048,576	567.3	881.7	263.1
11	4,194,304	845	1,110.1	314.1
12	16,777,216	913.4	1,282,9	367.8
13	67,108,864	941.9	1,585.7	371.1

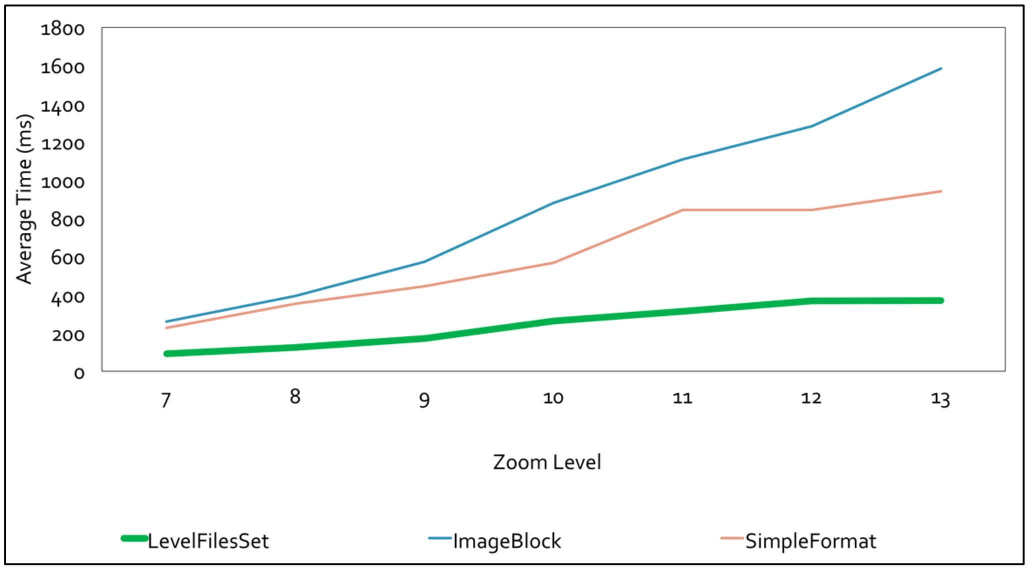


Figure 5.10 Graphical Representation Third benchmark for the zoom levels 7 until 13 (Windows, NTFS and SSD)

The performance improvement of LevelFilesSet over the SimpleFormat and ImageBlock, in the 13th zoom level, is approximately 327%.

5.5. Outcome

The LevelFilesSet's retrieval time is faster than that of SimpleFormat and ImageBlock in all the benchmarking scenarios. It is important to highlight the idea behind the LevelFilesSet methodology. Not only does LevelFilesSet provide better results than the other structures, but it also performs optimally under any operating and file system. However, one significant issue arises as the zoom level increases: the TileData file gets extremely large, rendering it impossible to store tiles using only one disk. For instance, for the 15th zoom level, it is expected to store 1,073,741,824 tiles, and since the average size of a tile is 4KB, the expected size of the entire zoom level is approximately 4 Terabyte (TB). As the file size increases, LevelFilesSet develops problems. A future update of LevelFilesSet, described in the chapter 4.3, is expected to provide the necessary guidelines that will show the way the tiles should be divided across different storage disks in an efficient and scalable manner. The LevelFilesSet's code can be found in the Appendix I.

5.6. Google Cloud Benchmark

With the recent rise of the cloud-based technologies, more and more applications are based entirely on the cloud and thus, it is worth examining the performance of SimpleFormat, ImageBlock, and LevelFilesSet on such platforms. Google Cloud offers a wide variety of APIs which create a suitable solution for tile storage and retrieval. Furthermore, Google Cloud offers a set of easy-to-use tools and documentation that make it easy for developers to implement tile systems. By choosing a cloud-based approach to tile storage and retrieval, developers can simply upload the dataset and retrieve it. Cloud-based approaches free the user from having to develop and implement from scratch a local data center. Consequently, the company can avoid hiring specialized personnel as well as saving space and funds that would be dedicated to a physical local data center. Lastly, choosing a cloud-based solution eliminates the need for backups, security, maintenance, hardware upgrade, and peripheral running costs. Importantly, however, there are a number of trade-offs when choosing cloud hosting. Namely, the company relies on a single vendor that can control the availability and price of the offered services. Further, this solution is mostly applicable for small to medium sized companies, as scalability will rapidly increase both the price and the reliance on a single external vendor. Beyond expanding on the intricacies of these benefits and drawbacks of using a cloud based solution, the next Chapter deals with serving tiles by using the Google Cloud.

6. STORING AND RETRIEVING TILES ON GOOGLE CLOUD

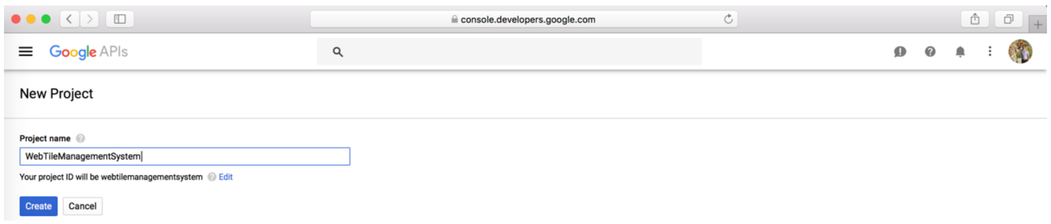
Vendors such as Google, Amazon, and ESRI provide rich collections of online cloud based API solutions. The cloud is a network of servers, each of which are developed to address different functionalities (Argyraki et al., 2009). Cloud computing provides storage and retrieval of data over the Internet instead of the computer’s hard drive. One of the main benefits of using Google Cloud is the dynamic availability of the server (including upgradability, automated backups, automated system updates, and easy integration). Multiple companies and developers are using Google Cloud as their primary solution for information storage and retrieval and thus, it is important to examine how a WTMMS could be built based on the Google Cloud Platform.

6.1. Interaction and tile storage with Google Cloud

Google provides graphical and command-line solutions for deploying information on the cloud. To upload the dataset on the server, the following steps are performed:

1. Log in the Google Cloud account by visiting the URL: https://console.developers.google.com.

2. Create a new Google Cloud project (Figure 6.1).

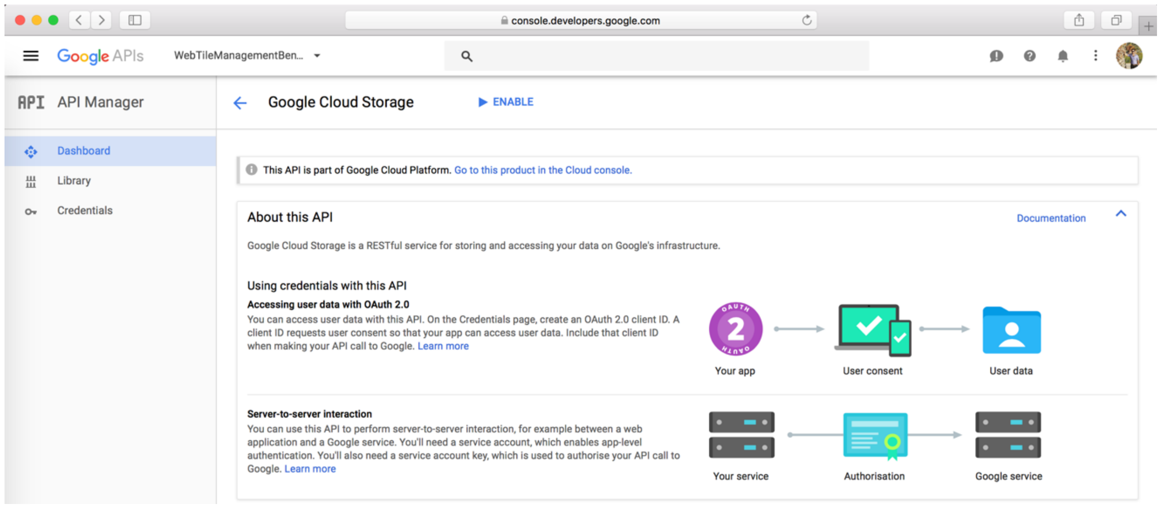


Project name: WebTileManagementSystem

Your project ID will be webtilemanagementsystem Edit

Figure 6.1 Creating new Google Cloud project

3. Enable the Google Cloud Pricing by assigning credit card credentials and other personal information.
4. Enable Google Storage API (Figure 6.2).



This API is part of Google Cloud Platform. Go to this product in the Cloud console.

About this API

Google Cloud Storage is a RESTful service for storing and accessing your data on Google's infrastructure.

Using credentials with this API

Accessing user data with OAuth 2.0

You can access user data with this API. On the Credentials page, create an OAuth 2.0 client ID. A client ID requests user consent so that your app can access user data. Include that client ID when making your API call to Google. Learn more

Server-to-server interaction

You can use this API to perform server-to-server interaction, for example between a web application and a Google service. You'll need a service account, which enables app-level authentication. You'll also need a service account key, which is used to authorise your API call to Google. Learn more

Figure 6.2 Enabling Google Cloud Storage API

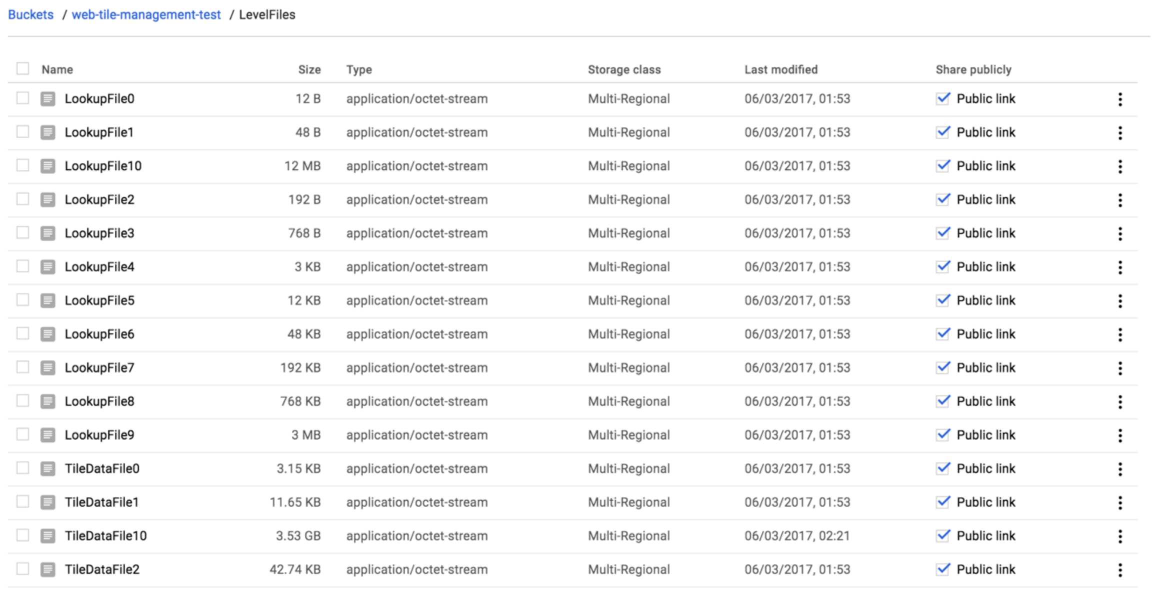
5. Download Google Cloud SDK in the local machine (URL: https://cloud.google.com/sdk
6. Install SDK and run the command from the command prompt (or terminal):

```
$ gcloud auth login --WebTileManagementSystem
```

7. Read and accept the Google Account permission statement.

The next details showcase how the tile dataset can be uploaded on the Google Cloud. Developers can either use command-prompt (with the *gsutil* command) or web user interface (UI). In the following examples, the steps are showcased with the UI approach:

1. In Google Cloud Storage, create a Google Bucket.
2. Select the tile dataset to upload in the different folders. For instance, the stored LevelFilesSet can be seen in Figure 6.3.



Buckets / web-tile-management-test / LevelFiles

Name	Size	Type	Storage class	Last modified	Share publicly
LookupFile0	12 B	application/octet-stream	Multi-Regional	06/03/2017, 01:53	✓ Public link
LookupFile1	48 B	application/octet-stream	Multi-Regional	06/03/2017, 01:53	✓ Public link
LookupFile10	12 MB	application/octet-stream	Multi-Regional	06/03/2017, 01:53	✓ Public link
LookupFile2	192 B	application/octet-stream	Multi-Regional	06/03/2017, 01:53	✓ Public link
LookupFile3	768 B	application/octet-stream	Multi-Regional	06/03/2017, 01:53	✓ Public link
LookupFile4	3 KB	application/octet-stream	Multi-Regional	06/03/2017, 01:53	✓ Public link
LookupFile5	12 KB	application/octet-stream	Multi-Regional	06/03/2017, 01:53	✓ Public link
LookupFile6	48 KB	application/octet-stream	Multi-Regional	06/03/2017, 01:53	✓ Public link
LookupFile7	192 KB	application/octet-stream	Multi-Regional	06/03/2017, 01:53	✓ Public link
LookupFile8	768 KB	application/octet-stream	Multi-Regional	06/03/2017, 01:53	✓ Public link
LookupFile9	3 MB	application/octet-stream	Multi-Regional	06/03/2017, 01:53	✓ Public link
TileDataFile0	3.15 KB	application/octet-stream	Multi-Regional	06/03/2017, 01:53	✓ Public link
TileDataFile1	11.65 KB	application/octet-stream	Multi-Regional	06/03/2017, 01:53	✓ Public link
TileDataFile10	3.53 GB	application/octet-stream	Multi-Regional	06/03/2017, 02:21	✓ Public link
TileDataFile2	42.74 KB	application/octet-stream	Multi-Regional	06/03/2017, 01:53	✓ Public link

Figure 6.3 LevelFilesSet storage in the Google Bucket

Each uploaded file has a public link that allows developers to access it through an HTTP request. There are several tools that can be used to process HTTP requests on the Google Cloud.

6.2. Java Servlets

A Java Servlet can be integrated with Google Cloud by following Google's specifications, as shown below. A Java servlet is a Java program that extends the capabilities of a server. Servlets have the purpose of building Web-based applications. They have access to the entire family of the Java APIs, including the JDBC API to access enterprise. The servlets architecture can be seen in Figure 6.4:

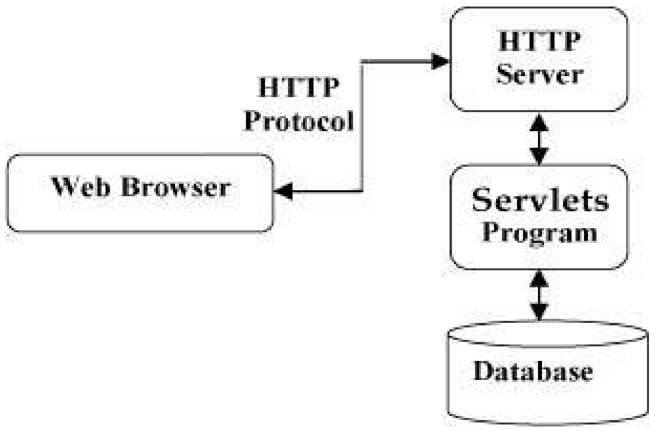


Figure 6.4 Java Servlet process (from Java EE, Oracle, 2000)

Java servlets perform the following tasks:

1. Read the data sent by the clients (browsers).
2. Read the implicit HTTP request sent by the clients (browsers).
3. Process the data and generate the results.
4. Send the explicit data back to the clients (browsers).
5. Send the implicit HTTP response to the clients (browsers).
6. The following figure depicts a typical servlet life-cycle scenario:

The lifecycle of a Java servlet can be described as follows:

- First, the HTTP requests coming to the server are delegated to the server container.
- The servlet container loads the servlet before invoking the *service* method.
- Then the servlet container handles multiple requests by spawning multiple threads, each of which executes the *service* method of a single instance of the servlet.

It is worth mentioning that in each request, the users access the same instance of the Java servlet; thus, the developer should be cautious about concurrency cases. For instance, when accessing files which change dynamically, if a file accessed by the end user changes, then the change will affect the end user who is accessing the file. For web tiled map management systems, there are no concurrency problems since the tiles are not updated dynamically. However, if the web tiled map management server would support dynamic tiles that change frequently, then concurrency should be resolved through careful design of the tile-loading algorithm.

Figure 6.5 describes the architecture of the deployed application.

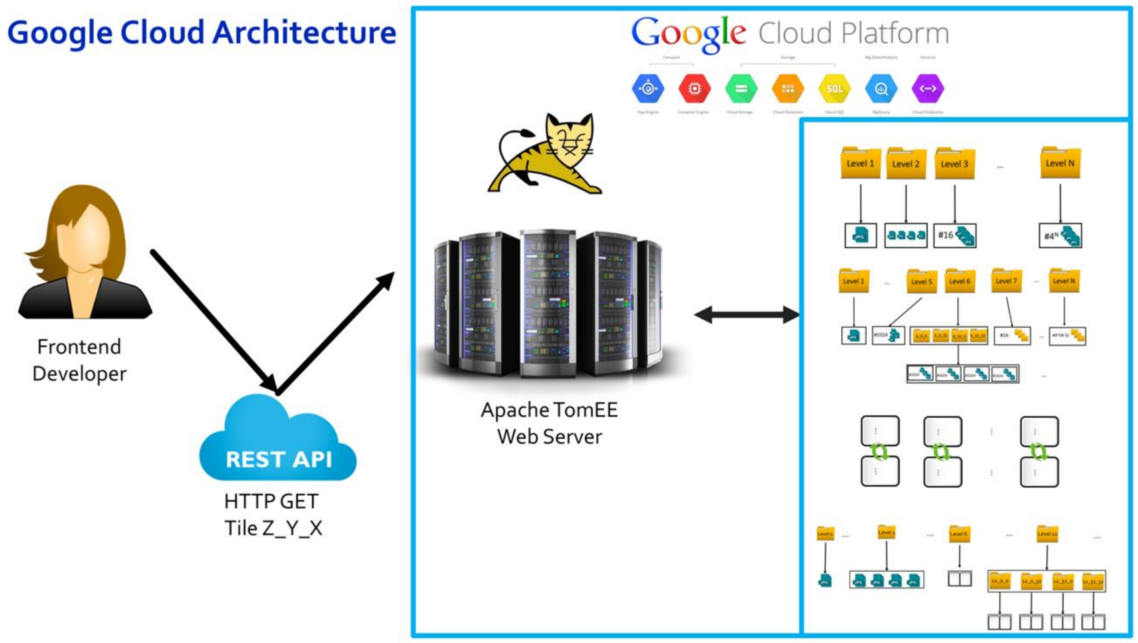
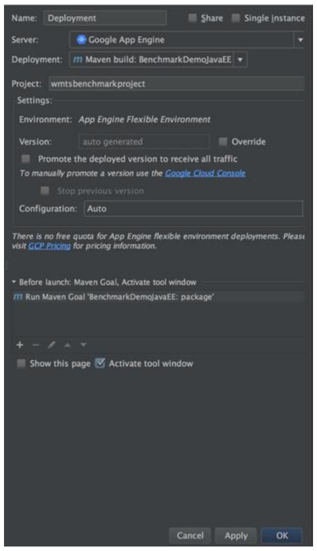


Figure 6.5 Architecture of Tile Serving in Google Cloud

IntelliJ offers a built-in plugin for easy-to-use Google Cloud Integration. The tool, as illustrated in Figure 6.6, provides a straightforward UI where the developer adjusts the configurations of the deployment.



