Pinky: Interactively Analyzing Large EEG Datasets

by

Joshua Blum

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Computer Science and Engineering
at the Massachusetts Institute of Technology

February 2016

© Massachusetts Institute of Technology 2016. All rights reserved.
Pinky: Interactively Analyzing Large EEG Datasets

by

Joshua Blum

Submitted to the Department of Electrical Engineering and Computer Science
on December 18, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

In this thesis, I describe a system I designed and implemented for interactively analyzing large electroencephalogram (EEG) datasets. Trained experts, known as encephalographers, analyze EEG data to determine if a patient has experienced an epileptic seizure. Since EEG analysis is time intensive for large datasets, there is a growing corpus of unanalyzed EEG data. Fast analysis is essential for building a set of example data of EEG results, allowing doctors to quickly classify the behavior of future EEG scans. My system aims to reduce the cost of analysis by providing near real-time interaction with the datasets. The system has three optimized layers handling the storage, computation, and visualization of the data. I evaluate the design choices for each layer and compare three different implementations across different workloads.
Acknowledgments

This work is dedicated to Herbert Blum.

First, I would like to thank my family for their enduring support and love throughout my time at MIT.

I would like to thank Professor Sam Madden, Dr. Brandon Westover and Professor Mark Silberstein for their guidance and support while advising me throughout this project. Their insights and suggestions greatly helped shape this work.

I would also like to thank Amir Watad, Sagi Shahar, and Feras Daoud for helping me have a home away from home while collaborating at the Technion. At MIT, my work would never have been completed if not for the great friendship and support of Tal Tchwella, Stephanie Wang, Max Kanter and Neha Patki. I would also like to thank Adam Marcus, Lydia Gu, and Eugene Wu for our initial discussions of research topics and continuing support throughout the project.

In addition, I would like to acknowledge collaboration with Stavros Papadopoulos on the TileDB project, Stephanie Wang with the Visgoth system, Siddharth Biswal and his help with algorithms for processing EEGs, Bastian Bechtold and his WebGL-Spectrogram implementation, and Ole Christian Eidheim for support with the websocket server.
## Contents

### 1 Introduction 15

1.1 Pinky ........................................... 15
1.2 Overview of EEG Analysis .................. 16
1.3 System Architecture ......................... 18
   1.3.1 Storage Layer ......................... 19
   1.3.2 Compute Layer ......................... 19
   1.3.3 Visualization Layer .................. 20
   1.3.4 Visgoth System ....................... 20
1.4 Usage .......................................... 20
1.5 Contributions ................................ 21

### 2 Storage Layer 23

2.1 Design ......................................... 23
   2.1.1 Ingestion Workload ................... 24
   2.1.2 Precomputation Workload .............. 24
   2.1.3 Visualization Serving Workload ....... 25
2.2 Implementation ............................... 25
   2.2.1 StorageBackend API .................. 25
   2.2.2 EDFBackend ............................. 27
   2.2.3 BinaryBackend ......................... 27
   2.2.4 HDF5Backend ......................... 28
   2.2.5 TileDBBackend ......................... 29
   2.2.6 Command Line Programs ............... 30
2.2.7 Optimizations ........................................... 30

2.3 Evaluation .................................................. 30

2.3.1 Ingestion Experiment ................................. 31

2.3.2 Precompute Experiment .............................. 33

2.3.3 Storage Overhead .................................. 35

3 Compute Layer ................................................. 37

3.1 Design ......................................................... 37

3.1.1 Websocket Server .................................. 37

3.1.2 Webapp Server ....................................... 38

3.1.3 File Ingestion Daemon .............................. 38

3.1.4 Spectrogram Calculation ......................... 38

3.2 Implementation ............................................ 39

3.2.1 Websocket Server .................................. 39

3.2.2 Webapp Server ....................................... 40

3.2.3 File Ingestion Daemon .............................. 41

3.2.4 Spectrogram Calculation ......................... 41

3.2.5 Command Line Programs ......................... 43

3.2.6 Optimizations ....................................... 43

4 Visualization Layer ........................................... 45

4.1 Design ......................................................... 45

4.1.1 Interface ............................................... 45

4.1.2 Communication .................................... 46

4.1.3 Rendering ............................................. 46

4.2 Implementation ............................................ 46

4.2.1 Interface ............................................... 46

4.2.2 Communication .................................... 49

4.2.3 Rendering ............................................. 50

4.2.4 Optimizations ....................................... 50
List of Figures

1-1 Etymology of Pinky’s name. ............................................. 16
1-2 EEG electrode placement on a patient’s scalp. ...................... 17
1-3 Spectrogram for one hour window of EEG data of the LL region of the brain. 18
1-4 Pinky system architecture. ............................................. 19

2-1 On disk data layout for the BinaryBackend implementation. ....... 28
2-2 Total runtime for different backend implementations in the ingestion experiment. ............................................. 32
2-3 Total read time for different backend implementations in the precompute experiment. ............................................. 34
2-4 Total write time for different backend implementations in the precompute experiment. ............................................. 35
2-5 Total runtime for different backend implementations in the precompute experiment. ............................................. 36

4-1 Screenshot of Pinky’s user interface. An analyst can view the four spectrograms corresponding to different regions of the brain and query for different time ranges to view. ............................................. 47
4-2 Screenshot of the settings modal. The options allow an analyst to tune the visualization coloring and change the default time window query size. .... 48
4-3 Screenshot of a zoomed spectrogram view of a single rendered region with dynamic axis labels. ............................................. 48
4-4 Screenshot of loading interface. When a new query is issued, the old data is blurred to avoid confusion of current and past results during query execution. 49
4-5 Screenshot of invalid mrn error message when a bad query is issued. . . . . 50

5-1 Sample request in the Visgoth system. . . . . . . . . . . . . . . . . . . . . 56
5-2 Sample response in the Visgoth system. . . . . . . . . . . . . . . . . . . . 56
5-3 Overall latency of an application client’s interaction, plotted against down-

sample, the factor by which the application server reduced data served back
to the client. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
5-4 Overall latency of an application client’s interaction, plotted against ren-
dering throughput, the number of spectrogram frames that can be rendered
per second. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
5-5 Overall latency of an application client’s interaction, plotted against ap-
proximate network bandwidth. Each curve represents the latency values
predicted by one regression model. The points plotted are from the test
dataset. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
List of Tables

2.1 Storage overhead costs for each StorageBackend type. ............ 36
Chapter 1

Introduction

A number of applications require a domain expert to visually inspect and process a stream of incoming data. The problem with manual inspection is the inability to scale as datasets grow exponentially [19]. As the dataset grows, it becomes difficult to visualize interactively [39]. In this thesis we focus on medical data, where doctors have to analyze a patient’s data and extract relevant information for treatment. Specifically, we focus on electroencephalogram (EEG) readings, a test which used to detect abnormalities related to the electrical activity of the brain.