Name: Deployment ☐ Share ☐ Single instance
Server: Google App Engine
Deployment: Maven build: BenchmarkDemoJavaEE
Project: wmtsbenchmarkproject
Settings:
Environment: *App Engine Flexible Environment*
Version: auto generated ☐ Override
☐ Promote the deployed version to receive all traffic
To manually promote a version use the Google Cloud Console
☐ Stop previous version
Configuration: Auto

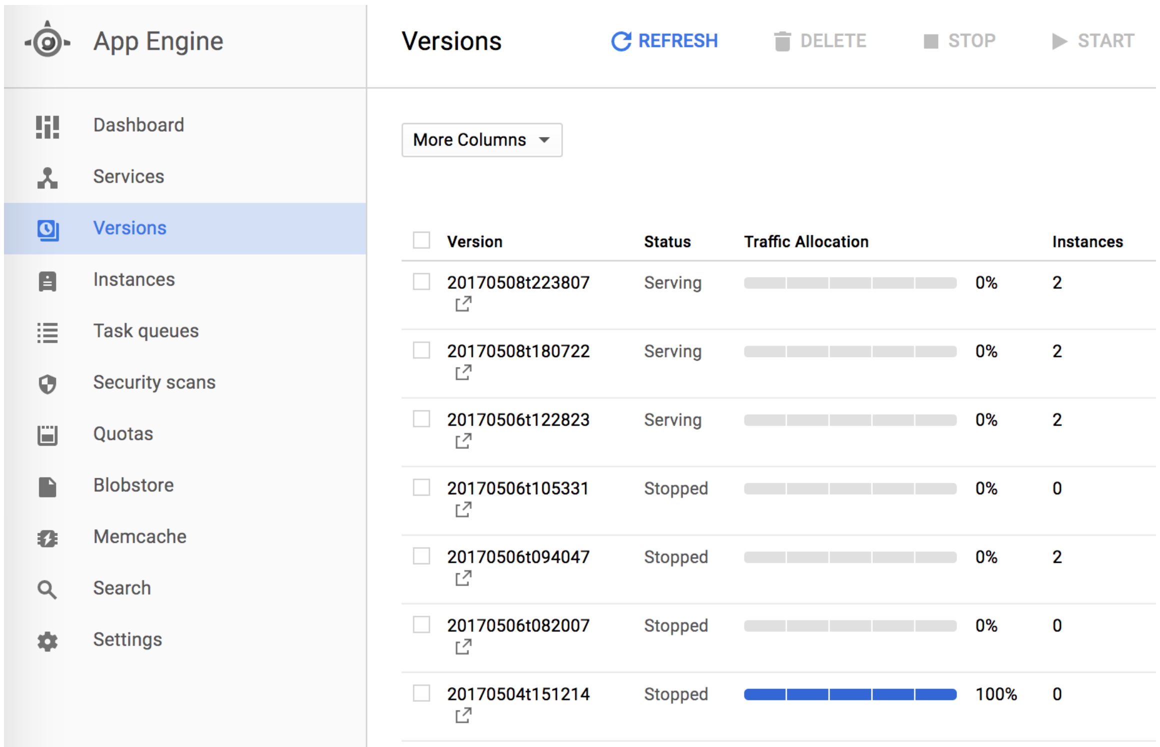
There is no free quota for App Engine flexible environment deployments. Please visit GCP Pricing for pricing information.

Before launch: Maven Goal, Activate tool window
Run Maven Goal 'BenchmarkDemoJavaEE: package'

☐ Show this page ✓ Activate tool window

Figure 6.6 IntelliJ and Google Cloud Integration

The project is configured with Apache Maven, which is a software project management and comprehension tool (Dixon et al., 2006). Maven can automate and manage the project’s build, reporting, and documentation from a central place of information without requiring further configuration from the developer. Maven offers great integration with JavaEE applications and it is used to build the project. From its Console, Google Cloud provides multiple versioning of the project. This means that whenever an update occurs, the developer has an option to update the current version of the project without altering the application which is already running on the cloud. This is rather useful for alpha and beta testing, as discussed by Craig et al. (2002; Figure 6.7).



Version	Status	Traffic Allocation	Instances
20170508t223807	Serving	0%	2
20170508t180722	Serving	0%	2
20170506t122823	Serving	0%	2
20170506t105331	Stopped	0%	0
20170506t094047	Stopped	0%	2
20170506t082007	Stopped	0%	0
20170504t151214	Stopped	100%	0

Figure 6.7 Google Cloud Application Versioning

6.3. Google Cloud Benchmarks

The same algorithm used for the local benchmarks presented in Chapter 5.4 will be used for this benchmark. However, since the Google environment hosts multiple applications and the traffic is different depending on temporal load during the test, the benchmark will run twice per day for three consecutive days, summing up to 6 different benchmark results, all of which will be performed at consistent times, separated by 12h: 10AM and 10PM (time zone UTC-3h). The times were chosen paradigmatically since high traffic is expected to occur in the morning and low traffic is expected to occur at night. However, since Google hosts multiple applications across different regions within the cloud, high traffic could occur at any time, and is not open to estimation (Savage et al., 2009). This benchmark controls for the differences in OS and hardware that are inescapable in local drive benchmarks. The tables with the analytical data which describe the benchmarks can be found in Appendix V.

This benchmark ran on May 10th 2017, at 10 AM. The results can be seen in Figures 6.8 and 6.9.

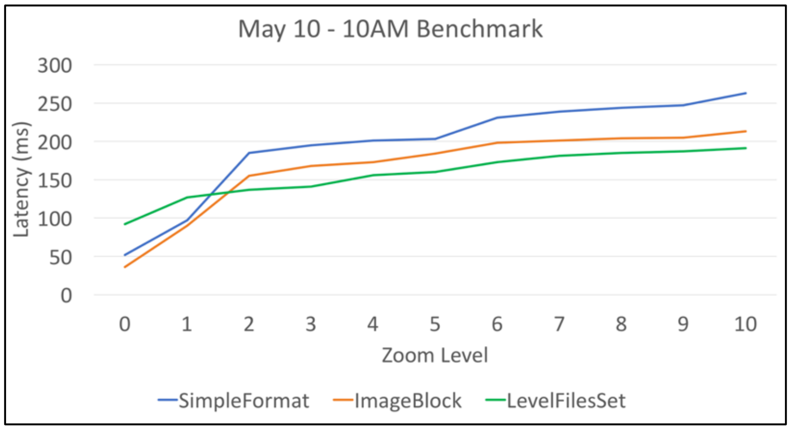


Figure 6.8 Benchmark ran on May 10th at 10AM

For the 10th zoom level, LevelFilesSet performed approximately 38% faster than SimpleFormat and 11.5% faster than ImageBlock.

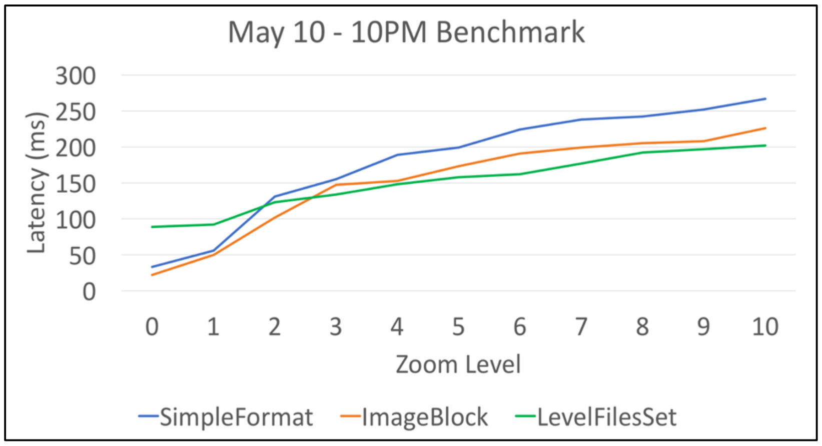


Figure 6.9 Benchmark ran on 10 May at 10PM

For the 10th zoom level, LevelFilesSet performed approximately 32% faster than SimpleFormat and 11.8% faster than ImageBlock.

The second day's benchmark ran on May 11th 2017, at 10 AM. The results are shown in Figures 6.10 and 6.11.

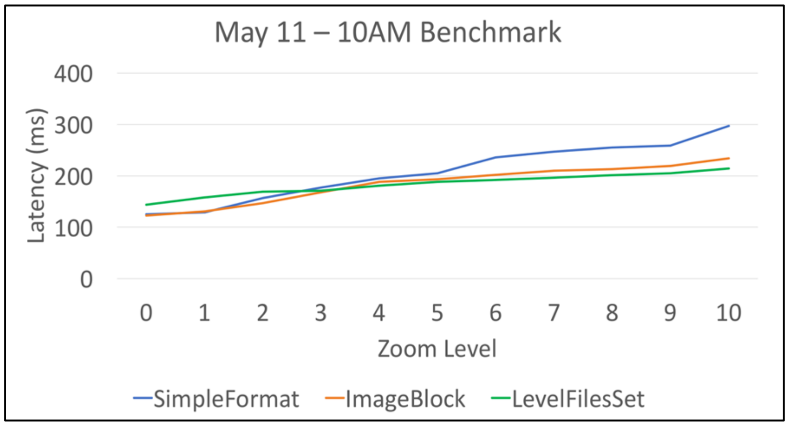


Figure 6.10 Benchmark ran on 11 May at 10AM

For the 10th zoom level, LevelFilesSet performed approximately 37.8% faster than SimpleFormat and 11.2% faster than ImageBlock.

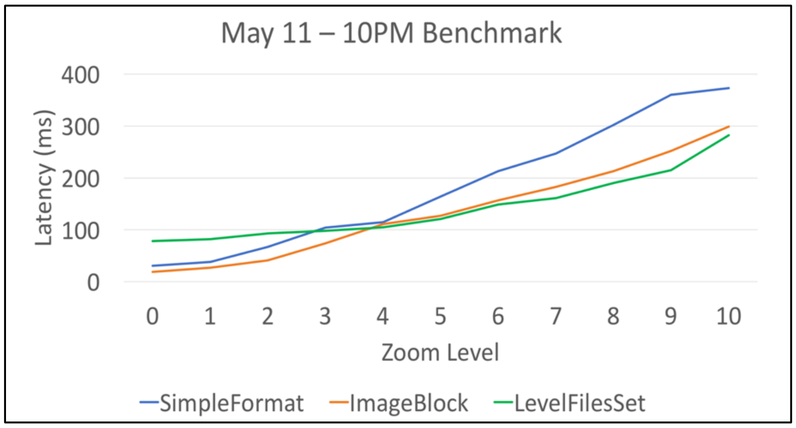


Figure 6.11 Benchmark ran on 11 May 2017 at 10AM

For the 10th zoom level, LevelFilesSet performed approximately 32.2% faster than SimpleFormat and 6% faster than ImageBlock.

The third benchmark results are illustrated in Figures 6.12 and 6.13.

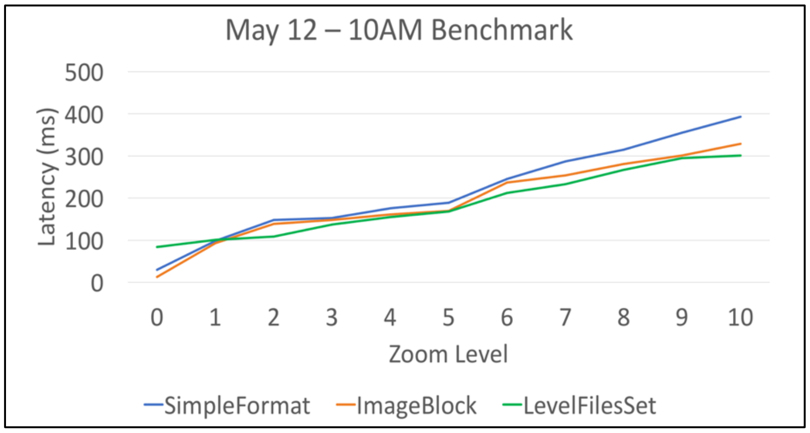


Figure 6.12 Benchmark ran on 11 May 2017 at 10PM

For the 10th zoom level, LevelFilesSet performed approximately 38% faster than SimpleFormat and 11.5% faster than ImageBlock.

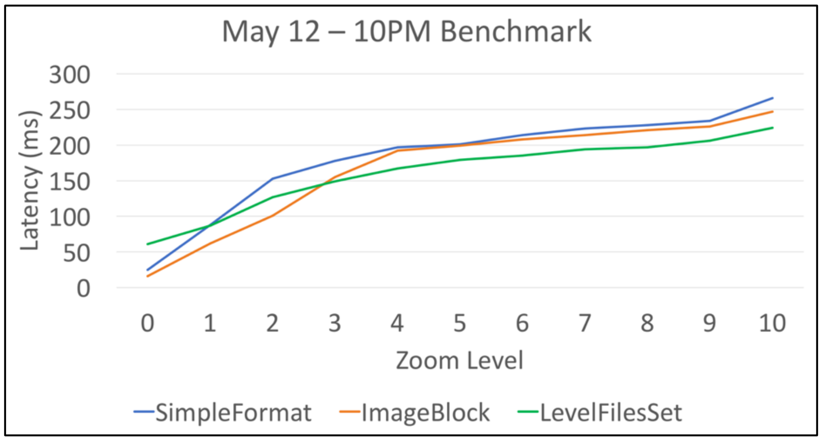


Figure 6.13 Benchmark ran on 12 May 2017 at 10PM

For the 10th zoom level, LevelFilesSet performed approximately 32% faster than SimpleFormat and 11.8% faster than ImageBlock.

6.6. Outcome

Within the span of three days, LevelFilesSet performed faster than the other approaches with an average 9.33% increased speed (for zoom level 10). ImageBlock performed, on average, 31.9% faster than SimpleFormat. Based on these benchmarks, it is logical to assume that Google's object based file system favors structured subdirectories which contain limited numbers of tiles (Mesnier et al., 2003); hence the increase of ImageBlock's performance in comparison with the benchmarks run locally. These results also enhance LevelFilesBlock's expected efficiency on the cloud since it also follows the ImageBlock's structure, as mentioned in Chapter 4. ImageBlock (and SimpleFormat) supports updating and deleting tile images easily, whereas LevelFilesSet, in its current version, does not. If performance is the main concern for developing the WTMMS, then LevelFilesSet would be the right data structure to choose. However, if the tile dataset gets updated frequently, then ImageBlock would be the most suitable solution. Google offers an easy to use deployment platform and the results are very fast, as shown in the previous section of this chapter. However, the biggest drawback of using Google Cloud is the price of the APIs, and this is discussed in the next section. A flowchart of the decision-making process can be seen in Figure 6.14.

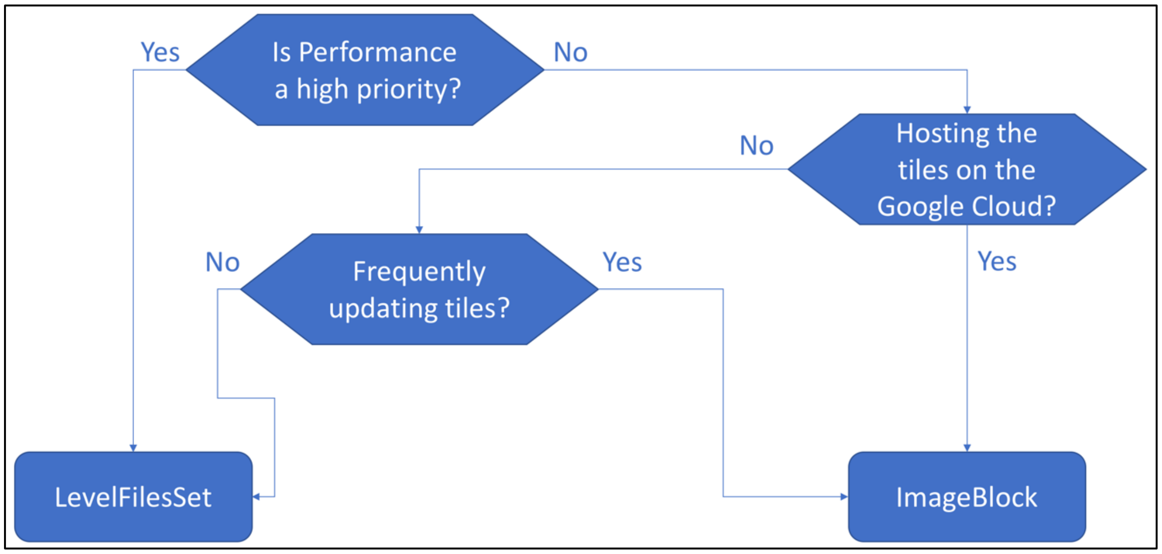
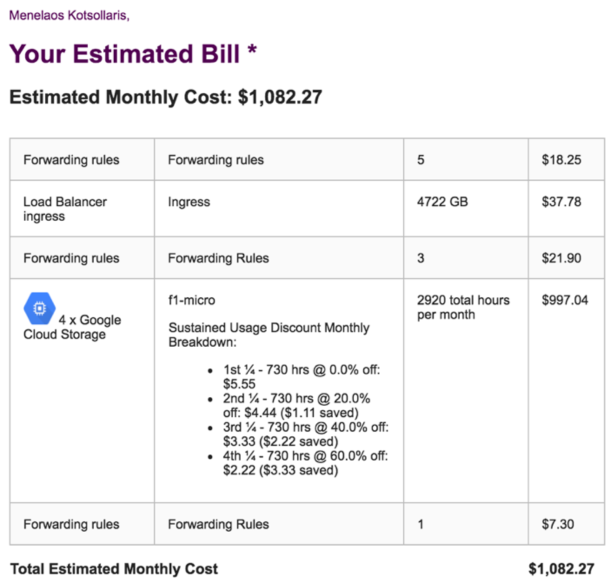

Is Performance a high priority?
Yes
No
Hosting the tiles on the Google Cloud?
No
Yes
Frequently updating tiles?
No
Yes
LevelFilesSet
ImageBlock

Figure 6.14 Flowchart for choosing the data structure

6.7. Further comments

The overall price of the benchmarks described above was about US $300. The price of Google Engine is based on multiple factors, such as the number of users, the dataset, the APIs used and so on. Running costs are a predominant concern for developers that are interested in using the Google Cloud APIs. Google offers a pricing-estimation tool that makes it easy for developers to pre-calculate the expected expense based on their usage. As previously stated, Google offers a rich number of APIs that developers can embed into their applications and configure easily. However, this does not come for free. For the average application, Google pricing tool estimates that the monthly cost will be US $1,082.27, as presented in Figure 6.15. It is highly recommended that developers, prior to taking the architectural decision of deploying their applications in the Google Cloud Platform, use the pricing tool to estimate the expected cost of the system.


Menelaos Kotsollaris,

Your Estimated Bill *

Estimated Monthly Cost: $1,082.27

Forwarding rules	Forwarding rules	5	$18.25
Load Balancer ingress	Ingress	4722 GB	$37.78
Forwarding rules	Forwarding Rules	3	$21.90
4 x Google Cloud Storage	f1-micro Sustained Usage Discount Monthly Breakdown: • 1st ¼ - 730 hrs @ 0.0% off: $5.55 • 2nd ¼ - 730 hrs @ 20.0% off: $4.44 ($1.11 saved) • 3rd ¼ - 730 hrs @ 40.0% off: $3.33 ($2.22 saved) • 4th ¼ - 730 hrs @ 60.0% off: $2.22 ($3.33 saved)	2920 total hours per month	$997.04
Forwarding rules	Forwarding Rules	1	$7.30

Total Estimated Monthly Cost **$1,082.27**

Figure 6.15 Google Cloud monthly cost estimation tool

Another significant factor to consider is the geolocation of the server. Google cloud operates only in specific locations (Figure 6.16).

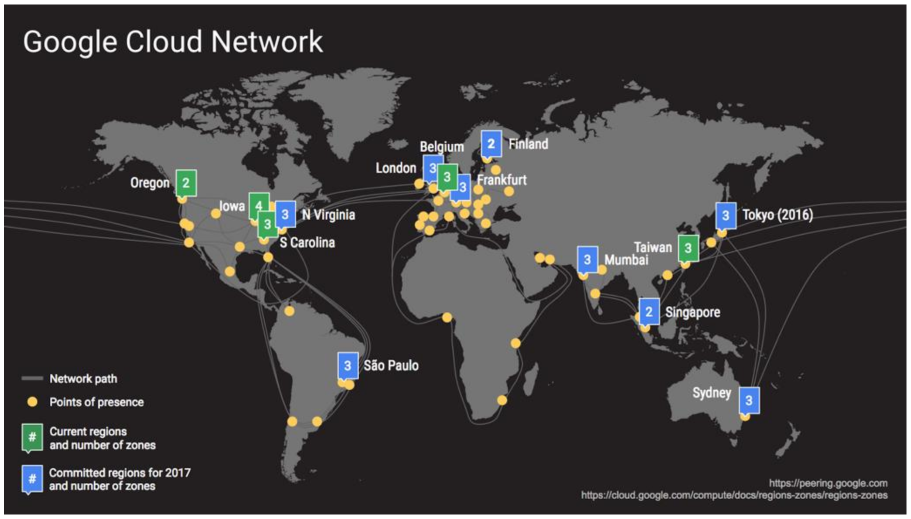

Google Cloud Network
Belgium
Finland
London
Frankfurt
Oregon
Iowa
N Virginia
S Carolina
Tokyo (2016)
Taiwan
Mumbai
Singapore
São Paulo
Sydney
Network path
Points of presence
Current regions and number of zones
Committed regions for 2017 and number of zones
https://peering.google.com
https://cloud.google.com/compute/docs/regions-zones/regions-zones

Figure 6.16 Google Cloud locations, taken from Google Cloud Platform Blog (2016)

If the majority of expected users come from a place which is not listed as a location for a potential server, then the latency is expected to be higher than for a place where Google Cloud contains multiple servers (e.g., North Virginia). The developers should consider all the mentioned issues prior to choosing to deploy their applications in Google Cloud and developing web tiled map management systems based on Google Cloud. Overall, cloud based solutions are convenient, but severely restricted by physical practicalities, which might improve in future, but currently pose limitations on usability.

7. SOFTWARE ARCHITECTURE

In designing a system, the developer must keep multiple aspects of the software architecture in mind. No one can safely predict what the trajectory of the next technological trends will be. For this reason, decision making on this level that is grounded in essential reasoning remains crucial. In this chapter, elements of software architecture (Clements et al., 2012) and software quality attributes (Offutt et al., 2002) are introduced and described in the following sections.

7.1. Programming Language

The chosen programming language is Java as it addresses all the scalability concerns, including cross-system scalability. Moreover, Java is one of the most popular programming languages and offers several libraries that are important for the development of WTMMSs, such as JDBC (see Chapter 3.5.1), which offers a user-friendly functionality in regard to database connectivity and manipulation by using ORM, image processing libraries and other useful APIs. One of the advantages of choosing a popular programming language to implement a project is that developers that might want to contribute to the project can easily participate in a later stage. Java has been established since 1995 and developers choose it because of its capabilities in OOP and cross platform systems.

7.1.1. IntelliJ IDEA

The IntelliJ IDEA is a Java integrated development environment (IDE). Although there are multiple IDEs for someone to choose from, IntelliJ was chosen based on the familiarity with the product and its constantly increasing user base. In the following diagram developed by Stackoverflow, it can be seen that IntelliJ ranks 6th in the total number of Development Environment (without counting other IntelliJ products such as PHPStorm, PyCharm; Figure 7.1):

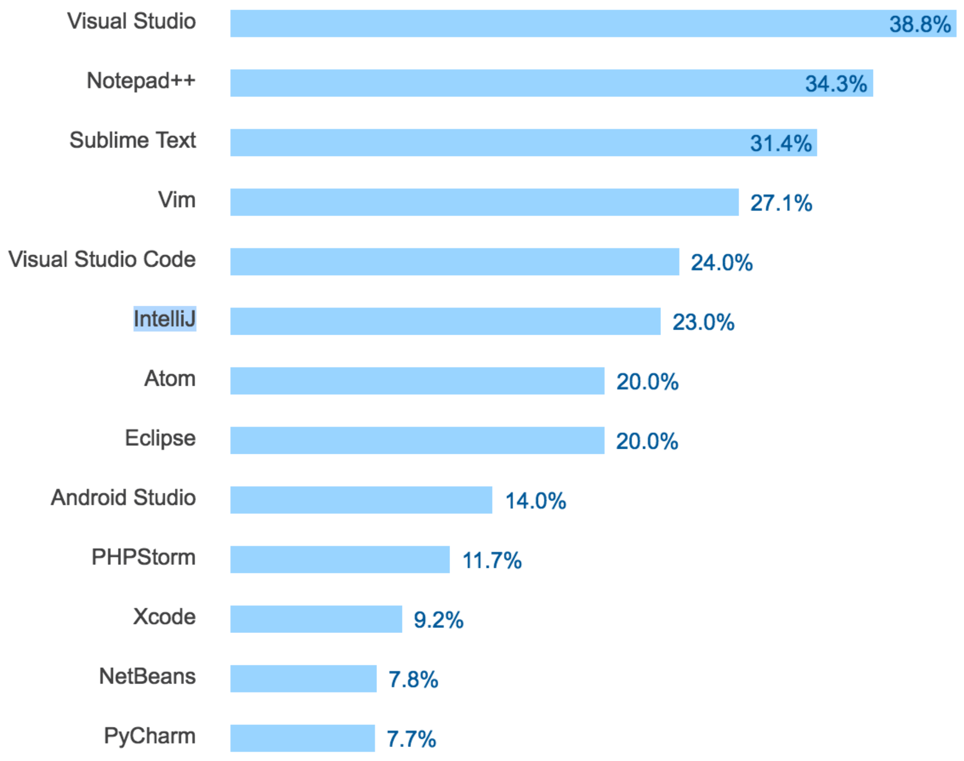


Figure 7.1 Analytics for IDE Usage (Stackoverflow Survey, 2017)

The IntelliJ IDEA provides several IDEs for the majority of programming languages. By using the IntelliJ IDEA, the code refactoring as well as the compatibility across different programming languages becomes easier to handle. Furthermore, IntelliJ supports continuous integration, sub-versioning and offers a wide range of plugins that allow developers to constantly increase their productivity.

7.2. Operating System (OS)

Just as the programming language, the system is expected to be compatible with any operating system, preferably, without affecting the performance. One question that is raised before choosing the OS is whether the system will pose an obstacle for the possible plugins and the available web servers. For instance, installing NGINX® for the caching mechanism on the backend web server eliminates the Windows® as an OS option since NGINX® is not compatible with the Windows® environment. Furthermore, several other operating systems, such as Macintosh®, Linux®, and so on are available. One of the most important requirements of the project is the ability to scale across different OSes, and that preferably, the performance of the web tiled map management system will remain consistent. As presented in the Chapter 5, the system was tested across different operating systems and the performance of the web tiled map management system remained stable, when the LevelFilesSet is used as the main data structure.

7.3. Web Server

The Web server is an important part of the application since it will maintain the core aspects of the application. While the most famous Web server is Apache because it can run on every OS, for the testing purposes of this project, the chosen server is TomCat EE (or TomEE). This offers cross operating system compatibility and modifiability when combined with Java.

7.4. Version Control

A version control characterizes the management of changes to documents and other information which allows the developer to trace changes made in the project and the development team to have a general view of the project's code (Pilato et al., 2008). Software developers use Subversion to maintain current and historical versions of files and source code to have better management of the producing code. Revision control manages the changes on the coding dataset over time and as the code changes over time, it can be structured and revised according to the developer's needs. There are multiple systems that support versioning, such as SVN and git. In this project, the SVN system was adopted because of relevant expertise and its apt applicability to the issue a hand.

7.5. System Use Cases

This section describes the use cases of each implemented component. The following use cases are the ones from the external developer, the LevelFilesSet generation, the configuration on the server by using Java Servlets, and usage from the backend and frontend developer.

7.5.1. The external developer

Figure 7.2 describes the use case of the system.

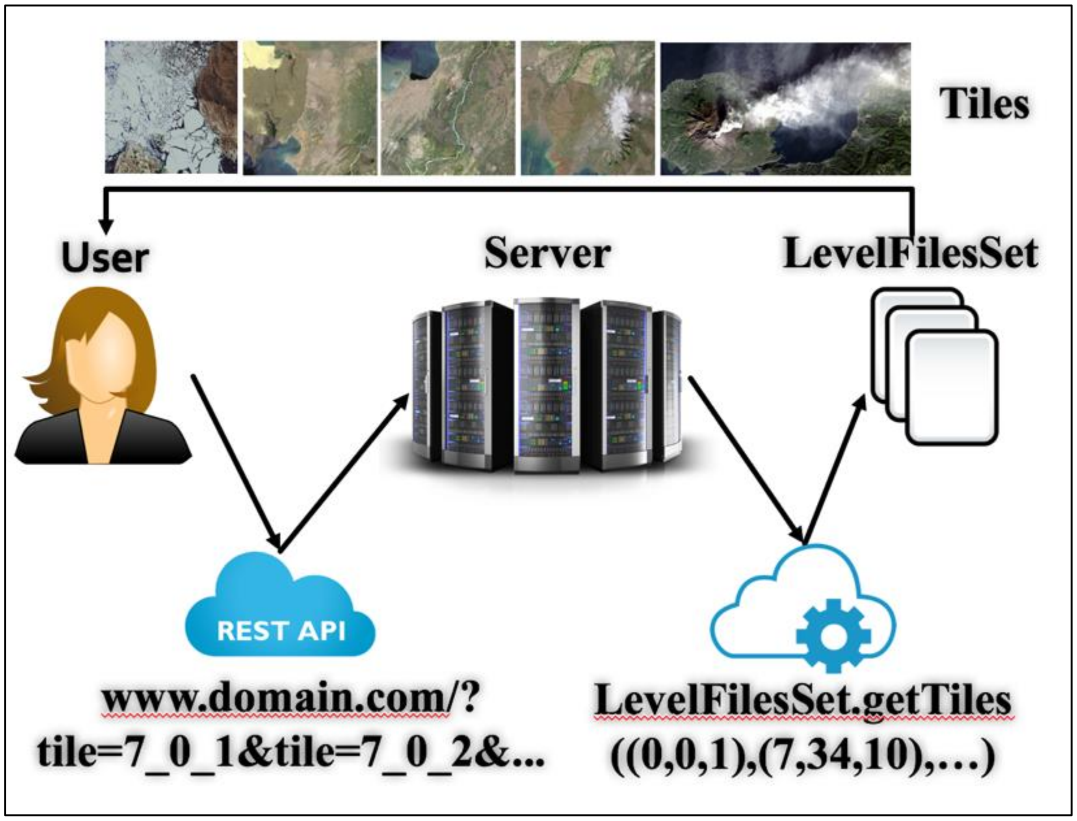
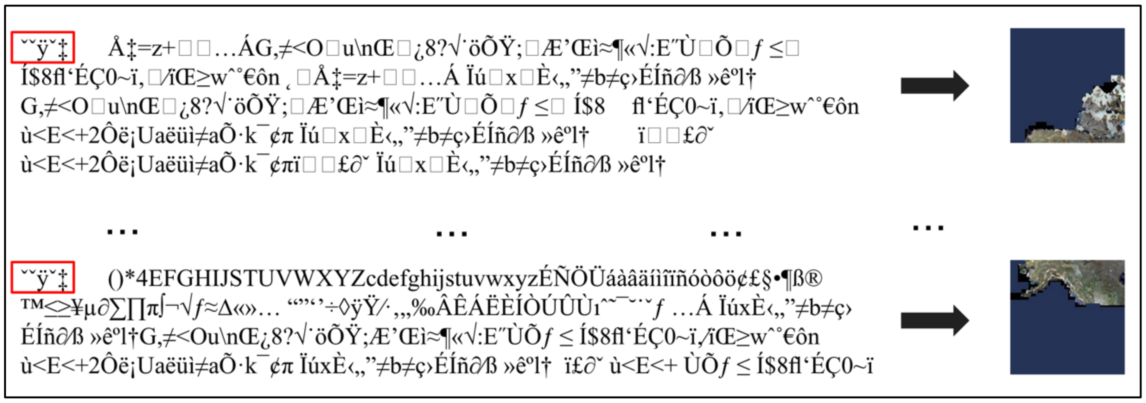


Figure 7.2 The use case of the system

Initially, the user (developer) requests certain tiles from the server. The message is processed and the server retrieves the tiles by using the LevelFilesSet data structure. Then, the server responds to the user by providing the requested images. An important aspect of this use case design is the fact that the library is encapsulated on the backend server, and no matter which of the alternative data structures is being used, the users will not have to change their actions' behavior for each method. Thus, no alteration of the frontend code is required. Figure 9 contains an example where the developer requests 2 tiles (7_0_1.jpg and 7_0_2.jpg). The server parses the request and then uses the LevelFilesSet to retrieve the tiles; following this, the server responds and the developer can retrieve the tile images. Note that it is the developer's responsibility to parse each tile accordingly. In this case, for the .jpg file the tiles can be divided by the ASCII prefix "ˇÿˇ‡" (Zhang, Y., et. al, 2008; Figure 7.3).



```
ˇˇÿˇ‡     Å‡=z+□□…ÁG,≠<O□u\nŒ□¿8?√˙öÕŸ;□Æ’Œì≈¶«√:E˝Ù□Õ□ƒ ≤□
Í$8ﬂ‘ÉÇ0~ï,□⁄ïŒ≥w˜˚€ôn ˛□Å‡=z+□□…Á Ïú□x□È‹„”≠b≠ç›ÉÍñ∂ß »êºl†
G,≠<O□u\nŒ□¿8?√˙öÕŸ;□Æ’Œì≈¶«√:E˝Ù□Õ□ƒ ≤□ Í$8    ﬂ‘ÉÇ0~ï,□⁄ïŒ≥w˜˚€ôn
ù<E<+2Ôë¡Uaëüì≠aÕ·k¯¢π Ïú□x□È‹„”≠b≠ç›ÉÍñ∂ß »êºl†       ï□□£∂ˇ
ù<E<+2Ôë¡Uaëüì≠aÕ·k¯¢πï□□£∂ˇ Ïú□x□È‹„”≠b≠ç›ÉÍñ∂ß »êºl†

        …                    …                    …                …

ˇˇÿˇ‡     ()*4EFGHIJSTUVWXYZcdefghijstuvwxyzÉÑÖÜáàâäíìîïñóòôö¢£§•¶ß®
™≤≥¥µ∂∑∏π∫¬√ƒ≈∆«»… “”‘’÷◊ÿŸ⁄·‚„‰ÂÊÁËÈÍÒÚÛÙı˜˜¯˘˙˚ƒ …Á ÏúxÈ‹„”≠b≠ç›
ÉÍñ∂ß »êºl†G,≠<Ou\nŒ¿8?√˙öÕŸ;Æ’Œì≈¶«√:E˝ÙÕƒ ≤ Í$8ﬂ‘ÉÇ0~ï,⁄ïŒ≥w˜˚€ôn
ù<E<+2Ôë¡Uaëüì≠aÕ·k¯¢π ÏúxÈ‹„”≠b≠ç›ÉÍñ∂ß »êºl†  ï£∂ˇ ù<E<+ ÙÕƒ ≤ Í$8ﬂ‘ÉÇ0~ï
```

Figure 7.3 Tile retrieval via HTTP response on the client side

The remaining use-cases present the way the LevelFilesSet is generated and configured to request multiple tiles per HTTP request.

7.5.2. The LevelFilesSet generation

In this use case, the LevelFiles are generated and stored on the web server. The prerequisite of this use-case is that the dataset must be in the storage disk for the library to read and write each tile directly onto the Lookup file and the TileDataset file (Figure 7.4). The LevelFilesSet, in its current stable version, supports only the generation of the LevelFiles. If the tiles of the system are frequently updated, then the process of the generation must be repeated each time, a procedure which inevitably imposes delays on the system.

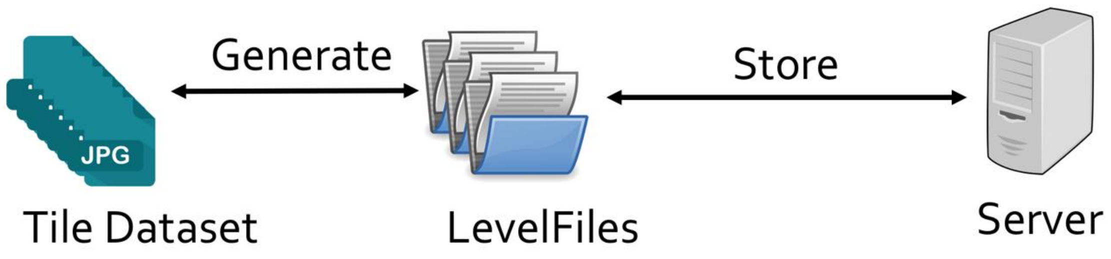


Figure 7.4 Use case of the LevelFiles Data Structure

Systems that update tile datasets frequently will need the feature of updating individual tiles instead of the whole datasets, and this is a feature that LevelFilesSet could support in later versions.