Today, doctors store large amounts of patient data that they cannot analyze because they lack tools to efficiently view datasets at scale. To address this issue, we have designed and implemented Pinky, a system for processing large amounts of EEG data, allowing near real-time interactive analysis.

1.1 Pinky

Pinky is a doctor’s newest tool for analyzing the brain, see Figure [1-1]. Working with a team of researchers at Massachusetts General Hospital (MGH), we have designed and implemented the system to handle the fast growing corpus of collected EEG data. This end-to-end system handles the storage, processing, and visualization of EEG data. The goal of the system is to provide a scalable architecture for concurrent analysis of patient
records with near real-time interactivity. Each layer of the system is optimized for use and evaluated across hundreds of gigabytes of patient data.

![Figure 1-1: Etymology of Pinky’s name.](image)

### 1.2 Overview of EEG Analysis

A seizure is a transient aberration in the brain’s electrical activity. People with the central nervous system disorder epilepsy suffer from recurrent seizures, often happening suddenly and at unpredictable times. A seizure can vary from a lapse of attention to a whole-body convolution. Frequent seizures are dangerous, as they can increase risk of sustaining physical injuries and can even result in death [29].

One method for detecting the onset of epileptic seizures is the analysis of scalp EEG data, a non-invasive measure of the brain’s electrical activity. Continuous EEG (cEEG) data is typically recorded using 19 silver/silver chloride electrodes, affixed to the scalp [36]. Figure [1-2] shows a drawing of the placement of sensors on a patient’s scalp.
Trained individuals, such as attending physicians, epilepsy/neurophysiology fellows, or registered EEG technicians (encephalographers), review and screen EEG recordings, which typically take place over a continuous 24-hour period [10]. Unlike traditional epilepsy monitoring units which focus on provoking and capturing seizures, the goal of cEEG studies is to efficiently identify future seizures and prevent them. This leads to an increase in the number of cEEG recordings for preventative measures. Intensive care unit centers are subsequently overwhelmed with the analysis of the growing dataset due to the small number of available trained individuals. Methods to screen long EEG recordings without sacrificing accuracy are necessary to be able to efficiently process this data.

Typically, EEGs displays show no more than 10 to 15 seconds of data per screen of raw voltage readings and requires an analyst to simultaneously inspect multiple channels. In contrast, a compressed spectral array [9] or spectrogram display may show 2 to 8 hours of data on a single color map [10]. This allows analysts to quickly screen long periods of EEG data, determining which segments, if any, require direct review of the raw data. Spectrogram review reduces cEEG review time by 78% [26], with minimal loss of sensitivity compared with conventional review. For these reasons, we focus on building a tool to rapidly analyze spectrogram data.

Spectrograms are the most widely used compressed data format for EEG data [36]. A spectrogram consists of three-dimensional plots with time on the x-axis, frequency on the
Figure 1-3: Spectrogram for one hour window of EEG data of the LL region of the brain.

y-axis, and EEG power on the z-axis. Figure 1-3 shows an example of rendered spectrogram data. An analyst typically views four spectrograms concurrently, mapped to different regions of the brain. Each region is formed by using multiple EEG channels where an EEG channel is the difference between voltages measured at two electrodes. This captures the summed potential of millions of neurons [29]. Figure 1-2 shows the electrode placement on the patient’s scalp, yielding four regions for analysis: left lateral power, LL, (Fp1-F7, F7-T3, T3-T5, T5-O1), left parasagittal power, LP, (Fp1-F3, F3-C3, C3-P3, P3-O1), right lateral power, RL, (Fp2-F8, F8-T4, T4-T6, T6-O2), right parasagittal power, RP, (Fp1-F4, F3-C4, C4-P4, P4-O2).

Data from a single patient can vary in size from tens to hundreds of gigabytes and the number of EEG tests performed each year is estimated to be between 10 and 25 million [15]. As this corpus of data collected at the ICU continues to grow, efficient mechanisms to store and visualize this data at scale are key for analysts to quickly view patient screenings. Pinky aims to provide this for analysts by giving them a simple yet powerful interface to view spectrogram data.

1.3 System Architecture

Pinky is comprised of three coupled layers which handle storage, computation and visualization. Figure 1-4 shows the overall architecture of the system.
1.3.1 Storage Layer

The storage layer, discussed in detail in Chapter 2, is responsible for storing raw EEG patient data and the calculated spectrogram. This datastore must optimize both reads and writes of array based data for multidimensional arrays on the order of tens to hundreds of gigabytes.

1.3.2 Compute Layer

The compute layer, discussed in detail in Chapter 3, is an extensible module which handles the algorithms to calculate the spectrogram and other EEG related calculations. As we discuss in Section 6.4, there are a number of extensions the project can take, thus it is
important that an interested developer can easily add functionality to this layer. In addition, the compute layer contains two servers. One server interfaces with the optimized EEG algorithms and the storage layer to serve array based data. The second server is a lightweight server for the web resources of the visualization layer.

1.3.3 Visualization Layer

The visualization layer, discussed in detail in Chapter 4, is a browser based module that renders the data to the client. The interface allows users to query based on a patient's id (medical record number, mrn) and view a spectrogram for a given time interval. An analyst may smoothly pan and zoom throughout the dataset.

1.3.4 Visgoth System

Since enabling interactivity is an important design criteria, we have designed and built an optimization module for browser based visualizations named Visgoth. The system uses profiling information from the client and server to suggest an adaptive scaling of the visualizations served in order to keep latency consistent, regardless of a client’s hardware or network bandwidth. We discuss Visgoth in detail in Chapter 5.

1.4 Usage

The project code base is available publicly on Github [17] at https://github.com/joshblum/eeg-toolkit with documentation for installing the project for development. In addition, we have created Docker [25] images that can easily be installed for production use. Armed with a dataset, any curious doctor is able to install the images and load the data for analysis. The docker images are available for public use on DockerHub: https://hub.docker.com/r/joshblum/eeg-toolkit-webapp and https://hub.docker.com/r/joshblum/eeg-toolkit-toolkit The Github project contains specific installation instructions.
1.5 Contributions

Pinky makes the following contributions:

- Implements an abstraction for array based storage systems.
- Implements three different backends which adhere to the abstraction.
- Evaluates the different backends for varying input ranges and workloads.
- Implements optimized algorithms for analyzing EEG data.
- Provides an extensible framework for accessing array based data and visualizing it in the browser.
- Implements scalable in-browser visualizations using the client’s GPU.
- Implements a new system, Visgoth, for reducing latency for browser based visualizations.