7.5.3. The LevelFilesSet configuration on the server using Java Servlets

The LevelFilesSet needs to be configured accordingly on the server side. The internal classes must be modified appropriately in regards to where the LevelFiles are stored (Figure 7.5).

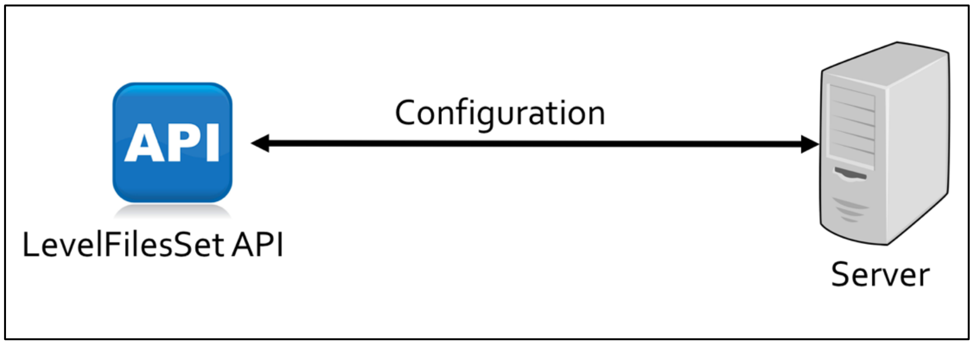


Figure 7.5 LevelFilesSet configuration on the server using Java Servlets

This action should be performed after the LevelFilesSet dataset has been generated, and thus the Java Servlet can be configured.

7.5.4. The Logical Application Tiers

The logical tiers highlight the standard architecture for distributed and large scale applications. The tiers represent a physical organization of the application's components in order to classify the available services and features of a system. The client tier contains the available end users of the system. The presentation tier consists of the Java Servlet expected components and the way the data are presented to the client tier. Furthermore, the business service tier consists of tightly coupled components that conform to the J2EE distributed component model. In this tier, the web services and standalone servers are described. Finally, in the Data tier, the main data sources under which the data are stored and retrieved are contained. The logical Tiers of the web tiled map management system can be seen in Figure 7.6.

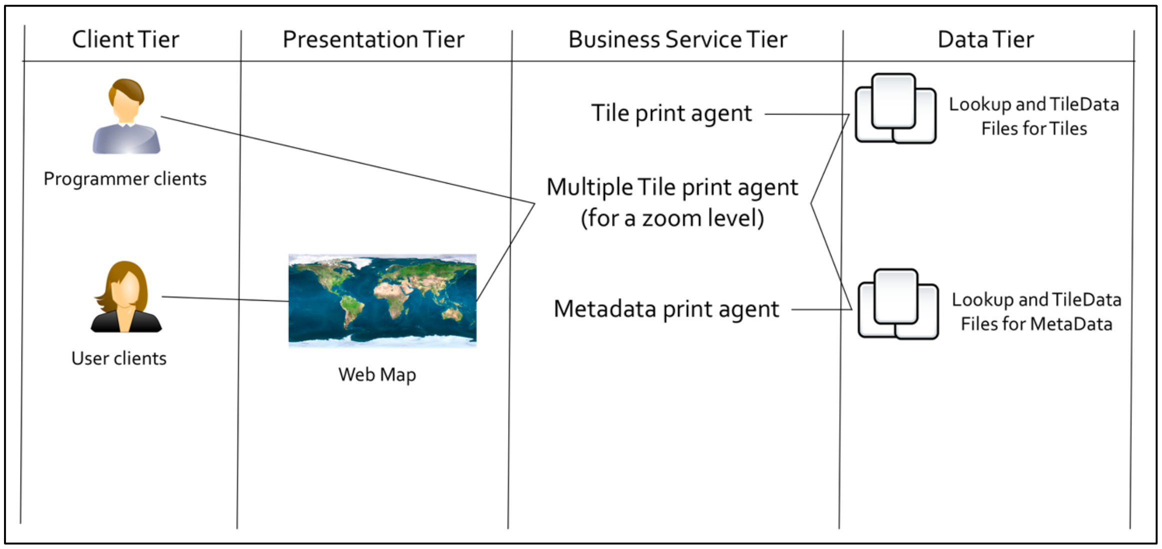


Figure 7.6 The logical application tiers

There are two main clients in this project; the programmer and user client. The programmer's goal is to make the mapping interface available for the user client to navigate through the mapping application. This includes making REST HTTP requests to the server, retrieving the requested tiles, and presenting them in the map grid. While the client navigates through the map, the underlying system runs the requests as developed by the programmer. In this design, the external programmer can retrieve Metadata by making an HTTP request to the server, similarly to the tile retrieval methods.

7.5.5. The backend developer

In this use case, the backend developer must create the Java Servlet so that when an HTTP request is made from the frontend developer (e.g., http://www.mydomain.com?tile=0_0_1) then the requested tile will be returned containing the desired raw pixels (Figure 7.7).

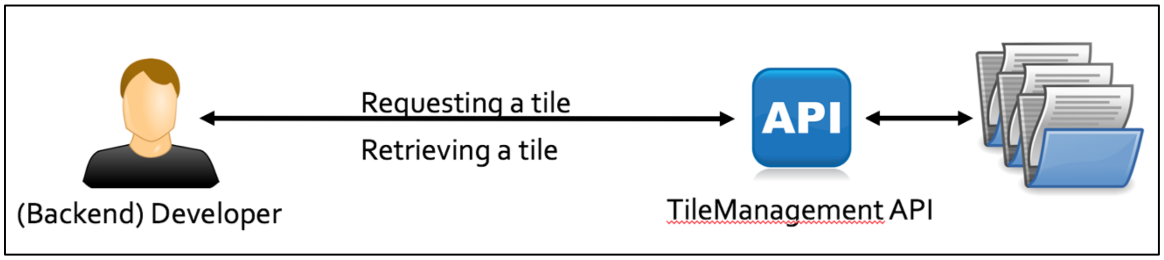


Figure 7.7 The backend developer requests and retrieves a tile by using the LeveFilesSet data structure

The Java Servlet will connect with the LevelFilesSet library (as explained above) and, once configured, it will render available the tile retrieval.

7.5.6. The Frontend Developer

It is important to note that the frontend developer is unaware of the data structure being used on the server for the tile retrieval. This means that regardless of the storage technique used on the backend, both frontend and backend developers can work independently (Figure 7.8).

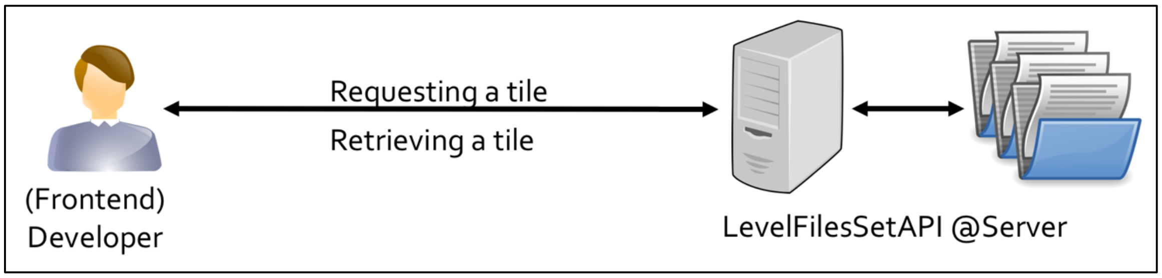


Figure 7.8 The frontend developer requests the tiles via an HTTP request

7.6. Sequence Diagrams

A sequence diagram describes the construction of messages between the objects. Specifically, a sequence diagram shows the object correlation arranged in time sequence. Sequence diagrams are very helpful in detecting crucial task latencies and improving the performance of a specific functionality. By optimizing the functionality and improving the performance, the latencies can be reduced to a minimum and thus the system can reach optimal results. This section showcases two sequence diagrams of the two main functions in LevelFilesSet; the LevelFilesSet’s generation and the tile retrieval.

7.6.1. LevelFilesSet Generation

The figure 7.9 describes the underlying sequence diagram which is used for the tile generation.

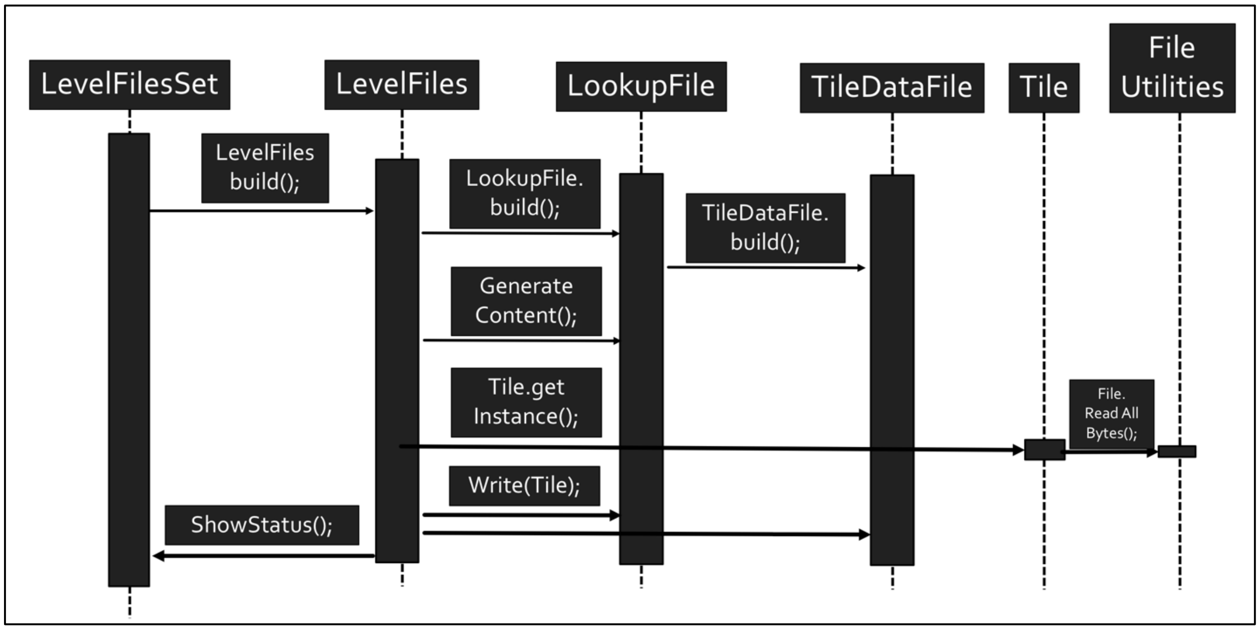


Figure 7.9 The sequence diagram for the tile generation

The LevelFilesSet uses the build function within the LevelFiles class (*LevelFiles.build*).Thereafter, the Lookup (*LookupFile.build*) and TileData files (*TileDataFile.build*) are built and the content of the tile dataset is generated by the *File.ReadAllBytes*) function provided by the Tile object. Finally, the *ShowStatus* function is the one which reveals the progress of the LevelFilesSet generation.

7.6.2. LevelFilesSet tile retrieval

Another important diagram is the one of the tile retrieval, shown in Figure 7.10.

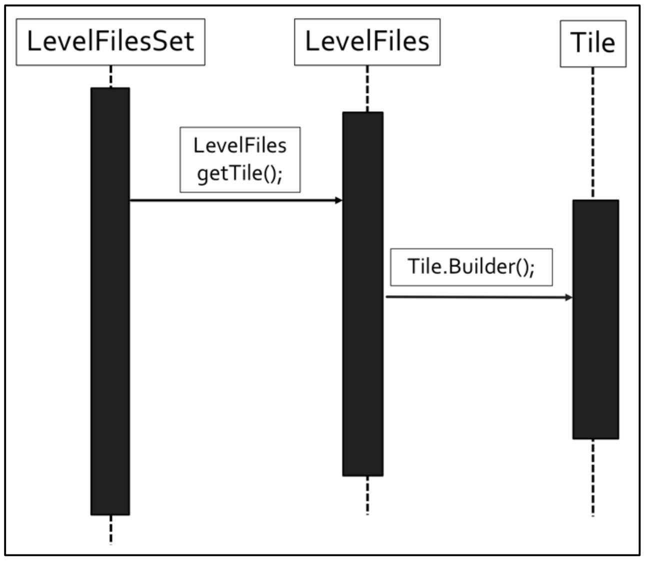


Figure 7.10 The Tile Retrieval sequence diagram

The LevelFilesSet.getTiles method uses the getTile iteratively; then the *Tile.Builder* retrieves the tile. Since this is one of the most important factors of the LevelFilesSet, it is important to highlight the potential of reducing the time needed for the runtime of the operation. If the time of tile retrieval is reduced, the LevelFilesSet's performance will be better, thus, improving the *LevelFilesSet.getTile* performance is a feature worth investigating in future versions of the data structure.

7.7. UML Class diagram

This section explains the architecture of the LevelFilesSet. The goal of this implementation was to maximize simplicity and usability. For instance, the developer should not have access to classes which are not required for direct access; that is the principle of encapsulation (Bloch, 2008). Another principle that was followed during the implementation is the one of preferring composition over inheritance; by following this principle, whenever applicable, the object's composition within the class should be preferred over the inheritance. This principle was first described by Gamma et al. (1995) where several benefits were analyzed and explained (Figure 7.11). The LevelFilesSet adapts the Static Factory pattern, as developed by Bloch, 2008. Every object must implement the Builder interface which will allow any ongoing instantiation. The Lookup and TileData files implement the singleton pattern. Since the LevelFilesSet data structure is not an in-memory data structure (Arge, 2002), then both the Lookup and TileData object hold a reference to the underlying files run through the operating system.

Each class describes the following

Public Classes:

1. LevelFilesSet: A set of LevelFiles.
2. FileNames: Contains all the file paths and the file names used to perform the I/O.
3. Tile: Represents a Tile image (file) that holds the name, level, column, row, data and the file type.

Private Classes:

4. LevelFiles: Contains the Lookup file and the TileData file and provides the necessary methods to store and retrieve the tiles into the files.
5. TileDataFile: Represents a file where the tiles' bytes are stored.

6. LookupFile: The file containing the pointer to the TileDataFile and the size of each tile.
7. Utilities: Contains static methods used within the framework.
8. MyFileNames: Enum for instantiating the FileNames class' parameters.
9. PerformanceTests: Includes the Performance Tests that measure the LevelFilesSet's API efficiency over the others.

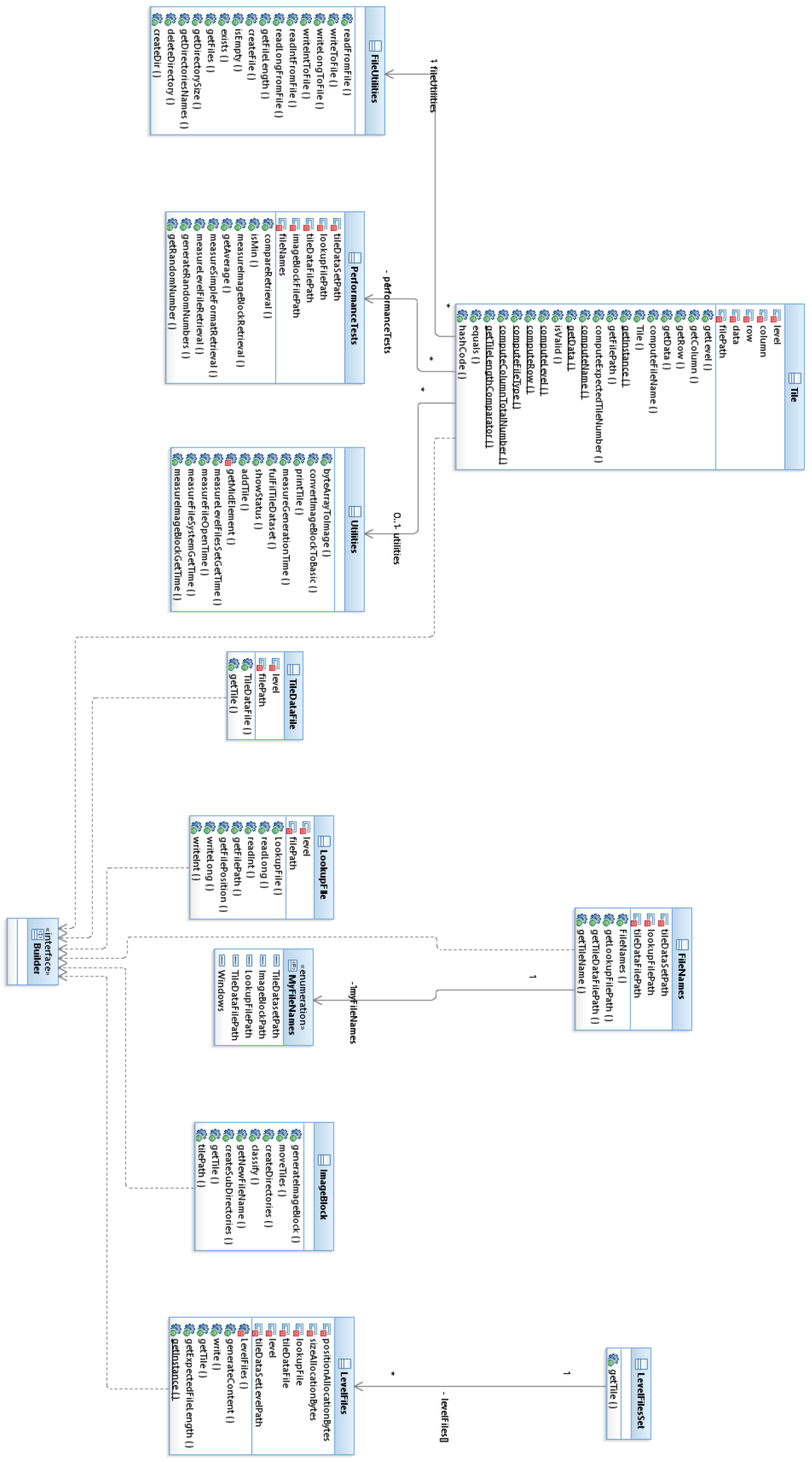


Figure 7.11 The LevelFilesSet's UML diagram

The developer initially has to set up the paths where he stores the dataset (tileDataSetPath, lookupFilePath, tileDataFilePath) and encapsulate the information in the fileNames variable. Once the fileNames is instantiated, it is passed to the LevelFilesSet class so it can be instantiated. The developer then can retrieve the tile by calling the getTile(Z,X,Y) method, where Z the zoom level, X the column number and Y the row number. The code of the above procedure is highlighted in Figure 7.12.

```
//Change the file paths with your values
//the tile dataset path
 String tileDataSetPath = "/Users/mkotsollaris/Desktop/tile_dataset_test/tile_dataset";
 //where the lookupFile is going to be stored
 String lookupFilePath = "/Users/mkotsollaris/Desktop/tile_dataset_test/LevelFiles/LookupFile";
 //where the tileDataFile is going to be stored
 String tileDataFilePath = "/Users/mkotsollaris/Desktop/tile_dataset_test/img_block";
FileNames fileNames =  new FileNames.Builder(tileDataSetPath, lookupFilePath,
                         tileDataFilePath).build();
LevelFilesSet levelFilesSet = new LevelFilesSet.Builder(fileNames).build();
levelFilesSet.getTile(0,0,0); //returns the "0_0_0.jpg" tile
```

Figure 7.12 Using LevelFilesSet

8. CONCLUSIONS AND RECOMMENDATIONS

8.1. Conclusions

The purpose of this research was to determine which data structure provided the most scalable and efficient system under which tile images could be stored and retrieved. In conducting this research, four tile storage solutions (database, SimpleFormat, ImageBlock, and LevelFilesSet) were chosen and implemented from scratch. The database solution offered the slowest results out of the alternative solutions; databases could be scaled across different systems but their performance was slow compared to the other techniques. During the locally performed benchmarks, the file system based solutions (SimpleFormat and ImageBlock) performed better than the database approach. Further, in these local benchmarks, SimpleFormat performed faster than the ImageBlock solution. This performance was explained due to the impact of the exponential growth of the subdirectories within the increasing number of zoom-levels. That is, the more the sub-directories within a zoom level, the greater the latencies on the locally-tested file systems, such as NTFS and ExFAT.

Different results were observed when the cloud-based benchmarks were performed. Specifically, the Google Cloud object-based file system favored the structured subdirectories and the ImageBlock performance was greater than its performance tested in the local environment. For ImageBlock to provide efficient results, a balance should be kept between the number of sub-divided directories (which should be the minimum applicable) and the number of tiles within the subfolder (which should be the maximum). On the contrary, LevelFilesSet provides the fastest performance and scales under any system. Importantly, this comes with drawbacks such as not supporting dynamic tile adding and deletion within a zoom level. If a tile needs to be replaced, then LevelFilesSet has to generate the tiles for the entire zoom level. Furthermore, in its current version, LevelFilesSet stores the entire tile dataset in two single files which renders tile serving for zoom levels more than 15 not scalable. A hybrid combination of ImageBlock and LevelFilesSet, named LevelFilesBlock, is proposed to take advantage of LevelFilesSet's performance superiority as well as ImageBlock's elegance and scalability on distributing the tiles across different storage systems. With the LevelFilesSet logic being applicable to any type of data storage, an ambitious extension could be the implementation of a generic form that supports information storage of any type. Another potential refinement would be examining the delays between the communication of the objects (e.g., TileData file with Lookup file). If these delays are reduced, LevelFilesSet could produce more efficient results.

In the long term, adoption of this data structure can transcend the limitations imposed by different environments, while improving upon the speed and efficiency of existing system-specific solutions. Moreover, the software architecture of the project was aimed to be as scalable as possible. The initial design contained inheritance as well as other decisions that were not following the principles and guidelines of the large-scale enterprises; the second version of the architecture was improved and follows most of the design architectural patterns, as proposed by Bloch (2008) and Gamma et al. (1994). Both scalability and performance were taken into account while the system was developed from scratch. The LevelFilesSet was designed and implemented and additional features were suggested for its further development. In the end, the proposed backend server seems to be able to handle large volumes of users. In Chapter 8.2, additional features are suggested that could prove to be valuable additions to the system.

Furthermore, for the web tiled map management system to be consistently scalable, emphasis was placed on how the software architecture could be implemented to render the system efficient across different operating systems with different specifications. Scalability and longevity of software systems is and has been always a field that researchers have tried to investigate and find solutions for (Weyuker et al., 2002). Even though software design patterns are time consuming and difficult to implement across large-scale systems, they are worth developing and can be rewarding. For instance, Test-Driven-Development (Halili, 2008) is one of the emerging techniques in developing software. However, the steep learning process of TDD (Pilato, 2008) makes it hard for the developers to adopt it into their frameworks. Another concern would be vertical scalability versus horizontal scalability (Wessels, 2001). Vertical scalability refers to increasing the capacity of the already existing systems, whereas horizontal scalability refers to the ability to add further systems (Figure 8.1).

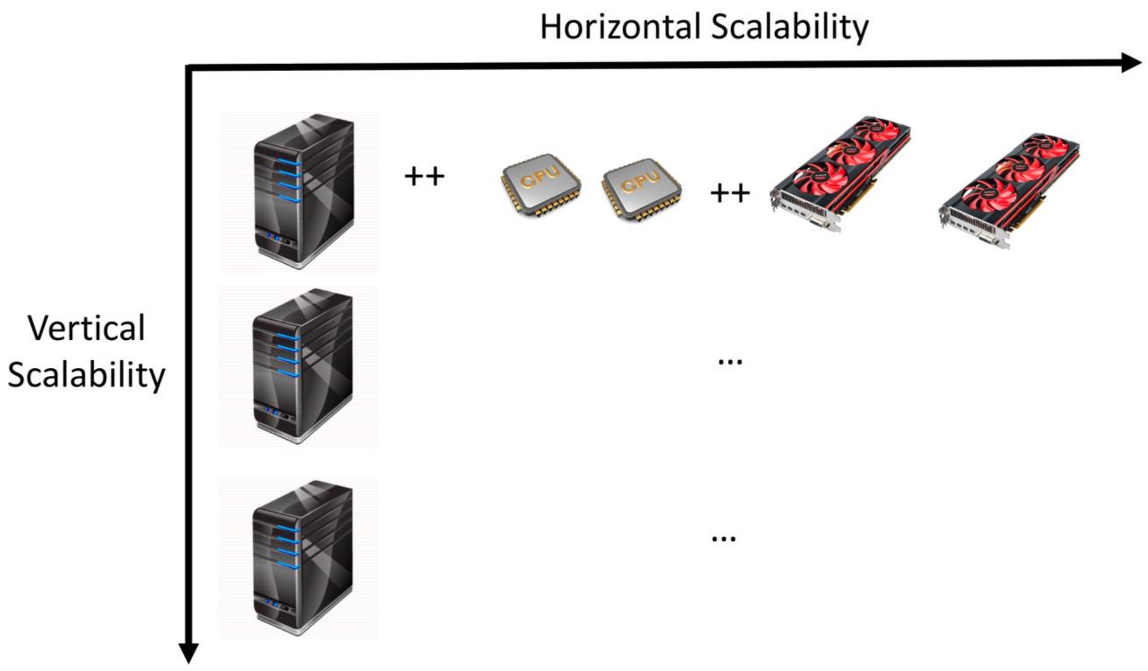


Figure 8.1 Horizontal and Vertical Scalability

The ideal system would be an error prone system with the ability to add limitless new features without interoperability or any other issues. The perfect system does not exist because of the dynamic requirements and the physical constraints of hardware at any given time point. Thus, it is important for the developers to continuously keep building, upgrading, and optimizing their systems, fixing errors of the previous versions and evolving the system to increase its efficiency and performance. When building a web tiled map management system, the developer must ask (and provide answers to) the following questions:

a. What will be the tiling scheme? (presented in Chapters 1 and 2).
b. What data structure will be used for tile storage and retrieval? (presented in Chapters 3 and 4).

c. How will the system react to a large load of user requests? (presented in Chapters 5 and 6).

d. What will be the specifications of the system? (presented in Chapter 7).

This thesis provided a thorough analysis of the principles and guidelines that must be taken into consideration in order to implement a scalable and efficient web tiled map management system. In the following section, additional topics are suggested for future research.

8.2. Future Research

This research was based primarily on analyzing and implementing optimal data structures for tile storage and retrieval. During the course of this project, additional research topics were identified, and are as follows:

- As presented in the Chapter 4.5, the LevelFilesSet data structure could be used to store Metadata that contain useful information for the tiles. By storing Metadata information, authors can be credited and the source of the tile datasets becomes available to the users. This feature enhances the capabilities of tile web server and follows the necessary guidelines of data acknowledgement.
- The LevelFilesSet can be extended to support any information storage. Systems that frequently access files and are heavily dependent on the file system's

performance can be benefited by a fast data structure. For the LevelFilesSet to support such a level of abstraction, a thorough analysis of how such a data structure could be developed must take place. The key component for why the LevelFilesSet can support tile storage is the fact that the size of the tile is known and thus the pixels can be stored in external files and accessed accordingly. Other domains such as Image Management systems could benefit from using LevelFilesSet-based approach for data storage, when performance and speed are concerns.

- For the local benchmarks, the reason why ImageBlock has worse results than SimpleFormat, is primarily due to the fact that the subdirectories introduce latencies and thus the greater the number of the subdirectories, the longer the time of retrieval. A way of reducing the number of retrievals would be to increase the tile volume limitation per subdirectory, for instance, increasing the limitation from 1024 to a million and if the average tile size is 4KB, then the expected size of the subdirectory would be 1GB. Depending on the average tile size, a balance between the size of each sub-directory and the ability to distribute the tiles across multiple systems can be achieved. In the best case, an average of 64GB per LevelFilesSet (within each subdirectory) would be an ideal outcome since the volume significantly decreases the number of the subdirectories and allows the storage across multiple systems.
- The current version of LevelFilesSet supports tile generation but not adding and deleting tiles dynamically. A system which frequently updates its tiles will need that feature. For supporting this feature, the implementation of the LevelFilesSet

should be upgraded to a more sophisticated version, as proposed by Sample et al. (2010). By storing an extra file which keeps the pointers of each tile, the LevelFilesSet can modify (add/delete) each tile separately. However, this feature is expected to increase the memory complexity of the systems since additional pointers must be stored, as presented in Chapter 3.

- LevelFilesBlock needs to be implemented and validated. Its hybrid approach on combining LevelFilesSet, which is the fastest available solution and ImageBlock, which is the most modifiable and scalable solution for distributing tiles across different storage systems, make it an efficient data structure worth investigating further.
- Several caching techniques, on top of LevelFilesSet, could be implemented on the server side that could significantly boost the performance of tile retrieval. While these caching abstractions are beyond the scope of this thesis project, proposals in Web Caching, from Gupta (2010) and Wessels (2001), would introduce a significant performance improvement when the external developer makes an HTTP request to retrieve one, or multiple, tile images.
- An analysis of the possible network architectures must be developed to improve the results of the tile retrieval. As stated in Chapters 5 and 6, the HTTP overhead causes further latencies on the system. Other networking solutions, such as Web-sockets, are expected to not have such delays since once the connection is established between the client and the server then there is no need to re-validate the client's information. Improving the network performance is crucial and thus a

robust benchmark analysis is expected to be useful in regard to scalability and performance improvement.

- While the number of users increases, the need of supporting load-balancing techniques across multiple servers becomes vital. Tools like Redis®, Memcached® and so on provide a user-friendly approach to load balancing and thus a benchmark between the available tools could provide the benefits and the drawbacks of each technique.
- LevelFilesSet data structure could be implemented for database management support. As highlighted in the Benchmark Results chapter, the performance of the databases provides the worst results amongst all the other techniques. By providing a plugin which implements the LevelFilesSet within the database-management-systems, tiles can be efficiently stored and retrieved and thus developers could consider the database option valuable.
- The LevelFilesSet could be implemented for different platforms. This can be achieved by either implementing it from scratch or using tools. For instance, an available tool for importing Java JAR documents to C# is IVKM. Other tools alike IVKM.NET exist for different programming languages.
- The dataset across different web tiled map management systems varies. Benchmarks regarding the best tile size and format could prove to be very helpful in improving performance. The dataset tested in this research were images of 256x256 pixels in the JPG format. Other image formats, such as PNG, SVN and so on, are widely used and thus an investigation of these types could prove to be useful.

- Multiple vendors, such as ESRI, Microsoft, Google, offer tile-hosting solutions. These vendors offer multiple APIs and user-friendly functionalities. An interesting topic for further investigation is to examine whether the performance of the data structures examined in this research remains the same across these vendors. Every individual vendor is not expected to use the same file system; thus, re-examining the performance on the cloud would provide a useful insight in regard to choosing the vendor, in case the system is not hosted on the developer's cloud.
- More design patterns, as mentioned by Gamma et al. (1994) and Bloch (2008), could be implemented for the aforementioned data structures. The implementation does significantly affect the performance of the system. If these data structures are implemented by following the architectural and low level design patterns, then the performance is expected to be better than currently.
- Big data frameworks could be tested and combined with the web tiled map management system's functionality. For instance, the Hadoop or Spark framework can be used to access the distributed tiles of the LevelFilesBlock across different systems. The benefit of using these frameworks is that they are frequently used by multiple developers and thus they provide a large volume of accessible APIs that can be used.
- Apart from Google Cloud, other vendors such as Amazon EC2, Microsoft Azure and ESRI provide cloud support for storing and retrieving tiles. A benchmark among these vendors would be useful to investigate the performance of the mentioned systems.

- Multiple libraries will be added with the release of Java 9 (OpenJDK, 2017). One of the libraries is the Java Microbenchmarking Harness. JMH is a Java harness for building, running, and analyzing Nano/Micro/Milli/Macro benchmarks. When it comes to accurate benchmarking, there are forces in play like warmup times and optimizations that can have a significant impact on results, especially when the accuracy is going down to micro and Nano seconds. JMH is expected to be the best choice to extract accurate results and run benchmarks in Java.
- Java 9 is expected to fully support HTTP 2.0 SPDY which has already shown great speed improvements over HTTP 1.1 ranging from 11.81% to 47.7% (Google SPDY 2017) and its implementation already exists in most modern browsers. Java 9 will have full support for HTTP 2.0 and feature a new HTTP client for Java that will replace HttpURLConnection, and implement HTTP 2.0 and websockets, which will allow for further benchmarking options in examining network performance.
- LevelFilesSet could be implemented in a faster language than Java, like C++ or Scala. If developed correctly, these languages will offer programming techniques to accelerate the calculations and tile retrieval and this would increase the performance of LevelFilesSet.
- LevelFilesSet could be extended to work in-memory and thus allow data storage in RAM. This feature is expected to prove useful to caching techniques applied on the server side (Gupta, 2010).
- LevelFilesSet could be extended to support on-the-fly tile update. In the current form, in the case of a tile getting updated, LevelFilesSet must generate all the tiles

of the level. In such a scenario, it would be logical to implement LevelFilesSet as a file-system itself since the mentioned features would simulate a file system behavior.

- Additional hardware components could be examined to check their efficiency. For instance, how would an AMD CPU perform against an Intel CPU, an NVIDIA GPU with an Intel GPU and others.
- Based on this research, a file system that imitates the behavior of LevelFilesSet which is specialized in providing efficient tile storage and retrieval could be developed. This file system must efficiently retrieve tile images with optimal performance and also make it possible for other type of data to be stored.
- Apart from Google Cloud, several other vendors, such as Microsoft Azure, ESRI and Amazon S3 support cloud-based storage. Further examination among these vendors could prove to be useful for deciding which offers the most efficient tile storage among different tile structures.
- An analysis of the differences between different network protocol, such as TCP/IP, HTTP and so on, could provide significant information of which network protocol is the most efficient for tile retrieval.

In conclusion, the best solution needs to be tailored to the given needs of the developer. This research provides relevant evidence of a systems engineering approach that can guide the implementation of efficient web tiled map management systems followed by scalable data structures and technologies that are optimal and render tile serving efficient for large numbers of users.

REFERENCES

Abreu, F. B., & Carapuça, R. (1994, October). Object-oriented software engineering: Measuring and controlling the development process. In Proceedings of the 4th international conference on software quality (Vol. 186, pp. 1-8).

Amazon SE, URL: https://aws.amazon.com/se3/

Apache Subversion, URL: https://subversion.apache.org/

Argyraki, K., & Cheriton, D. R. (2009). Scalable network-layer defense against internet bandwidth-flooding attacks. IEEE/ACM Transactions on Networking (ToN), 17(4), 1284-1297.

AWS Terrain Tiles, URL: https://aws.amazon.com/public-datasets/terrain/

Arge, L. (2002). External memory data structures. In Handbook of massive data sets (pp. 313-357). Springer US.

Azevedo, L. G., Santoro, F., Baião, F., Souza, J., Revoredo, K., Pereira, V., & Herlain, I. (2009). A method for service identification from business process models in a SOA approach. In Enterprise, Business-Process and Information Systems Modeling (pp. 99-112). Springer Berlin Heidelberg.

Barish, G., Obraczka, K., Way, A., & Rey, M. (2000). World Wide Web Cahing: Trends and Techniques. *IEEE Communications Magazine*, *38*(5), 178–184.

Batty, M., Hudson-Smith, A., Milton, R., & Crooks, A. (2010). Map mashups, Web 2.0 and the GIS revolution. Annals of GIS, 16(1), 1-13.

Blower, J. D. (2010, June). GIS in the cloud: implementing a Web Map Service on Google App Engine. In Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research & Application (p. 34). ACM.

Boldi, P., Codenotti, B., Santini, M., & Vigna, S. (2004). UbiCrawler: A scalable fully distributed Web crawler. *Software - Practice and Experience*, *34*(8), 711–726. http://doi.org/10.1002/spe.587.

Breslau, L., Cao, P., Fan, L., Phillips, G., & Shenker, S. (1999). Web caching and Zipf-like distributions: evidence and implications. *Ieee Infocom, 1*, 126–134 http://doi.org/10.1109/INFCOM.1999.749260.

Buyya, R., Yeo, C. S., & Venugopal, S. (2008, September). Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on (pp. 5-13). IEEE.

Chapter 2 Java Enterprise System Architecture, URL: https://docs.oracle.com/cd/E19263-01/817-5764/architecture.html#wp22243

Choosing A Proxy Server (2014), ApacheCon, Bryan Call.

Class RandomAcessFile, Oracle, URL: https://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.html.

Clements, P. C. (2002). Software architecture in practice (Doctoral dissertation, Software Engineering Institute).

Curl, command line tool and library for transferring data with URLs, URL: https://curl.haxx.se/

Dimension 2: Logical Tiers (Sun Java Enterprise System 5 Technical Overview), URL: https://docs.oracle.com/cd/E19528-01/820-0167/aauba/index.html

Dixon, M., & Hamilton, M. (2006). U.S. Patent Application No. 12/088,116.

Dutilleul, P. (1999). The MLE algorithm for the matrix normal distribution. Journal of statistical computation and simulation, 64(2), 105-123.

Fan, L., Cao, P., Almeida, J., & Broder, A. Z. (2000). *Summary cache: A scalable wide-area Web cache sharing protocol*. *IEEE/ACM Transactions on Networking* (Vol. 8) http://doi.org/10.1109/90.851975.

Ganger, G. R., & Patt, Y. N. (1994, November). Metadata update performance in file systems. In Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation (p. 5). USENIX Association.