These contributions enable doctors and medical expert analysts to interactively analyze EEG data at scale.
Chapter 2

Storage Layer

Pinky stores both raw EEG voltage readings and calculated spectrograms. The raw EEG data is simply a matrix with a column for each sensor (see Figure 1-2) and a data sample read by the sensor at each row. Similarly, we store the spectrogram, a range of frequencies across time, as a multidimensional array. Since matrices of floating point values represent both raw EEG data and the spectrogram visualization, the system requires an efficient way to store and query such large multidimensional arrays. The storage layer meets this requirement by providing an abstraction called the StorageBackend for storing array based data. The abstraction gives read and write access to 2-dimensional matrices of floating point values. Since the interactivity the system demands requires efficient querying of the data, we evaluate different array based storage systems and compare the trade-offs of each against a simple storage system that we implement.

2.1 Design

The StorageBackend abstraction provides a thin layer over existing array based storage systems to provide a consistent API for evaluation across different systems. The goal is to efficiently expose read and write access to large array based datasets. A patient EEG scan typically lasts 24 hours, but can run for up 7 days [10]. The raw output of voltage readings yield a file from tens to hundreds of gigabytes in size. We assume that the entire dataset cannot always fit into the memory of the server, given that it must concurrently respond
to multiple client queries. Thus, the StorageBackend maintains an efficient on disk representation of the data.

A StorageBackend implementation must primarily handle three different workloads. The first is data ingestion. This is a write heavy workload – reading patient data files from disk and storing them in the system. Secondly, spectrogram data is typically written once in a precomputation step and saved for later analysis. This workload is primarily read heavy, since the computed spectrogram is much smaller in size than the raw input file. After being stored, efficient reads are required for an analyst to access the visualizations.

### 2.1.1 Ingestion Workload

Once a patient scan is complete, the system ingests the raw data file for analysis. The raw data is initially stored in the European Data Format (EDF) [23], a simple binary format for storing multichannel biological and physical signals. During ingestion, we read the data in chunks of configurable size from an input EDF file and convert them for storage within the system. This workload requires the storage system to efficiently handle the addition of large datasets. We evaluate the performance when ingesting data in Section 2.3.1.

### 2.1.2 Precomputation Workload

After ingesting a file into the system, the spectrogram for a patient is computed by reading the raw values from the storage system. This precomputation step is important for minimizing the latency when serving the visualization. The precomputation calculation involves reading columns of the raw data to perform the spectrogram calculation and writing the spectrogram matrix back to the datastore. We describe the details of the calculation in Section 3.1.4. This workload requires efficient reading of individual columns in the raw data and similar to the ingestion workload, the ability to write dense matrices efficiently.
2.1.3 Visualization Serving Workload

In practice, the primary workload the storage system handles is serving chunks of a calculated visualization. Analysts will request data for a given patient and time interval and need to be able to quickly have the results rendered. This workload requires efficient reading of chunks of the matrix.

2.2 Implementation

The storage layer, implemented in C++, provides a common interface to access array based datasets. We create a StorageBackend by inheriting from the AbstractStorageBackend interface and implementing the methods described in Section 2.2.1 using an array storage system. This design allows us to interchange storage systems for evaluation without affecting other layers within the overall system. To compare the trade-offs between different array store systems for each workload, we implement three versions of StorageBackend using two existing systems, HDF5 [33] and TileDB [31] and our own implementation as a baseline.

We begin by describing the API in Section 2.2.1 followed by a description of each implementation in sections 2.2.3–5. The command line programs that are available for this layer are described in Section 2.2.6. Finally we discuss some overall optimizations in Section 2.2.7.

2.2.1 StorageBackend API

The StorageBackend API allows creating, reading, and writing of arrays. Each array corresponds to a patient’s unique medical record number (mrn). The API is as follows:

```c
1  ArrayMetadata get_array_metadata(string mrn);
2  void create_array(string mrn, ArrayMetadata* metadata);
3  void open_array(string mrn);
4  void read_array(string mrn, int ch, int start_offset, int end_offset, float* buf);
```
void write_array(string mrn, int ch, int start_offset, int end_offset, float* buf);

void close_array(string mrn);

**ArrayMetadata get_array_metadata(string mrn)**

Each array has an associated `ArrayMetadata` object which describes the size of the matrix. Specifically, `ArrayMetadata` stores three integer values which are necessary for the spectrogram calculation and for the client to prepare the visualization. The values are as follows:

```c
int fs;
int nrows;
int ncols;
```

`fs` is the frequency rate at which the sampling occurred during the EEG scan. `nrows` and `ncols` represent the number of rows and columns in the matrix respectively. Since each array based system has a different way of storing metadata values, the `ArrayMetadata` object provides a common interface to access and deserialize them.

void create_array(string mrn, ArrayMetadata* metadata);

The `create_array` function takes a patient’s `mrn` and a pointer to a `ArrayMetadata` object as input. The function is responsible for defining the array and persisting the `ArrayMetadata` object to disk.

void open_array(string mrn);

The `open_array` function prepares an array for reading or writing. This function typically caches certain values for more optimized used. Value caching is discussed further in Section 2.2.7.
void read_array(string mrn, int ch, int start_offset, int end_offset, float* buf);

read_array takes in a channel (column) to read, ch, and a start_offset and end_offset describing the subset of rows to retrieve. When requesting raw EEG data, a single column is passed in. When requesting a spectrogram, we give a special channel value ALL to retrieve all columns for a given time range. We store data values in the provided buffer, buf.

void write_array(string mrn, int ch, int start_offset, int end_offset, float* buf);

Similar to read_array, write_array writes the given column ch beginning at start_offset and ending at end_offset with the values written from the buffer, buf.

void close_array(string mrn);

close_array frees any cached values for the given mrn and releases any other cached resources back to the storage system.

2.2.2 EDFBackend

The EDFBackend is a read-only implementation of the StorageBackend interface. This provides easy access for testing other parts of the system and conversion between backend types. The backend uses an existing EDF library implementation to read given EDF files. This backend is not included in the evaluation since the ingestion cost is zero and the visualizations are not stored in the EDF format.