Chicago

Git Tool, URL: https://git-scm.com/

Google Buckets, URL:

Google Cloud Engine, URL: https://cloud.google.com/compute/

Google Cloud Platform Blog, Brian Stevens (2016), Vice President of Google Cloud, URL: https://cloudplatform.googleblog.com/2016/09/Google-Cloud-Platform-sets-a-course-for-new-horizons.html

Google Cloud Storage Documentation, URL: https://cloud.google.com/storage/docs/

Google Maps for Work, "Serving raster layers on Google Cloud Platform", URL: https://storage.googleapis.com/support-kms-prod/57E913590B6A12FED962C112597B54F99DA7

Google Maps Structure, MicroImages (2010), URL:
http://www.microimages.com/documentation/TechGuides/78googleMapsStruc.pdf

Gupta, P. (2010). *Providing caching abstractions for web applications* (Doctoral dissertation, Massachusetts Institute of Technology).

Hamilton, G., Cattell, R., & Fisher, M. (1997). JDBC Database Access with Java (Vol. 7). Addison Wesley.

Heidemann, J., Obraczka, K., & Touch, J. (1997). Modeling the performance of HTTP over several transport protocols. IEEE/ACM Transactions on Networking (TON), 5(5), 616-630.

Henning, J. L. (2000). SPEC CPU2000: Measuring CPU performance in the new millennium. Computer, 33(7), 28-35.

Henning, J. L. (2006). SPEC CPU2006 benchmark descriptions. ACM SIGARCH Computer Architecture News, 34(4), 1-17.

Heydon, A., & Najork, M. (1999). Mercator: A scalable, extensible Web crawler. *World Wide Web, 2*(4), 219–229. http://doi.org/10.1023/A:1019213109274.

HTTP Range Requests, Mozilla Developer Network, URL:
https://developer.mozilla.org/en-US/docs/Web/HTTP/Range_requests

Housel III, B. C. (2003). U.S. Patent No. 6,535,869. Washington, DC: U.S. Patent and Trademark Office.

Hundt, R. (2011). Loop Recognition in C++/Java. *Go/Scala*.

IKVM.NET Tool, URL: https://www.ikvm.net/

Information Technology – Digital Compression and Coding of Continuous-Tone Still Images – Requirements and Guidlines, T. (1993). CCITT T. 81.

J2EE Oracle, URL: http://www.oracle.com/technetwork/java/javaee/overview/index.html

Java EE, Oracle Documentation, URL (2000): http://docs.oracle.com/javaee/6/tutorial/doc/bnafe.html

Java SE Documentation, URL: https://docs.oracle.com/javase/tutorial/jdbc/overview/architecture.html

Java Servlet Tutorial, URL: https://www.tutorialspoint.com/servlets/images/servlet-lifecycle.jpg

Karger, D., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Yerushalmi, Y. (1999). Web caching with consistent hashing. *Computer Networks*, *31*(11), 1203–1213 http://doi.org/10.1016/S1389-1286(99)00055-9.

Lin, K. J., & Richard, G. (2003). Web services computing: advancing software interoperability.

Lua, E. K., Crowcroft, J., Pias, M., Sharma, R., & Lim, S. (2005). A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, *7*(2), 72–93. http://doi.org/10.1109/COMST.2005.1610546.

MapTiler: Coordinates, Tile Bounds and Projection, URL: http://www.maptiler.org/google-maps-coordinates-tile-bounds-projection/

McGrath, H. (2014). *Historical Maps of Grand Lake Meadows* (Doctoral dissertation, UNIVERSITY OF NEW BRUNSWICK).

Mesnier, M., Ganger, G. R., & Riedel, E. (2003). Object-based storage. IEEE Communications Magazine, 41(8), 84-90.

MS-EFSR: Encrypting File System Remote (EFSRPC) Protocol. Microsoft. 14 November 2013.

Nagappan, N., Maximilien, E. M., Bhat, T., & Williams, L. (2008). Realizing quality improvement through test driven development: results and experiences of four industrial teams. Empirical Software Engineering, 13(3), 289-302.

Offutt, J. (2002). Quality attributes of web software applications. IEEE software, 19(2), 25-32.

OpenJDK JDK 9, URL: http://openjdk.java.net/projects/jdk9/

Open Data Canada, URL: http://open.canada.ca/en/open-data

Phansalkar, A., Joshi, A., Eeckhout, L., & John, L. K. (2005, March). Measuring program similarity: Experiments with SPEC CPU benchmark suites. In Performance Analysis of

Quinn, S., & Gahegan, M. (2010). A predictive model for frequently viewed tiles in a web map. Transactions in GIS, 14(2), 193-216.

Rajlich, V. T., & Bennett, K. H. (2000). A staged model for the software life cycle. Computer, 33(7), 66-71.

Rosenblum, M., & Ousterhout, J. K. (1992). The design and implementation of a log-structured file system. ACM Transactions on Computer Systems (TOCS), 10(1), 26-52.

Rowstron, A., & Druschel, P. (2001). Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *ACM SIGOPS Operating Systems Review*, *35*(5), 188. http://doi.org/10.1145/502059.502053.

Savage, S. J., & Waldman, D. M. (2009). Ability, location and household demand for Internet bandwidth. International Journal of Industrial Organization, 27(2), 166-174.

Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on (pp. 10-20). IEEE.

Sears, R., Van Ingen, C., & Gray, J. (2007). To blob or not to blob: Large object storage in a database or a filesystem?. *arXiv preprint cs/0701168*.

Sen, S., & Wang, J. (2004). Analyzing Peer-To-Peer Traffic, *12*(2), 219–232.

Small, T., Li, B., & Liang, B. (2007). Outreach: Peer-to-peer topology construction towards minimized server bandwidth costs. IEEE Journal on Selected Areas in Communications, 25(1).

SPDY: An experimental protocol for a faster web (2010), URL: http://dev.chromium.org/spdy/spdy-whitepaper.

Stackoverflow Developer Survey Results 2017, URL: http://stackoverflow.com/insights/survey/2017/?utm_source=so-owned&utm_medium=hero&utm_campaign=dev-survey-2017&utm_content=hero-ind-ques

Weyuker, E. J., & Avritzer, A. (2002). A metric to predict software scalability. In Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on (pp. 152-158). IEEE.

Wieczorek, J., Guo, Q., & Hijmans, R. (2004). The point-radius method for georeferencing locality descriptions and calculating associated uncertainty. International journal of geographical information science, 18(8), 745-767.

Yee, K. P., Swearingen, K., Li, K., & Hearst, M. (2003, April). Faceted metadata for image search and browsing. In Proceedings of the SIGCHI conference on Human factors in computing systems (pp. 401-408). ACM.

Zhang, Y., Li, D., & Zhu, Z. (2008). A server side caching system For efficient web map services. *Proceedings - The 2008 International Conference on Embedded Software* *and Systems Symposia, ICESS Symposia*, 32–37

http://doi.org/10.1109/ICESS.Symposia.2008.39.

Bibliography

Bloch, J. (2008). Effective java (the java series). Prentice Hall PTR.

Clements, P. C. (2002). Software architecture in practice (Doctoral dissertation, Software Engineering Institute).

Craig, R. D., & Jaskiel, S. P. (2002). Systematic software testing. Artech House.

Gamma, E. (1995). Design patterns: elements of reusable object-oriented software. Pearson Education India.

Dombrowski, M. (2017). Personal communication. Dombrowski Marcel, Web Application Software Development, Geodesy and Geomatics Engineering Dept., University of New Brunswick, New Brunswick, Canada, May.

Du, W. (2017). Personal communication. Dr. Weichang Du, Professor, Computer Science Dept., University of New Brunswick, New Brunswick, Canada, May.

Fraser, D. (2017). Personal communication. David Fraser, Mapping Technologist, Geodesy and Geomatics Engineering Dept., University of New Brunswick, New Brunswick, Canada, May.

Halili, E. H. (2008). Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites. Packt Publishing Ltd.

Hamilton, G., Cattell, R., & Fisher, M. (1997). JDBC Database Access with Java (Vol. 7). Addison Wesley.

Mak, G., Rubio, D., & Long, J. (2010). Spring Recipes: A problem-solution approach. Apress.

Pilato, C. M., Collins-Sussman, B., & Fitzpatrick, B. (2008). Version control with subversion. " O'Reilly Media, Inc.".

O'Sullivan David (2017). Personal communication. Review of journal paper in International Journal of Geographical Information Systems (IJGIS). Dr. O'Sullivan David, Professor in UC Berkeley, April

Sample, J. T., & Ioup, E. (2010). Tile-based geospatial information systems: principles and practices. Springer Science & Business Media.

Shaw, M., & Garlan, D. (1996). Software architecture: perspectives on an emerging discipline (Vol. 1, p. 12). Englewood Cliffs: Prentice Hall.

Stefanakis, E. (2017). Personal communication. Dr. Emmanuel Stefanakis, Associate Professor, Geodesy and Geomatics Engineering Dept., University of New Brunswick, New Brunswick, Canada, May.

Stefanakis, E., Course Notes, GGE5403, php:hypertextpreprocessor, 2012

Wessels, D. (2001). Web caching. " O'Reilly Media, Inc.".

William, L. (2017). Personal communication. William Liu, Research Technician, Geodesy and Geomatics Engineering Dept., University of New Brunswick, New Brunswick, Canada, May.

Zhang, Y. (2017). Personal communication. Dr. Yun Zhang, Professor, Geodesy and Geomatics Engineering Dept., University of New Brunswick, New Brunswick, Canada, May.

Appendix I: LevelFilesSet Implementation Code

I.1. FileNames class

```
package unb.mkotsollaris.tilemanagement;

import java.io.File;
import java.util.ArrayList;

/**
 * Contains all the file paths and the file names.
 *
 * @author mkotsollaris
 * @since 1.0
 */
final public class FileNames
{
   /** The tile dataset full directory path */
   private final String tileDataSetPath;
   /** The LookupFile path */
   private final String lookupFilePath;
   /** The TileData path */
   private final String tileDataFilePath;

   private FileNames()
   {
      throw new AssertionError();
   }

   private FileNames(Builder builder)
   {
      this.lookupFilePath = builder.lookupFilePath;
      this.tileDataFilePath = builder.tileDataFilePath;
      this.tileDataSetPath = builder.tileDataSetPath;
   }

   /**
    * Returns the {@link FileNames#tileDataSetPath}.
    */
   public String getTileDataSetPath()
   {
      return tileDataSetPath;
   }

   /**
    * Returns the {@link FileNames#lookupFilePath}.
    */
   public String getLookupFilePath()
   {
      return lookupFilePath;
   }

   /**
    * Returns the {@link FileNames#tileDataFilePath}.
    */
   public String getTileDataFilePath()
   {
      return tileDataFilePath;
   }

   /**
    * Returns the expected {@link Tile} name.
    *
    * @param level  the level of the {@link Tile}.
    * @param column the column of the {@link Tile}.
    * @param row    the row of the {@link Tile}.
    */
   public String getTileName(int level, int column, int row)
   {
      return getTileDataSetPath() +
             File.separator +
             level +
             File.separator +
             level +
             "_" +
             column +
             "_" +
             row +
             ".jpg";
   }

   /**
    * Provides the Builder pattern for the object initialization.
    */
   public static class Builder
   {
      /** The tile dataset full directory path */
      private final String tileDataSetPath;
      /** The LookupFile path */
      private final String lookupFilePath;
      /** The TileData path */
      private final String tileDataFilePath;

      /**
       * Implements the Builder Pattern for the object initialization.
       *
       * @param tileDataSetPath  the tile dataset's path.
       * @param lookupFilePath   the lookupFile's path.
       * @param tileDataFilePath the tileDataFile path.
       */
      public Builder(String tileDataSetPath, String lookupFilePath,
                 String tileDataFilePath)
```

```
        {
            this.tileDataSetPath = tileDataSetPath;
            this.tileDataFilePath = tileDataFilePath;
            this.lookupFilePath = lookupFilePath;
        }

        /**
         * Initializes the object.
         */
        public FileNames build()
        {
            return new FileNames(this);
        }
    }
}
```

I.2. LevelFiles class

```
package unb.mkotsollaris.tilemanagement;

import java.io.File;
import java.io.IOException;

/**
 * Contains the Lookup file and the TileData file and provides the necessary
 * methods to store and retrieve the tiles into the files.
 *
 * @author mkotsollaris
 * @since 1.0
 */
final class LevelFiles
{
    /** the number of the bytes that we use for the position pointer */
    final static int positionAllocationBytes = 8;
    /** the number of the bytes that we use for the size pointer */
    final static int sizeAllocationBytes = 4;
    /** the Lookup File */
    private final LookupFile lookupFile;
    /** the TileData File */
    private final TileDataFile tileDataFile;
    /** the level of the according tile dataset */
    private final int level;
    /** the path of the tile dataset in the particular level */
    private final String tileDataSetLevelPath;

    /**
     * Provides the Builder pattern for the object initialization.
     */
    public static class Builder
    {
```

```
    /** the level of the according tile dataset */
    private final int level;
    /** the file names information */
    private final FileNames fileNames;

    /**
     * Implements the Builder Pattern for the object initialization.
     *
     * @param fileNames the {@link FileNames}.
     * @param level     the {@link Tile} level.
     */
    Builder(FileNames fileNames, int level)
    {
       this.level = level;
       this.fileNames = fileNames;
    }

    /**
     * Initializes the object.
     */
    LevelFiles build() throws IOException
    {
       return new LevelFiles(this);
    }
}

//re-insures non instantiability
LevelFiles()
{
    throw new AssertionError();
}

/**
 * Private constructor.
 */
private LevelFiles(Builder builder) throws IOException
{
    boolean generate = false;
    level = builder.level;
    /* the file names information */
    FileNames fileNames = builder.fileNames;
    String lookupFileName = fileNames.getLookupFilePath() + level;
    String tileDataFileName = fileNames.getTileDataFilePath() + level;
    if(!(FileUtilities.exists(lookupFileName) &&
          FileUtilities.exists(tileDataFileName))) generate = true;
    tileDataSetLevelPath = builder.fileNames.getTileDataSetPath() +
          File.separator +
          level;
    lookupFile = new LookupFile.Builder(lookupFileName, level).build();
    tileDataFile =
          new TileDataFile.Builder(tileDataFileName, level).build();
    if(generate)
    {
       System.out.println(
             "Generating Content for the level: " + level + " ...");
       generateContent();
```

```
        }
/*        else System.out.println(
                "LevelFiles already exists for the level: " + level + ".");*/
    }

    /**
     * Generates the LevelFiles (LookupFile and TileDataFile).
     */
    private void generateContent() throws IOException
    {
        int columnNumber = Tile.computeColumnTotalNumber(level);
        long expectedTileNumber = Tile.computeExpectedTileNumber(level);
        long tileCounter = 0;
        for(int column = 0; column < columnNumber; column++)
        {
            for(int row = 0; row < columnNumber; row++)
            {
                String fileName = tileDataSetLevelPath + File.separator +
                        Tile.computeName(level, column, row);
                Tile tile = Tile.getInstance(fileName);
                if(tile.isValid())
                {
                    write(tile);
                }
                tileCounter++;
                Utilities.showStatus(tileCounter, expectedTileNumber, 50000);
            }
        }
    }


    /**
     * Updates the tileData and LookUpFile based on the given tile.
     *
     * As explained in page 135. Note that there will be 2^zoomLevel rows and
     * columns. The tile generation algorithm follows the WGS84 standards.
     *
     * File format: First byte: 32 bit (8 byte) for the first number that shows
     * which byte to read from the File Dataset. Second byte: 16 bit (4 byte)
     * for the second number that shows what's the size of the particular tile
     * image.
     *
     * @param tile: the wanted tile
     */
    private void write(Tile tile) throws IOException
    {
        long
                lookupFilePosition =
                lookupFile.getFilePosition(tile.getColumn(), tile.getRow());
        long
                tileDataFileLength =
                FileUtilities.getFileLength(tileDataFile.getFilePath());
        lookupFile.writeLong(tileDataFileLength, lookupFilePosition);
        lookupFile.writeInt(
                tile.getData().length,
                lookupFilePosition + positionAllocationBytes);
    tileDataFile.write(tile.getData(), tileDataFileLength);
}

/**
 * Returns the tile by parsing both lookup and tile data files.
 *
 * @param column the column of the {@link Tile}
 * @param row    the row of the {@link Tile}
 *
 * @return : The corresponding tile object
 */
public Tile getTile(int column, int row) throws IOException
{
    long lookupFilePosition = lookupFile.getFilePosition(column, row);
    long tileDataPosition = lookupFile.readLong(lookupFilePosition);
    int
            tileSize =
            lookupFile
                    .readInt(lookupFilePosition + positionAllocationBytes);
    byte[] tileData = tileDataFile.getTile(tileDataPosition, tileSize);
    return new Tile.Builder(tileData, level, column, row).build();
}

/**
 * Returns the expected position of the tile based on the following
 * equation:
 *
 * position = column * (2^level+row)*(12), where 12 the total number of the
 * allocation byte per record.
 *
 * @param level  the level of the {@link Tile}
 * @param column the column of the {@link Tile}
 * @param row    the row of the {@link Tile}
 */
long getPosition(int level, int column, int row)
{
    return ((column * (int) (Math.pow(2, level))) + row) *
            (positionAllocationBytes + sizeAllocationBytes);
}

/**
 * Returns the expected file length depending on the given level. The
 * returning number is given by the following equation: wanted_number =
 * (2^level)*2^level*12
 *
 * @param level the level of the {@link Tile}
 *
 * @return the expected file length
 */
static long getExpectedFileLength(int level)
{
    return (int) Math.pow(2, level) *
            (int) Math.pow(2, level) *
            12;
}
```

```
    /**
     * Generates the Level Files for a particular level.
     *
     * @param level the level of the LevelFiles that will be generated
     */
    static LevelFiles getInstance(FileNames fileNames, int level)
            throws IOException
    {
        return new LevelFiles.Builder(fileNames, level).build();
    }

    /**
     * Generates the level files for all the possible levels.
     *
     * @param fileNames the {@link FileNames}.
     */
    static LevelFiles[] getInstance(FileNames fileNames) throws IOException
    {
        int
                totalTileDataSetDirectoryNumber =
                FileUtilities.getDirectoriesNames(
                        fileNames.getTileDataSetPath()).length;
        LevelFiles[]
                levelFiles =
                new LevelFiles[totalTileDataSetDirectoryNumber + 1];
        for(int i = 0; i < totalTileDataSetDirectoryNumber; i++)
        {
            levelFiles[i] = new LevelFiles.Builder(fileNames, i).build();
        }
        return levelFiles;
    }


    @Override public String toString()
    {
        return "LeveFiles for the level " + level;
    }
}
```