2.2.3 BinaryBackend

The BinaryBackend, inspired by the EDF format, acts as a baseline for other implementations. We chose this as the baseline since the format was designed to meet the workload requirements and is not a general array storage system. Comparing the performance of
other systems to the BinaryBackend allows us to see the overhead a more general purpose system incurs.

We use a simple on disk representation for the BinaryBackend. As Figure 2-1 shows, the first byte of the file contains a uint32_t, containing the length of the header information, \( n \). The following \( n \) bytes contain a JSON encoded string with the ArrayMetadata information. Following this, we write the array to disk in column-major order for efficient read access. Since the dataset is dense and the number of rows are known in advance (\( \text{nrows} \)), simple offset calculations determine the location to read or write the file.

![Figure 2-1: On disk data layout for the BinaryBackend implementation.](image)

When performing multiple I/O operations consecutively, for example, writing a file out in chunks, this implementation suffers from multiple disk seeks. Each time we perform a read or write operation, the file is opened, we seek to the appropriate location and finally file is closed. To amortize the cost of disk seeks, we increase the size of the data chunks we read or write.

### 2.2.4 HDF5Backend

The HDF5Backend uses the Hierarchical Data Format (HDF) version 5 [33] to store the data. HDF5 is an open source ‘technology suite’ which meets the requirements for storage
for our system. HDF5 is capable of supporting diverse datasets at scale, fitting the array based model of our EEG data nicely.

Integrating HDF5 as a StorageBackend is straightforward. Using the HDF5 C++ bindings, we wrap the library calls to read and write data without API methods. HDF5 has its own metadata storage which we utilize for keeping the ArrayMetadata. One interesting note is in HDF5 you must specify if the array will read or write the data in chunks so that the system can layout the data in the chunk sizes you specify. Our initial implementation left this detail out, resulting in a 2x cost in performance.

2.2.5 TileDBBackend

TileDB [31] is an ongoing research project by Intel Labs. TileDB specializes in storing sparse arrays, offering scalable and efficient access to these datasets. An input dataset can be an arbitrary multidimensional array, fitting our model for a storage system. Although our datasets are dense, we wanted to investigate the performance of TileDB for dense datasets and see if it can compete with a more mature project such as HDF5.

TileDB offers a C API in a shared object library which we use to create the TileDBBackend. At the time of writing the project does not yet support metadata, so we store the ArrayMetadata as JSON data in a separate file.

Although TileDB primarily supports sparse array data, it allows reading and writing of dense arrays. When defining an array, a special dense flag is set, allowing the system to make dense array optimizations, similar to HDF5. First, in the dense format, TileDB does not need to store coordinate information for each attribute. The array ranges are defined when the array is created – if the data is dense, the coordinate information is implicit. This greatly affects performance, since without this flag, each write incurs a 3x overhead. The system must write the float32 attribute and two int32 coordinate, (i, j), values. TileDB allows GZIP compression of the coordinates, but this leads to a space versus computation
trade-off for compression and decompression when processing the array.

Initially we used the sparse format with GZIP compression for the coordinate values, however the implementation was 2x slower than the dense implementation when performing writes.

### 2.2.6 Command Line Programs

The storage layer offers two command line programs `edf_converter <mrn> <desired_size>` and `data_to_file <mrn> <type>`. The `edf_converter` program takes a mrn as input and converts it to the appropriate StorageBackend format. There is an optional parameter `desired_size`, specified in gigabytes. We use this parameter when evaluating the implementations – a small data file can be replicated to produce a testing file of arbitrary size. The `data_to_file` program is used to serialize data from a StorageBackend to CSV or binary output. These programs are used extensively for testing, ingestion, and evaluation.

### 2.2.7 Optimizations

When designing the AbstractStorageBackend, performance was an important factor. Internally, we allow child classes to specify a template type T as a cache value and also store a mapping from mrn to ArrayMetadata. Each implementation stores some basic information in memory rather than repeatedly fetching information from disk. For example, by calling `open_array`, we cache objects related to the array for future use until a call to `close_array`. This allows us to keep a consistent external API and internally manage access to each library efficiently.

### 2.3 Evaluation

We evaluate the performance of the different StorageBackend implementations by measuring the time taken to perform file ingestions (described in Section 2.3.1) and spec-
program precomputation (described in Section 2.3.2). All of these experiments ran on the CSAIL OpenStack infrastructure. We allocated three instances on ‘isolated’ hosts, where the virtual machine runs using an actual hardware thread. Each machine had four cores available with 8GB of RAM and Intel Xeon 2.27GHz processors running Ubuntu 14.03.04. Each machine had 330GB of disk space available for the calculations. To reduce variability between experiments, we ran each experiment a total of three times, once on each machine. Between experiment runs, we cleared the file system cache to reduce variations from caching.

These two use cases were chosen since they are the primary workloads of the system and we wanted to understand how the different implementations performed for varying input sizes.

The source code and necessary scripts to run the experiments are available in the project repository [21]. The repository contains instructions for building the project and importing a dataset for testing.

2.3.1 Ingestion Experiment

The ingestion experiment takes an EDF file and converts it to the appropriate StorageBackend format. In the MGH corpus, the largest EDF file is 150GB and files commonly range from 5GB to 20GB. Thus, to understand how the different implementations would scale, we use files from 1GB to 128GB in size, varying by powers of 2, for the experiment. In addition to varying the file size, we also vary the chunk size of the reads and writes. To reduce system memory consumption, we varied chunk sizes by powers of 2 from 64MB to 512MB. Experimentally, we found that a chunk size of 256MB was optimal across the file sizes, thus we evaluate the different backends with this chunk size.

Since this workload is essentially just a data copy, we use the Unix cp program as a reference for the ‘best’ possible performance. Each system must convert values from the
EDF format to the implementation format, thus any overhead from cp is a result of the conversion, not the disk I/O.

Figure 2-2: Total runtime for different backend implementations in the ingestion experiment.

As Figure 2-2 shows, when compared to the cp program, we can see the conversion overhead for the BinaryBackend and HDF5Backend implementations. For 128GB files, these implementations outperform cp, it is possible that cp incurs more disk seeks than the other implementations if cp uses smaller chunks to copy to and from disk.

The HDF5Backend and BinaryBackend scale much more efficiently than the TileDBBackend. Since the ingestion experiment is a write heavy workload, the Figure shows both the overall performance as file size increases and speed of writes in each system. Since the TileDBBackend scales linearly with the file size, we assume that writes in this system are much more expensive than the others.

TileDB writes data cell-by-cell, whereas the HDF5Backend and BinaryBackend implementations can write an entire buffer at once. The cell-by-cell writing incurs a large number of calls to memcpy which may account for the performance gap that Figure 2-2 shows.
Future version of TileDB will support bulk writes, this functionality is currently being implemented.

2.3.2 Precompute Experiment

The precomputation experiment calculates the spectrogram for a given mrn. We use the same file sizes as in the ingestion experiment, using the ingested data for this calculation of the spectrogram. As in the ingestion experiment, we vary the chunk sizes for writing the spectrogram to the StorageBackend. In the evaluation, we use the same 256MB chunk size used in the ingestion experiment.

For this experiment, we evaluate the read time, write time, and total time across the different StorageBackend implementations. The read time is calculated as the sum of all reads across regions when producing the spectrogram and dominates the overall runtime. The write time is computed as the total time to write the calculated spectrogram values across all regions. The total time accounts for the read and write times in addition to the FFT calculation for computing the spectrogram.

Read Time

As Figure 2-3 shows, both the BinaryBackend and HDF5Backend near identical read times. Compared to Figure 2-2, Figure 2-3 shows that reads in the TileDB system are much more expensive that writes. This is likely caused by the system not optimizing for the dense case. In the sparse case, coordinate values are generated on the fly to yield each nonzero attribute in a query result. In the dense case, the values need not be generated since each attribute must be consecutive. TileDB does not yet support this optimization.

Another reason for the large performance gap could be the way TileDB internally reads data from the disk. The BinaryBackend takes an entire chunk of memory and interprets
it as an array of float values. Given the performance similarity to the HDF5Backend, we can assume a similar computation is done when extracting data from HDF5. As with writes, TileDB reads data cell-by-cell, copying values into a destination buffer, forcing a call to `memcpy` for each cell that is read.

**Write Time**

Figure 2-4 shows that the HDF5Backend outperforms the BinaryBackend for large inputs. It is possible this difference is caused by HDF5 buffering writes which the BinaryBackend does not. Since the write sizes are much smaller than in the ingestion experiment, it is likely that the overhead of disk seeks dominates the BinaryBackend implementation.

**Total Time**

We show the total runtime of the precompute program in Figure 2-5. We can see that the total time is dominated by reading the data, with a small amount of time for processing the FFT across the data windows.
2.3.3 Storage Overhead

In this section, we discuss the storage overhead for using each StorageBackend type. We define this overhead as any additional disk space required aside from the desired size of the file. As Table 2.1 shows, the HDF5Backend is the only implementation with a small overhead. Both the TileDBBackend and BinaryBackend have a small, constant overhead for bookkeeping structures, but this is not a considerable amount of space relative to the data storage.

TileDB does not incur a storage overhead since we defined the array to be dense. When an array is sparse, TileDB has a significant storage overhead. Our initial evaluation showed a 38% overhead with GZIP compression of the coordinates. This overhead is because of the storing the coordinate values (the \((i, j)\) coordinates within the matrix) for each attribute. TileDB does have some small bookkeeping structures, in both the dense and sparse cases, however these are of constant size for a given array and also compressed using GZIP. As Table 2.1 shows, these do not contribute any overhead.

Figure 2-4: Total write time for different backend implementations in the precompute experiment.
Figure 2-5: Total runtime for different backend implementations in the precompute experiment.

Table 2.1: Storage overhead costs for each StorageBackend type.
Chapter 3

Compute Layer

The compute layer is responsible for two important parts of Pinky’s functionality. First, this layer handles interaction between the storage layer by using the `StorageBackend` abstraction to read and write datasets. Secondly, it interacts with the visualization layer by performing calculations at the client’s request and transferring data back for rendering. In addition, this layer also contains implementations of optimized algorithms for processing EEG data to calculate the spectrogram.

3.1 Design

Since this layer of the system supports a variety of features, we separate the design points for each. Similar to the storage layer, minimizing runtime was one of the key design goals. Another design point we focused on was enabling developers to easily modify existing or add new processing algorithms.

3.1.1 Websocket Server

The websocket server is responsible for transferring array based data to the client for rendering. The server must be able to quickly serialize data from the storage layer, namely multidimensional arrays of floating point values. The server must be able to run concurrently in order to handle multiple client requests, processing the requests in parallel to
reduce latency.

3.1.2 Webapp Server

The webapp server is responsible for serving web resources such as JavaScript, HTML, and CSS to the client. The scripts sent to the client are responsible for communicating with the websocket server to retrieve data for display. This server is separate from the websocket server to simplify the implementation since the webapp server is not subject to the same performance requirements when transferring datasets over the network. The separation of these two servers allows for future modules to be easily added to the webapp framework using existing frameworks (see Section 3.2.2) such as authentication.

3.1.3 File Ingestion Daemon

The file ingestion daemon is responsible for collecting new files that are added to the system and converting them for use. This makes use of conversions between the EDF format and the required format for the StorageBackend that is in use. Input files are initially received in the EDF format and must subsequently be converted without affecting other clients who are using the system.

3.1.4 Spectrogram Calculation

The EEG spectrogram transforms the raw voltage readings from a patient screening, converting them into frequency space. Formally, if \( x \) is a signal of length \( N \), we consecutively take windows of \( x \) of length \( m \) (\( m \ll N \)), creating the matrix \( X \in \mathbb{R}^{m \times (N-m+1)} \) where consecutive segments of \( x \) are consecutive columns in \( X \). Thus, the first column of \( X \) is \([x[0], x[1], \ldots, x[m-1]]^T\), followed by \([x[1], x[2], \ldots, x[m]]^T\), and so forth.

Following this, the spectrogram of \( x \) with a window size \( m \) is defined as the matrix \( \hat{X} \), where the columns of \( \hat{X} \) are the Discrete Fourier Transform (DFT) of \( X \), or \( \hat{X} = \hat{F}X \).
The spectrogram calculation depends on the windowing function, the window size, the overlap between windows and the number of points chosen when performing the DFT. We make these parameters configurable to suit an analysts’ needs; supporting different kinds of analysis and also for testing different configurations. The default configuration uses the Hamming windowing function, a window size defined as \((\text{nsamples} - \text{nfft}) / (\text{fs} * 4)\), and overlap equal to \(\text{fs}\).

The Hamming window is defined as \(w(n) = 0.54 - 0.46 \cos(\frac{2\pi n}{N-1})\) for a discrete signal \(w[n], 0 \leq n \leq N-1\). \text{nsamples} is the number of samples in the input signal, \(\text{nfft} = \text{next\_power\_of\_2}(\text{fs})\), and \(\text{fs}\) is the sampling frequency of the input signal.