I.3. LevelFilesSet class

```
package unb.mkotsollaris.tilemanagement;

import java.io.IOException;

/**
 * A set of LevelFiles.
 *
 * @author mkotsollaris
 * @since 1.0
 */
final public class LevelFilesSet
{
   /** The levelFiles per each Level */
   private final LevelFiles[] levelFiles;

   /**
    * Private constructor.
    */
   private LevelFilesSet(Builder builder)
   {
      levelFiles = builder.levelFiles;
   }

   private LevelFiles getLevelFile(int level)
   {
      return levelFiles[level];
   }

   // Suppresses default constructor, ensuring non-instantiability.
   private LevelFilesSet()
   {
      throw new AssertionError();
   }

   /**
    * Returns a {@link Tile} object.
    *
    * @param level  the level.
    * @param column the column of the tile.
    * @param row    the row of the tile.
    */
   public Tile getTile(int level, int column, int row) throws IOException
   {
      return levelFiles[level].getTile(column, row);
   }

   /**
    * Provides the Builder pattern for the object initialization.
    */
   public static class Builder
   {
      /** The levelFiles per each Level */
      private LevelFiles[] levelFiles;

      /**
       * Implements the Builder Pattern for the object initialization.
       *
       * @param fileNames the {@link FileNames}.
       */
      public Builder(FileNames fileNames) throws IOException
      {
         levelFiles = LevelFiles.getInstance(fileNames);
      }

      /**
```

```
         * Initializes the object.
         */
        public LevelFilesSet build()
        {
            return new LevelFilesSet(this);
        }
    }
}
```

I.4. LookupFile class

```
package unb.mkotsollaris.tilemanagement;

import java.io.IOException;

/**
 * The file containing the pointer to the TileDataFile and the size of the
 * tile.
 *
 * @author mkotsollaris
 */
final class LookupFile
{
    /** the level of the tiledata file */
    private int level;
    /** the filepath of where the LookupFile is being allocated */
    private final String filePath;

    /**
     * Creates an Empty lookup file or retrieves an existing one.
     */
    private LookupFile(Builder builder) throws IOException
    {
        level = builder.level;
        filePath = builder.filePath;
        if(!FileUtilities.exists(filePath))
        {
            FileUtilities.createFile(filePath,
                            LevelFiles.getExpectedFileLength(level));
        }
    }

    // Suppresses default constructor, ensuring non-insatiability.
    private LookupFile()
    {
        throw new AssertionError();
    }

    /**
     * Reads a long variable.
     *
     * @param position the byte of the file.
     */
    long readLong(long position) throws IOException
    {
        return FileUtilities.readLongFromFile(filePath, position);
    }

    /**
     * Provides the Builder pattern for the object initialization.
     */
    static class Builder
    {
        /** the level of the tile dataset */
        private final int level;
        /** the filepath of where the LookupFile is being allocated */
        private final String filePath;

        /**
         * Implements the Builder Pattern for the object initialization.
         *
         * @param filePath the file path
         * @param level    the level of of the {@link Tile}
         */
        Builder(String filePath, int level)
        {
            this.filePath = filePath;
            this.level = level;
        }

        /**
         * Initializes the object.
         */
        LookupFile build() throws IOException
        {
            return new LookupFile(this);
        }
    }

    /**
     * Reads an Integer from the file.
     *
     * @param position the byte of the file.
     */
    int readInt(long position) throws IOException
    {
        return FileUtilities.readIntFromFile(filePath, position);
    }

    /** gets the file name */
    String getFilePath()
    {
        return filePath;
    }

    /**
```

```
     * Calculates the position of the lookup file based on the level, row and
     * column of the tile.
     *
     * @param row:    the row of the wanted tile
     * @param column: the column of the wanted tile
     *
     * @return : the position (in bytes) that the requested tile should be found
     * in the file
     */
    long getFilePosition(int column, int row)
    {
        if(Math.pow(2, level) < column || Math.pow(2, level) < row)
            throw new IllegalArgumentException(
                    "level: " + level + ", column: " + column + ", row: " +
                            row);
        return ((column * (int) (Math.pow(2, level))) + row) *
                (LevelFiles.positionAllocationBytes +
                        LevelFiles.sizeAllocationBytes);
    }

    @Override public String toString()
    {
        return "LookupFile for the level: " + level + " with the filepath:" +
                filePath;
    }

    /**
     * Writes a long variable to the file.
     *
     * @param tileDataFileLength the length of the {@link TileDataFile}
     * @param lookupFilePosition the position (byte) of the {@link LookupFile}
     */
    void writeLong(long tileDataFileLength, long lookupFilePosition)
            throws IOException
    {
        FileUtilities.writeLongToFile(filePath, tileDataFileLength,
                                      lookupFilePosition);
    }

    /**
     * Writes a long variable to the file.
     *
     * @param tileLength the length of the {@link Tile}
     * @param position   the position (byte) of the file
     */
    void writeInt(int tileLength, long position) throws IOException
    {
        FileUtilities.writeIntToFile(filePath, tileLength, position);
    }
}
```

I.5. TileDataFile class

```
package unb.mkotsollaris.tilemanagement;

import java.io.IOException;

/**
 * Class Explanation: Represents a tileDataFile where the tiles' bytes are being
 * saved.
 *
 * @author mkotsollaris
 * @since 1.0
 */
final class TileDataFile
{
    /** the level of the tiledata file */
    private int level;
    /** the filename */
    private final String filePath;

    // Suppresses default constructor, ensuring non-instantiability.
    private TileDataFile()
    {
        throw new AssertionError();
    }

    /**
     * Creates an empty Tile Data File or Retrieves an existing one. The new
     * object is created only if the filename does not exist.
     *
     * @param builder the builder object
     */
    private TileDataFile(Builder builder) throws IOException
    {
        level = builder.level;
        filePath = builder.filePath;
        //delete any previous existing file with the same name
        if(!FileUtilities.exists(filePath))
        {
            FileUtilities.deleteFile(filePath);
            FileUtilities.createFile(filePath);
        }
    }

    /**
     * Returns the {@link Tile} from a given position within the file.
     *
     * @param tileDataPos the {@link TileDataFile} position
     * @param tileSize    the {@link Tile} size
     */
    byte[] getTile(long tileDataPos, int tileSize) throws IOException
    {
        return FileUtilities.readFromFile(filePath, tileDataPos, tileSize);
    }

    /**
     * Provides the Builder pattern for the object initialization.
     *
     * @author mkotsollaris
     * @since 1.0
     */
    static class Builder
    {
        /** the level of the tiledata file */
        private final int level;
        /** the filename */
        private final String filePath;

        Builder(String filePath, int level)
        {
            this.filePath = filePath;
            this.level = level;
        }

        /**
         * Initializes the object.
         */
        TileDataFile build() throws IOException
        {
            return new TileDataFile(this);
        }
    }

    /**
     * Returns the filepath
     */
    String getFilePath()
    {
        return filePath;
    }

    /**
     * Writes the data of the tile.
     *
     * @param tileData the data of the tile
     * @param position the position that the data will be written
     */
    void write(byte[] tileData, long position) throws IOException
    {
        FileUtilities.writeToFile(filePath, tileData, position);
    }

    @Override public String toString()
    {
        return "LookupFile for the level: " + level + " with the filepath:" +
                filePath;
    }
}
```

I.6. Tests Class

```
package unb.mkotsollaris.tilemanagement;

import java.io.IOException;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

/**
 * Created by Menelaos Kotsollaris on 2017-01-22.
 *
 * Class Explanation: Includes the performance tests.
 *
 * PseudoCode for the retrieval algorithm:
 * <pre>
 * Loop: 10 times //10 different parts of the world randomly selected tiles
 * foreach zoom level.
 *
 * Measure Time: LevelFilesSet
 * Measure Time: ImageBlock
 * Measure Time: SimpleFormat
 * End Loop
 * Print Average Time for all
 * </pre>
 */
public class PerformanceTests
{
    private final static String
            tileDataSetPath =
            MyFileNames.TileDatasetPath.getFileName();
    private final static String
            lookupFilePath =
            MyFileNames.LookupFilePath.getFileName();
    private final static String
            tileDataFilePath =
            MyFileNames.TileDataFilePath.getFileName();
    private final static String
            imageBlockFilePath =
            MyFileNames.ImageBlockPath.getFileName();
    private final static FileNames
            fileNames =
            new FileNames.Builder(tileDataSetPath, lookupFilePath,
                    tileDataFilePath).build();

    static void compareRetrieval(int level) throws IOException
    {
        float levelFileSum = 0, simpleFormatSum = 0, imageBlockSum = 0;
        float levelFileMin = Float.MAX_VALUE, simpleFormatMin = Float.MAX_VALUE,
                imageBlockMin = Float.MAX_VALUE;
    int levelFileThreshold = -1, simpleFormatThreshold = -1,
            imageBlockThreshold = -1;
    int iterationTimes = 10;
    //threshold is the number of the requesting tiles
    for(int threshold = 1; threshold <= iterationTimes; threshold++)
    {
        int iterationsTimes = 10;
        float[] levelFilesSetTimer = new float[iterationsTimes];
        float[] simpleFormatTimer = new float[iterationsTimes];
        float[] imageBlockTimer = new float[iterationsTimes];
        for(int j = 0; j < iterationsTimes; j++)
        {
            int[] columnNumbers = generateRandomNumbers(threshold, level);
            int[] rowNumbers = generateRandomNumbers(threshold, level);

            simpleFormatTimer[j] =
                    measureSimpleFormatRetrieval(level, rowNumbers,
                                    columnNumbers);
            levelFilesSetTimer[j] =
                    measureLevelFilesSetRetrieval(level, rowNumbers,
                                    columnNumbers);
            imageBlockTimer[j] =
                    measureImageBlockRetrieval(level, rowNumbers,
                                    columnNumbers);
            System.out
                    .printf("Threshold %d levelFilesSetTimer: %f, imageBlockTimer: %f, simpleFormatTimer:
%f\n",
                            threshold, levelFilesSetTimer[j],
                            imageBlockTimer[j], simpleFormatTimer[j]);
            simpleFormatSum += simpleFormatTimer[j];
            levelFileSum += levelFilesSetTimer[j];
            imageBlockSum += imageBlockTimer[j];
            if(isMin((simpleFormatTimer[j] / threshold), simpleFormatMin))
            {
                simpleFormatMin = simpleFormatTimer[j];
                simpleFormatThreshold = threshold;
            }
            if(isMin((levelFilesSetTimer[j] / threshold), levelFileMin))
            {
                levelFileMin = levelFilesSetTimer[j];
                levelFileThreshold = threshold;
            }
            if(isMin((imageBlockTimer[j] / threshold), imageBlockMin))
            {
                imageBlockMin = imageBlockTimer[j];
                imageBlockThreshold = threshold;
            }
        }
    }
    float levelFileAvg = levelFileSum / iterationTimes;
    float simpleFormatAvg = simpleFormatSum / iterationTimes;
    float imgBlockAvg = imageBlockSum / iterationTimes;
    System.out.println(
            "\nLevel: " + level + "\nAverage for levelFilesSetTimer: " +
                    levelFileAvg +
                    ", imageBlockTimer: " +
                imgBlockAvg +
                ", simpleFormatAvg: " +
                simpleFormatAvg + "");
    System.out.println("Min for levelFileThreshold: " +
                        levelFileThreshold +
                        ", imageBlockThreshold: " +
                        imageBlockThreshold +
                        ", simpleFormatThreshold: " +
                        simpleFormatThreshold + "\n");
}

private static boolean isMin(float value, float simpleFormatMin)
{
    return (value < simpleFormatMin);
}

private static long measureImageBlockRetrieval(int level, int[] rowNumbers,
                                        int[] columnNumbers)
        throws IOException
{
    long time1 = System.nanoTime();
    for(int columnNumber : columnNumbers)
    {
        for(int j = 0; j < rowNumbers.length; j++)
        {
            String
                filePath =
                fileNames.getTileName(level, columnNumber,
                                columnNumbers[j]);
            Tile
                tile =
                ImageBlock.getTile(
                    MyFileNames.ImageBlockPath.getFileName(), level,
                    columnNumber, columnNumbers[j]);
        }
    }
    long time2 = System.nanoTime();
    return TimeUnit.MILLISECONDS
            .convert(time2 - time1, TimeUnit.NANOSECONDS);
}

private static long getAverage(long[] timerLevelSimpleFormat)
{
    long sum = 0;
    for(int i = 0; i < timerLevelSimpleFormat.length; i++)
    {
        sum += timerLevelSimpleFormat[i];
    }
    return sum / timerLevelSimpleFormat.length;
}

private static long measureSimpleFormatRetrieval(int level,
                                        int[] rowNumbers,
                                        int[] columnNumbers)
        throws IOException
{
    long time1 = System.nanoTime();
    for(int columnNumber : columnNumbers)
    {
        for(int j = 0; j < rowNumbers.length; j++)
        {
            String
                    filePath =
                    fileNames.getTileName(level, columnNumber,
                                      columnNumbers[j]);
            Tile tile = Tile.getInstance(filePath);
        }
    }
    long time2 = System.nanoTime();
    return TimeUnit.MILLISECONDS
            .convert(time2 - time1, TimeUnit.NANOSECONDS);
}

private static long measureLevelFilesSetRetrieval(int level,
                                                  int[] columnNumbers,
                                                  int[] rowNumbers)
        throws IOException
{
    LevelFilesSet
            levelFilesSet =
            new LevelFilesSet.Builder(fileNames).build();
    long time1 = System.nanoTime();
    for(int i = 0; i < columnNumbers.length; i++)
    {
        for(int j = 0; j < rowNumbers.length; j++)
        {
            Tile
                    tile =
                    levelFilesSet.getTile(level, columnNumbers[i],
                                      rowNumbers[j]);
        }
    }
    long time2 = System.nanoTime();
    return TimeUnit.MILLISECONDS
            .convert(time2 - time1, TimeUnit.NANOSECONDS);
}

public static void testTime() throws IOException
{
    LevelFilesSet
            levelFilesSet =
            new LevelFilesSet.Builder(fileNames).build();
    long time1 = System.nanoTime();
    levelFilesSet.getTile(6, 0, 0);
    long time2 = System.nanoTime();
    System.out.println("LevelFilesSse TIME: " + (time2 - time1));
    String filePath = fileNames.getTileName(6, 1, 8);
    time1 = System.nanoTime();
    Tile tile = Tile.getInstance(filePath);
    time2 = System.nanoTime();
    System.out.println("SimpleFormat TIME: " + (time2 - time1));
}
```

```
    private static int[] generateRandomNumbers(int threshold, int level)
    {
        int max = Tile.computeColumnTotalNumber(level) - 1;
        int[] randomArray = new int[threshold];
        for(int i = 0; i < threshold; i++)
        {
            randomArray[i] = getRandomNumber(0, max);
        }
        return randomArray;
    }

    private enum TileMember
    {
        COLUMN,
        ROW
    }

    private static int getRandomNumber(int min, int max)
    {
        // nextInt is normally exclusive of the top value,
        // so add 1 to make it inclusive
        return ThreadLocalRandom.current().nextInt(min, max + 1);
    }
}
```

I.7. Tile class

```
package unb.mkotsollaris.tilemanagement;

import org.apache.commons.lang3.builder.EqualsBuilder;
import org.apache.commons.lang3.builder.HashCodeBuilder;

import java.io.IOException;
import java.util.Comparator;

/**
 * Represents a Tile Image (file) that holds the name, level, column, row, data,
 * and the file type.
 *
 * The tile name in the format X_Y_Z where X = Level, Y = Column, Z = Row.
 *
 * @author mkotsollaris
 * @since 1.0
 */
public final class Tile
{
    /** The Level of the tile (e.g. 0 for the 0_1_2 tile) */
    private final int level;
    /** The Column of the tile (e.g. 1 for the 0_1_2 tile) */
    private final int column;
```

```
/** The Row of the tile (e.g. 2 for the 0_1_2 tile) */
private final int row;
/** The data of the tile (bytes) */
private final byte[] data;
/** The File path */
private final String filePath;

/** @return the {@link Tile#level} */
public int getLevel()
{
   return level;
}

/** @return the {@link Tile#column} */
public int getColumn()
{
   return column;
}

/** @return the {@link Tile#row} */
public int getRow()
{
   return row;
}

/** @return the {@link Tile#data} */
public byte[] getData()
{
   return data;
}

/** @return the fileName */
public String computeFileName()
{
   return getLevel() + "_" + getColumn() + "_" + getRow() + ".jpg";
}

/**
 * Private constructor for internal initialization.
 */
private Tile(Builder builder)
{
   data = builder.data;
   level = builder.level;
   column = builder.column;
   row = builder.row;
   filePath = builder.filePath;
}


/**
 * Initializes a {@link Tile} object.
 *
 * @param filePath the file path of the particular tile
 *
 * @return the {@link Tile}
 * @throws IllegalArgumentException if the filepath is not valid
 */
public static Tile getInstance(String filePath)
        throws IllegalStateException, IOException
{
    try
    {
        String name = computeName(filePath);
        return new Builder(getData(filePath), computeLevel(name),
                        computeColumn(name), computeRow(name))
                .filePath(filePath).build();
    }
    catch(Exception e)
    {
        throw new IllegalArgumentException(
                "Not A valid tile path: " + filePath);
    }
}

/** @return the {@link Tile#filePath} */
public String getFilePath()
{
    return filePath;
}

/**
 * Computers the number of the expected tiles per zoom level. Assumes that
 * there are no missing tiles.
 *
 * @return long the number of the expected tiles
 */
static long computeExpectedTileNumber(int level)
{
    return (long) Math.pow(4, level);
}

/**
 * Provides the Builder pattern for the object initialization.
 *
 * @author mkotsollaris
 * @since 1.0
 */
static class Builder
{
    /** The Level of the tile (e.g. 0 for the 0_1_2 tile) */
    private final int level;
    /** The Column of the tile (e.g. 1 for the 0_1_2 tile) */
    private final int column;
    /** The Row of the tile (e.g. 2 for the 0_1_2 tile) */
    private final int row;
    /** The data of the tile (bytes) */
    private final byte[] data;
    /** The File path */
    private String filePath;

    /**
```

```
     * Implements the Builder Pattern for the object initialization.
     *
     * @param data   the data to be written in the file
     * @param level  the level of the {@link Tile}
     * @param column the column of the {@link Tile}
     * @param row    the row of the {@link Tile}
     */
    Builder(byte[] data, int level, int column, int row)
    {
        this.data = data;
        this.level = level;
        this.column = column;
        this.row = row;
    }

    /**
     * Implements the Builder Pattern for the object initialization.
     *
     * @param filePath the file path
     */
    Builder filePath(String filePath)
    {
        this.filePath = filePath;
        return this;
    }

    /**
     * Initializes the object.
     */
    Tile build()
    {
        return new Tile(this);
    }

}

/**
 * Returns the name of the file. For instance, given the path:
 * /Users/mkotsollaris/Desktop/tile_dataset/2/2_0_1.jpg will return
 * '2_0_1.jpg'.
 *
 * FIXME won't work for mac, linux etc. (path hardcoded)! Test
 *
 * @param filePath the filePath (e.g.: /Users/mkotsollaris/Desktop/tile_dataset/2/2_0_1.jpg)
 */
static String computeName(String filePath)
{
    String[]
            strArray =
            (System.getProperty("os.name")
                    .contains(MyFileNames.Windows.getFileName())) ?
                    filePath.split("\\\\") : filePath.split("/");

    return strArray[strArray.length - 1].split("\\.")[0];
}
```

```
static String computeName(int level, int column, int row)
{
   return level + "_" + column + "_" + row + ".jpg";
}

/**
 * Returns the byte[] array containing the image information.
 *
 * @param filePath the filePath (e.g.: /Users/mkotsollaris/Desktop/tile_dataset/2/2_0_1.jpg)
 */
private static byte[] getData(String filePath) throws IOException
{
   return FileUtilities.readFromFile(filePath);
}

/**
 * Returns if the file has the proper format.
 */
boolean isValid()
{
   return level != -1;
}

/**
 * Returns the level of the string.
 *
 * @param fileName: the tile name in the format X_Y_Z where X = Level, Y =
 *              Column Z = Row. Example Input: 3_5_1.jpg then the
 *              function will return 3.
 */
static int computeLevel(String fileName)
{
   try
   {
      return Integer.parseInt(fileName.split("_")[0]);
   }
   catch(Exception e)
   {
      return -1;
   }
}

/**
 * Returns the column of the string.
 *
 * @param fileName: Example Input: 3_5_1.jpg then the function will return
 *              5.
 */
static int computeColumn(String fileName)
{
   return Integer.parseInt(fileName.split("_")[1]);
}

/**
 * Returns the row of the string. For instance, if the input is
 * "/Users/mkotsollaris/Desktop/tile_dataset/2/2_0_1.jpg" the the output
 * will be 0.
 *
 * @param fileName: the tile filePath in the format X_Y_Z where X = Level, Y
 *                = Column Z = Row. Example Input: 3_5_1.jpg then the
 *                function will return 1.
 */
static int computeRow(String fileName)
{
   return Integer.parseInt((fileName.split("_")[2]));
}

/**
 * Returns the file type (e.g. if input "/Users/mkotsollaris/Desktop/tile_dataset/2/2_0_1.jpg"
 * then it returns "jpg".
 *
 * @param filePath the filename (e.g. "0_1_2.jpg")
 */
static String computeFileType(String filePath)
{
   return filePath.split("\\.")[1];
}

@Override public String toString()
{
   return "Tile level: " +
          level +
          ", column: " +
          column +
          ", row: " +
          row + ", data Length: " + data.length;
}

/**
 * Returns the total number of columns for a particular level. The same
 * number stands for the total number of rows as well.
 */
public static int computeColumnTotalNumber(int level)
{
   return (int) Math.pow(2, level);
}

private static Comparator<Tile> tileLengthComparator = (tile1, tile2) -> {
   if(tile1.getData().length == tile2.getData().length) return 0;
   if(tile1.getData().length > tile2.getData().length) return 1;
   return -1;
};

static Comparator<Tile> getTileLengthComparator()
{
   return tileLengthComparator;
}

/**
 * A tile is equal to another tile if they both have the same level, column,
 * row and data (bytes).
 *
```