### 3.2 Implementation

We implement the compute layer almost entirely in C++ except for the file ingestion daemon and webapp server, which are written in Python. The Python implementations comprise a much smaller part of the compute layer codebase since there are a number of open source libraries that we make use of. Although there is a development and infrastructure overhead for using multiple languages within the system, Python seemed the appropriate choice in this circumstance due to the reduced developer time and availability of mature libraries. C++ was chosen for the other parts of the module since the performance gains with a compiled language were necessary. C++ was chosen over pure C because of the available libraries for linear algebra processing [28] and network communication [27].

#### 3.2.1 Websocket Server

The websocket server primarily depends on the open source library Simple-WebSocket-Server [27]. This library implements the websocket protocol and provides a simple interface for transferring data across the network. The library was not without its own bugs, over the course of developing Pinky we reported and helped debug concurrency issues with the generous help of the project’s maintainer. These bug fixes resulted in large performance gains from the original library implementation.
When loading the Pinky frontend into their browser, the client sends a request to the websocket server to open a connection. When the client wishes to browse a patient’s data, a request is sent specifying the patient’s mrn, a start_time and end_time given in hours.

The websocket server then requests a spectrogram with the given values either computing the spectrogram on the fly or using a precomputed spectrogram calculation. The websocket server can potentially downsample the response to meet client latency requirements, the details of this are described in Chapter 5.

To send data back to the client, a JSON encoded header is created, notifying the client with the calculation results. The header contains a small amount of metadata such as the sampling rate, fs, calculated start_time and end_time (we validate these values to the bounds of the dataset), spectrogram matrix dimensions and the region (LL, LP, RL, or RP) for the calculation. The messaging protocol transfers the binary array data encoding a header length in a uint32_t followed by the serialized header information and the matrix data. All messages are byte aligned to 8 bytes, sometimes a small amount of padding is added for performance reasons. We use the same binary representation as the BinaryBackend uses. See Figure 2-1 for details.

The websocket spawns \( n_{\text{processors}} - 1 \) threads to serve requests with. For development and testing we use the CSAIL OpenStack framework to create multicore virtual machines of different sizes for testing.

### 3.2.2 Webapp Server

We implement the webapp server using the Python microframework Flask [3]. The Flask framework allows simple serving of HTML webpages with support of the Jinja2 [4] templating system. This is vastly easier to develop and maintain than a comparable C++ im-
plementation. Since performance is not an issue, Python is a viable option.

The server provides web resources for the client as well as pages with information about the project.

### 3.2.3 File Ingestion Daemon

The file ingestion daemon monitors the server’s filesystem for changes in order to ingest new data files. Taking advantage of the Python Watchdog [37] library, the script is able to receive events for changes to the filesystem for a given folder. Upon receiving a notification, the script will convert the file to the necessary format and also precompute the spectrogram. To process the file, the script calls the command line programs `edf_converter <mrn>` (Section 2.2.6) and `precompute_spectrogram <mrn>` (Section 3.2.5).

This functionality allows an administrator to dump EDF files onto the filesystem of the server and allow an analyst to automatically query them after conversion.

### 3.2.4 Spectrogram Calculation

The spectrogram calculation involves reading portions of the raw EEG data, taking the FFT of the data with a sliding window in time and storing the results in a matrix for visualization. The spectrogram computation can vary based on a number of parameters. To simplify the implementation, we store all relevant spectrogram parameters in a `SpecParams` object. When a client requests a spectrogram for a given region of the brain, we create this object, building the parameters from the input `mrn`, `start_time` and `end_time` parameters.

The `SpecParams` object contains the following attributes, the comments next to each parameter describe it’s use.

```plaintext
string mrn; // patient medical record number
StorageBackend* backend; // array storage backend
float start_time; // start time of the spectrogram
float end_time; // end time of spectrogram
```
Using the `SpecParams` object, we can calculate the spectrogram for the EEG data.

Section 3.1.4 details the description of the algorithm. The algorithm’s core implementation is as follows:

```c
void eeg_spectrogram(SpecParams* spec_params, int ch, fmat& spec_mat)
{
    // Get the column which contains the first channel for the region.
    ch_idx1 = DIFFERENCE_PAIRS[ch].ch_idx[0];
    frowvec vec1, vec2; // initialize read vectors
    read_array(mrn, ch_idx1, start_offset, end_offset, vec1);

    // Get the column which contains the next channel for the region.
    ch_idx2 = DIFFERENCE_PAIRS[ch].ch_idx[i];
    read_array(mrn, ch_idx2, start_offset, end_offset, vec2);
    // take the difference between the channel pair
    frowvec diff = vec2 - vec1;

    // fill in the spec matrix with FFT values
    FFT(spec_params, diff, spec_mat);
    swap(vec1, vec2);
}
spec_mat /= (NUM_DIFFS - 1); // average diff spectrograms
spec_mat = spec_mat.t(); // transpose the output
```
The definition of the constants DIFFERENCE_PAIRS and NUM_DIFFS are omitted for simplicity. The DIFFERENCE_PAIRS simply defines which channels to take differences to form a region (see Section 1.2) and NUM_DIFFS=4 since we compute the spectrogram across four different regions of the brain.

We implement the FFT algorithm with the FFTW library [16] for optimal performance. We make use of the Armadillo C++ linear algebra library [28] for simplifying vector and matrix calculations.

3.2.5 Command Line Programs

The compute module offers two command line scripts, test and precompute_spectrogram <mrn>. The test script tests the functionality to an algorithm or StorageBackend. The precompute_spectrogram program will take a mrn as input and calculate and store the spectrogram for the given mrn on disk.

3.2.6 Optimizations

The implementation of the eeg_spectrogram algorithm design aims to minimize memory consumption. For this reason we reuse the vec1, vec2, and spec_mat buffers during the calculation. We considered computing each of the differences for a region in parallel, however computing each region in parallel (parallelized at the websocket server) was performant enough. Previous iterations involved serializing the output of the spectrogram matrix (spec_mat), however we found that instead we could directly access a pointer of the raw matrix memory. This optimization help significantly since it eliminated a memory allocation and data copying before sending over the network.

Another minor optimization is the use of the static inline keyword. We use this for helper functions to reduce function call overhead. In addition, we always pass Armadillo objects by reference and not value, avoiding a copy on function calls.
Chapter 4

Visualization Layer

The primary function of the visualization layer is to render a spectrogram served by the compute layer. This module is also responsible for providing a usable interface for an analyst to work with the datasets. Minimizing latency is an important design goal since increased latency can dramatically reduce an analyst’s efficiency if they must continually wait for the interface to respond to queries.

We chose to build browser based visualizations to simplify the use for an analyst – the only software required is the browser itself. In addition, the analyst does not require special hardware since the server handles the intensive storage and computations. This choice makes the job of visualizing interactively more difficult to design and implement since the browser is much more limited in network bandwidth and rendering capabilities compared to a native application.

4.1 Design

4.1.1 Interface

The interface should provide the analyst with the ability to specify a patient mrn to query, a start_time and end_time and also afford quick navigation between time ranges. The workflow that we anticipate is that an analyst will load a single patient file and browse
subsequent time windows. Upon reaching a section of interest, the analyst should be able to
zoom in to see further details. The interface should also give the analyst information about
the visualization, such as interactive axes. The interface should also provide user feedback
for errors, validating data on the client to prevent the inadvertent issuing of queries.

4.1.2 Communication

Optimizing communication is important to avoid creating a bottleneck that can affect la-
tency. The point which is most likely to be a bottleneck is deserializing the data received
from the network.

4.1.3 Rendering

The client rendering must be able to efficiently render large matrices of floating point val-
ues, on the order of millions of points. The spectrogram visualization is created by taking
the intensity of the \((i, j)\)-th entry in the spectrogram and mapping it to a rendered color in
the \((i, j)\)-th pixel on screen. Some amount of data aggregation is acceptable, for example
downsampling, however the analyst must not experience degradation of the overall data
quality.

4.2 Implementation

We implement the visualization layer primarily using HTML, JavaScript and CSS. WebGL
performs the spectrogram rendering to take advantage of a client’s GPU.

4.2.1 Interface

Figure 4-1 shows the implementation of the interface. The interface shows a sample of a
patient with mrn ‘005’ rendered between the second and third hour of the scan. These
parameters are located in the top bar, in addition to controls which enable the analyst to
scroll to previous or next hour with a single click. The scrolling interval defaults to 1 hour,
but is configurable in the settings window.

![Figure 4-1: Screenshot of Pinky’s user interface. An analyst can view the four spectrograms corresponding to different regions of the brain and query for different time ranges to view.](image)

Clicking the small gear on the far right of the interface opens the settings page, the available options are shown in Figure 4-2. The settings page allows an analyst to change the rendering mode, time interval and select options for interpolation and the visualization scale. In addition, there are two keyboard shortcuts which allow an analyst to change the amplitude scale or zoom in on the visualization. Figure 4-3 shows the result of a single region when a user has zoomed in.

The interface shows a spectrogram for each region of the brain, each region is label in the upper left hand corner with LL, LP, RP, or RL. As Figure 4-3 shows, next to these labels, is a small box containing axis information, specifying where the user’s mouse currently is. In this box the current timestamp (x-axis) and frequency (y-axis) are show. In addition, the box shows the current amplitude value in decibels, dB, and the current amplitude range.
Figure 4-2: Screenshot of the settings modal. The options allow an analyst to tune the visualization coloring and change the default time window query size.

Figure 4-3: Screenshot of a zoomed spectrogram view of a single rendered region with dynamic axis labels.

While loading, the interface blurs the spectrograms and presents a loading bar to the user, as shown in Figure 4-4. A delay between the rendering of each region can cause confusion about the current rendered data. Blurring a region and placing a loading bar on it distinguishes a region that has not yet updated with the client’s latest query results.

If an analyst enters an invalid mrn, the interface responds by clearing all of the rendered spectrograms and displaying a small error message, shown in Figure 4-5. An invalid mrn is simply one which the StorageBackend does not contain an array for. This could be from a user slip error or if the system is currently ingesting the array. This user interaction is important to avoid analyst confusion when issuing an invalid query.
Figure 4-4: Screenshot of loading interface. When a new query is issued, the old data is blurred to avoid confusion of current and past results during query execution.

The CSS library Materialize [2] was used to layout the page structure and keep a consistent style throughout the webapp.

4.2.2 Communication

A websocket communicates with the compute layer websocket server using a binary protocol described in Section 3.2.1. We send requests as JSON encoded data and receive binary responses containing the computed spectrogram data for a given region.

We make use of the reconnecting-websocket [20] library to ensure a smooth user experience if the analyst does not use the page and the connection closes. This library automatically reopens the connection, instead of forcing the analyst to refresh the page altogether.
4.2.3 Rendering

WebGL is a JavaScript API which can render interactive 2D or 3D computer graphics without the use of any third party plugins. The initial implementation used the open-source library WebGL-Spectrogram [5]. This library has the functionality to render the spectrogram of an audio file using a lightweight Python websocket server. We modified this library to a more general version to contain multiple canvases, one for each brain region, and to communicate with the compute layer websocket server.

4.2.4 Optimizations

WebGL was chosen since it is much more performant than using a browser’s canvas object or rendering DOM elements directly. Since each spectrogram is an array on the order of millions of points, we would not be able to achieve the latency required for interactivity without the GPU rendering. Development time suffers from the use of WebGL since it is difficult to understand the programming model without some background in graphics rendering. In addition, we use JavaScript typed arrays to transfer the binary data from the
websocket to the GPU. JavaScript typed arrays are array-like objects providing access raw binary data. JavaScript engines optimize these arrays giving higher performance than the traditional JavaScript Array object.
Chapter 5

Visgoth System

The client-server architecture is a common model for interacting with remotely stored datasets. There has been substantial work supporting the storage and processing of large datasets [30] [35] [38]. However, we find there is a gap when an analyst is attempting to access the stored data. The size of storage and compute clusters allow petabyte scaling of datasets, but a local client machine can only process a fraction of this data at any given time. With the exception of mobile web designs, current architectures ignore the heterogeneity among client hardware. When serving data to a client, regardless of the current state of available client resources, the same dataset is returned in response to a query. If a system were able to take the client state into account, responses could be tailored to the individual client, providing a more uniform response latency across clients.

Visgoth is a system aimed at dynamically changing a server response based on profiling the state of the client and server machines. One use case is visualization applications, in which the response data that the user views can vary, since some degradation of data quality can be tolerated in return for low latency responses. The way that a visualization is changed is an application-specific trait, but could include downsampling or aggregating data points. Visgoth provides profiling information about the current system state to allow developers to adapt their results.