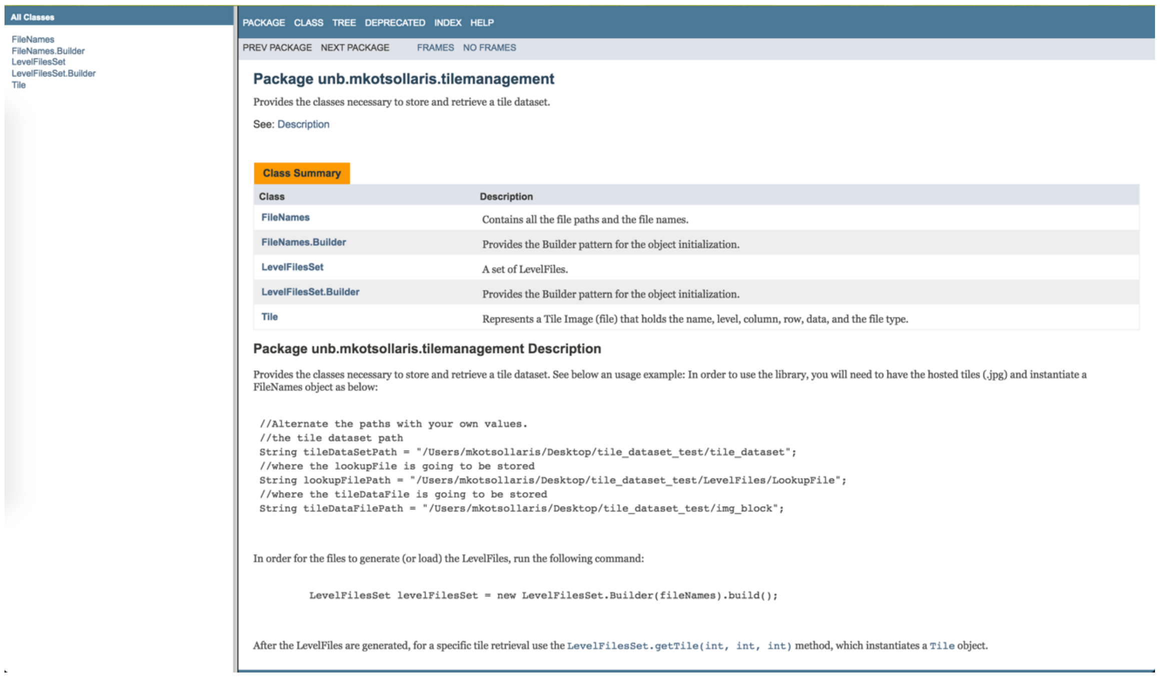
```
     */
    @Override public boolean equals(Object obj)
    {
        if(obj == null) return false;
        if(!(obj instanceof Tile)) return false;
        if(obj == this) return true;

        Tile otherTile = (Tile) obj;
        return new EqualsBuilder().
                append(level, otherTile.level).
                append(column, otherTile.column).
                append(row, otherTile.row).
                append(data, otherTile.data).
                isEquals();
    }


    @Override public int hashCode()
    {
        return new HashCodeBuilder(17, 31). // two randomly chosen prime numbers
                append(level).
                append(column).
                append(row).
                append(row).
                append(data).
                toHashCode();
    }
}
```

Appendix II. LevelFilesSet's Javadoc

The Javadoc of the LevelFilesSet can be seen in Figure 0.1.


All Classes
FileNames
FileNames.Builder
LevelFilesSet
LevelFilesSet.Builder
Tile
PACKAGE CLASS TREE DEPRECATED INDEX HELP
PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Package unb.mkotsollaris.tilemanagement

Provides the classes necessary to store and retrieve a tile dataset.

See: Description

Class Summary

Class	Description
FileNames	Contains all the file paths and the file names.
FileNames.Builder	Provides the Builder pattern for the object initialization.
LevelFilesSet	A set of LevelFiles.
LevelFilesSet.Builder	Provides the Builder pattern for the object initialization.
Tile	Represents a Tile Image (file) that holds the name, level, column, row, data, and the file type.

Package unb.mkotsollaris.tilemanagement Description

Provides the classes necessary to store and retrieve a tile dataset. See below an usage example: In order to use the library, you will need to have the hosted tiles (.jpg) and instantiate a FileNames object as below:

```
//Alternate the paths with your own values.
//the tile dataset path
String tileDataSetPath = "/Users/mkotsollaris/Desktop/tile_dataset_test/tile_dataset";
//where the lookupFile is going to be stored
String lookupFilePath = "/Users/mkotsollaris/Desktop/tile_dataset_test/LevelFiles/LookupFile";
//where the tileDataFile is going to be stored
String tileDataFilePath = "/Users/mkotsollaris/Desktop/tile_dataset_test/img_block";
```

In order for the files to generate (or load) the LevelFiles, run the following command:

```
        LevelFilesSet levelFilesSet = new LevelFilesSet.Builder(fileNames).build();
```

After the LevelFiles are generated, for a specific tile retrieval use the `LevelFilesSet.getTile(int, int, int)` method, which instantiates a `Tile` object.

Figure 0.1 LevelFilesSet's Javadoc

Appendix III. Java Servlet & LevelFilesSet Integration

```
package com.mkotsollaris.projects;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

@WebServlet(name = "com.mkotsollaris.projects.MyServlet") public class Servlet
        extends HttpServlet
{
    private LevelFilesSet levelFilesSet;

    public void init()
    {
        String tileDataSetPath = MyFileNames.TileDatasetPath.getFileName();
        String lookupFilePath = MyFileNames.LookupFilePath.getFileName();
    String tileDataFilePath = MyFileNames.TileDataFilePath.getFileName();
    String
        imageBlockFilePath =
        unb.mkotsollaris.tilemanagement.MyFileNames.ImageBlockPath
            .getFileName();
    FileNames
        fileNames =
        new FileNames.Builder(tileDataSetPath, lookupFilePath,
                      tileDataFilePath, imageBlockFilePath)
            .build();
    try
    {
      levelFilesSet = new LevelFilesSet.Builder(fileNames).build();
    }
    catch(IOException e)
    {
      e.printStackTrace();
    }

  }

  protected void doPost(HttpServletRequest request,
                  HttpServletResponse response)
      throws ServletException, IOException
  {

  }

  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException
  {
    try
    {
      String[] tileNames = request.getParameterValues("tile");
      Tile[] tiles = Utilities.retreiveTiles(tileNames, levelFilesSet);
      outputTileToBrowser(tiles, response);
    }
    catch(Exception e)
    {
      e.printStackTrace();
    }
  }

  public void destroy()
  {
    // Finalization code...
  }

  /**
   * Checks if the file exists and if it does then it prints it to the user's
   * browser, otherwise in throws an error.
   */
  public void outputTileToBrowser(Tile[] tiles, HttpServletResponse response)
      throws Exception
  {
    response.setContentType("text/html");
```

```
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        for(Tile tile : tiles)
        {
            outputStream.write(tile.getData());
        }
        byte c[] = outputStream.toByteArray();
        response.getOutputStream().write(c, 0, c.length);
    }
}
```

Appendix IV. Database Helper Class with JDBC Implementation for PostgreSQL connection

```
/**
 * Created by Menelaos Kotsollaris on 2016-11-24.
 *
 * Class Explanation: Contains the PostgreSQL server connection.
 */
class PostgresHelper
{

    private Connection conn;
    private String host;
    private String dbName;
    private String user;
    private String pass;

    protected PostgresHelper()
    {
    }

    PostgresHelper(String host, String dbName, String user, String pass)
    {
        this.host = host;
        this.dbName = dbName;
        this.user = user;
        this.pass = pass;
    }

    boolean connect() throws SQLException, ClassNotFoundException
    {
        if(host.isEmpty() || dbName.isEmpty() || user.isEmpty() ||
                pass.isEmpty())
        {
            throw new SQLException("Database credentials missing");
        }
```

```
   Class.forName("org.postgresql.Driver");
   this.conn =
          DriverManager.getConnection(this.host + this.dbName, this.user,
                                 this.pass);
   return true;
}

ResultSet execQuery(String query) throws SQLException
{
   return this.conn.createStatement().executeQuery(query);
}

/**
 * Adds a Tile to PostgreSQL.
 * */
int addTileQuery(Tile tile) throws SQLException, FileNotFoundException
{
   if(!tile.isValid()) throw new IllegalArgumentException("Illegal Tile");
   String tableName = "level" + tile.getLevel();
   long
          id =
          LevelFiles.getPosition(tile.getLevel(), tile.computeColumn(),
                              tile.computeRow());
   //String insert = "INSERT INTO "+tableName;
   String insert = "INSERT INTO " + tableName +
          "(" + COLUMNS.TILE_ID + "," + COLUMNS.TILE_DATA + "," +
          COLUMNS.TILE_LEVEL + "," + COLUMNS.TILE_COLUMN +
          "," + COLUMNS.TILE_ROW +
          "," + COLUMNS.TILE_IMAGE_FORMAT + "," + COLUMNS.TILE_SOURCE +
          ")";
   String values = id + ",?" + "," +
          tile.getLevel() + "," + tile.computeColumn() + "," +
          tile.computeRow() + ",\'" + tile.getFileType().toUpperCase() +
          "\'," + "\'" +
          tile.getSource() + "\');";
   String query = insert + " VALUES (" + values;
   PreparedStatement statement = conn.prepareStatement(query);
   statement.setBytes(1, tile.getData());
   int result = statement.executeUpdate();
   statement.close();
   return result;
}

/**
 * Loads the dataset to the database.
 * */
static void loadDataSet(PostgresHelper client, int level)
       throws IOException, SQLException
{
   File[]
          files =
          FileUtilities.getFiles(
                 "/Users/mkotsollaris/Desktop/tile_dataset/" + level);
   for(int j = 0; j < files.length; j++)
   {
      Tile tile = Tile.getInstance(files[j].getAbsolutePath());
```

```
        if(tile.isValid()) client.addTileQuery(tile);
    }
}

/**
 * Drops and creates a brand new table.
 */
public void resetDatabase(int level) throws SQLException
{
    String tableName = "level" + level;
    String query = "DROP TABLE " + tableName;
    //System.out.println(query);
    this.conn.createStatement().executeUpdate(query);
    query = "CREATE TABLE " + tableName + " (" +
            "    tile_id           BIGINT," +
            "    tile_data         BYTEA," +
            "    tile_level        SMALLINT," +
            "    tile_row          INTEGER," +
            "    tile_column       INTEGER," +
            "    tile_image_format image_format," +
            "    tile_source       CHARACTER VARYING(30)," +
            "    PRIMARY KEY (tile_id));";
    this.conn.createStatement().executeUpdate(query);
}

/**
 * Returns a Tile.
 * */
public Tile getTile(int level, int column, int row)
        throws SQLException, IOException
{
    long position = LevelFiles.getPosition(level, column, row);
    String tableName = "level" + level;
    PreparedStatement
            preparedStatement =
            conn.prepareStatement("SELECT * FROM " + tableName + " WHERE " +
                                  COLUMNS.TILE_ID.column_name +
                                  " = " +
                                  position);
    ResultSet resultSet = preparedStatement.executeQuery();
    boolean hasNext = resultSet.next();
    if(!hasNext) return Tile.getInvalidTile();
    return new Tile.Builder(resultSet.getBytes(COLUMNS.TILE_DATA.position),
                        level, column, row).build();
}

/**
 * The columns of the database.
 */
private enum COLUMNS
{
    TILE_ID("tile_id", 1),
    TILE_DATA("tile_data", 2),
    TILE_LEVEL("tile_level", 3),
    TILE_ROW("tile_row", 4),
    TILE_COLUMN("tile_column", 5),
        TILE_IMAGE_FORMAT("tile_image_format", 6),
        TILE_SOURCE("tile_image_format", 7);

        private String column_name;
        /**
         * The position starts from the number 1.
         */
        private int position;

        COLUMNS(String column_name, int position)
        {
            this.column_name = column_name;
            this.position = position;
        }
    }

    /**
     * Returns a tile (similar to getTile).
     *
     * */
    public static Tile printTile(String tileName) throws Exception
    {
        PostgresHelper
                client =
                new PostgresHelper(DbContract.HOST, DbContract.DB_NAME,
                        DbContract.USERNAME, DbContract.PASSWORD);
        try
        {
            if(client.connect())
            {
                int level = Tile.computeLevel(tileName);
                int column = Tile.computeColumn(tileName);
                int row = Tile.computeRow(tileName);
                return client.getTile(level, column, row);
            }
        }
        catch(Exception e)
        {

        }
        throw new Exception();
    }
}
```

Appendix V. Google Cloud Benchmark Tables

Table 0.1 First day benchmark results, 10 May, 10AM at UTC-3h

Level	SimpleFormat (ms)	ImageBlock (ms)	LevelFilesSet (ms)
0	52	36	92
1	97	90	127
2	185	155	137
3	195	168	141
4	201	173	156
5	203	184	160
6	231	198	173
7	239	201	181
8	244	204	185
9	247	205	187
10	263	213	191

Table 0.2 Benchmark ran on May 10th at 10 PM at UTC-3h

Level	SimpleFormat (ms)	ImageBlock (ms)	LevelFilesSet (ms)
0	33	22	89
1	56	50	92

2	131	102	123
3	155	147	134
4	189	153	148
5	199	173	158
6	224	191	162
7	238	199	177
8	242	205	192
9	252	208	197
10	267	226	202

Table 0.3 Benchmark ran on May 11th at 10 AM at UTC-3h

Level	SimpleFormat (ms)	ImageBlock (ms)	LevelFilesSet (ms)
0	31	30	77
1	69	68	83
2	155	172	105
3	179	180	116
4	220	190	125
5	239	198	139
6	284	205	183
7	285	220	198

8	297	230	205
9	303	230	215
10	306	247	222

Table 0.4 Benchmark ran on May 11th at 10 PM at UTC-3h

Level	SimpleFormat (ms)	ImageBlock (ms)	LevelFilesSet (ms)
0	31	19	78
1	38	27	82
2	67	41	93
3	104	74	98
4	115	111	105
5	164	127	121
6	213	157	149
7	247	183	161
8	302	213	190
9	360	252	215
10	373	299	282

Table 0.5 Benchmark ran on May 12 at 10 AM at UTC-3h

Level	SimpleFormat (ms)	ImageBlock (ms)	LevelFilesSet (ms)
0	30	13	84
1	99	93	101
2	148	139	109
3	153	148	137
4	176	161	155
5	189	170	168
6	245	237	212
7	287	254	233
8	315	281	267
9	355	301	295
10	393	329	301

Table 0.6 Benchmark ran on May 12 at 10 PM at UTC-3h

Level	SimpleFormat (ms)	ImageBlock (ms)	LevelFilesSet (ms)
0	25	16	61
1	88	62	87
2	153	101	127

3	178	155	149
4	197	192	167
5	201	199	179
6	214	208	185
7	223	214	194
8	228	221	197
9	234	226	206
10	266	247	224

Curriculum Vitae

Menelaos Kotsollaris

2016, Diploma in University Teaching, University of New Brunswick, Canada

2015 – 2017, MScE. University of New Brunswick, NB, Canada

2015, Hons. B.Sc. University of Piraeus, Athens, Greece

Conference Presentations:

1. **Kotsollaris, M**., Liu, W., Stefanakis, E. & Zhang, Y. "Conceptual Design of a Scalable Web Tile Management Framework". The 2017 Graduate Research Conference (GRC), UNB, March 18 2017, Fredericton, Canada.

Conference Posters:

1. **Kotsollaris, M**., Liu, W., Stefanakis, E. & Zhang, Y. "Improving the efficiency of the Web Image Management Systems". The 2017 Graduate Research conference (GRC), UNB, March 18 2017, Fredericton, Canada.
2. **Kotsollaris, M**., Liu, W., Stefanakis, E. & Zhang, Y. "Implementing a Scalable Web Image Management System". 14th Annual Research Expo; Faculty of Computer Science, UNB, April 7 2017, Fredericton, Canada

Journal Papers:

1. **Kotsollaris, M**., Liu, W., Stefanakis, E. & Zhang, Y. "LevelFilesSet: An efficient Data Structure for Web Tiled Map Management Systems" – *(under review)*