Examples where such a system would be valuable could be a data scientist working
with large amounts of time series data, for which coarser results are acceptable for partial analysis. Another case could be a user viewing a social media site with a low bandwidth connection. A text-only webpage or one with limited multimedia could be an acceptable experience as opposed to no progress being made while loading images.

We have built a prototype system to handle rendering the EEG spectrogram visualizations in the browser. We begin with related work in Section 5.1 followed by an outline of the design and implementation in Sections 5.2 and 5.3 experimental results in Section 5.4 future work in Section 6.4.1

5.1 Related Work

Currently, no visualization library takes advantage of information about the client’s real-time state, only static hardware configuration. There has, however, been substantial research on optimizing performance of large dataset visualization.

Interactive querying of multidimensional dataset visualization was explored by the im-Mens project [39], a web-based visualization library with a server-client architecture. im-Mens was able to achieve higher performance for data interaction through preprocessing of large multidimensional data cubes. Data was decomposed into data tiles, or subsets of the larger dataset, which turned out to be more flexible to compute on and serve to the client.

Similarly, the M4 system [22] uses an aggregation-based time series dimensionality reduction technique to provide error-free visualizations at high data reduction rates. This system targets particular data reduction techniques, but does not account for client heterogeneity or varying system resources to apply the reductions.

The ForeCache system [6] uses predictive models based on recent user interactions and requested data characteristics (e.g. histograms) to prefetch similar data tiles for display. This system is able to adaptively serve visualizations to the client, but currently only ac-
counts for user behavior, not differences in the clients’ machines. Also, while ForeCache can choose which tiles to prefetch, it does not ever adjust the tile size itself.

BigDawg [14] is a big data storage system with proposed solutions for visualization interfaces. It is currently under development and affirms the need for a large-scale visualization system. Part of the motivation for BigDawg’s visualization interface is the need for an analyst to ‘drill down’ into datasets, interactively choosing portions to focus on and smoothly panning and zooming among these sections.

The work done by Lee, Ko, and Fox [24] addresses adapting content for mobile devices. This is done through a series of “transcoding” techniques, essentially data reductions on different types of media content based on the client hardware. The system bases transcoding operations on client hardware types, but does not take into account client resource state that change over time, such as network connectivity or bandwidth.

5.2 Design

The Visgoth system is split into three main components: the Visgoth server, the application server, and the application client (the web browser). The lifetime of a client interaction in an application using Visgoth can be split into two phases: request and response.

First, during the request phase, profiling information about current performance is collected on both the client and the application server. These statistics are sent to the Visgoth server, which uses static regression models to predict a data reduction factor. The application server then uses the data reduction factor to adapt the visualizations that the client receives. In Figure 5-1 we show an example request that the Visgoth server would receive. The request includes the normal application request parameters that are used to request a visualization, along with the Visgoth profiling information (highlighted).
Second, during the response phase, the Visgoth server uses the regression model to suggest a data reduction factor to the application server. Figure 5-2 shows an example response from the Visgoth server, which suggests a data reduction by a factor (extent) of 10. The application server then applies this reduction, serving coarser data to the client. In the context of the visualization application, this would be equivalent to the application server sending a blurrier image back to the browser.

In the following sections, we describe these two phases in more detail. In Section 5.2.1, we discuss the Visgoth profilers that are installed in the application server and client. In Section 5.2.2, we discuss how the Visgoth server uses this information to predict a data reduction factor.


5.2.1 Profile Collection

Profile collection is needed for both the initial training of the Visgoth regression model, as well as the actual application requests. These two phases of profile collection are essentially the same, except that during the training phase, the Visgoth server sets a default data reduction factor, versus a predicted factor during the application phase. Eventually, Visgoth’s regression model could be made dynamic by also training on incoming application requests, but dividing profile collection into two phases is simpler to implement for this prototype.

Profiling information is collected both at the client and the server to capture different types of data. We collect profiling information from three different categories: static, dynamic and application specific data. Static data such as the client or server hardware is useful, especially since the client hardware can be heterogeneous. This allows Visgoth to fine tune it’s models based on the hardware being used. Dynamic system statistics such as network or memory usage allow Visgoth to make up-to-date prediction decisions. Finally, application specific profiling such as rendering or computation time allow a developer to choose which factors are most important on an application basis.

On both the client and server, Visgoth allows the developer to set a default window size for individual statistics. This indicates to the Visgoth profilers how many of the most recent samples for a profile statistic to keep track of. The idea behind using a time-based window is to keep a running average of the most recent profile values, but not to keep it so large that the values sent to the Visgoth server are no longer relevant.

In general, profile information on the application server is easier to collect because the developer should have direct access to the server’s machine. There is already widespread support for profiling a local machine, which we discuss in more detail in Section 5.3. Here,
we focus instead on the more difficult case of collecting information on the client.

**Client Profiler**

Profiling the client is tricky because, for security reasons, most browsers purposefully do not expose information about the client machine. Certain pieces of static information are readily available, such as the browser and machine type from the “User-Agent” string, assuming that the client does not mock it. However, static hardware information such as the RAM size is in general inaccessible, to our best knowledge.

The Visgoth client profiler has better support for application-specific dynamic profiling. Most modern browsers support `window.performance`, a high-resolution time data API that allows a web client to set marks and get the current timestamp (see Section 5.3). With this API, we are able to get real-time information about the network and where in the code the client spends most of its time.

We can get approximate numbers for network latency and network bandwidth by using the `window.performance` API and slightly modifying the application server. Because the client and server’s clocks are not synced, the best that we can do is measure the total round-trip time between the client’s request and the client’s receipt of the response. In that case, we do not want to measure an actual application request’s time since this would likely include extra computation on the server side that the client cannot measure. Instead, we can make a small modification to the application server to include a dummy endpoint that serves a small static piece of nonsense data. The portion of time on server computation for such a request is unlikely to be significant relative to the overall request time.

Assuming some margin of error, we can now profile the network by sending dummy requests to the application server. The network latency is equal to the round-trip time. The network bandwidth is equal to the size of the request and response packets divided by the round-trip time.
Network health can vary significantly across time. In order to get accurate real-time statistics, the client must poll the application server with the dummy requests more frequently than it would send normal application requests, which are triggered by user interactions. However, the client should not poll the network so often that it interferes with the application’s overall latency. Similarly, the nonsense data that the server returns should not be so large that it dominates the client’s bandwidth. A developer using Visgoth will have to tune the polling rate and the size of the response data to make sure there is no noticeable difference in their application. Eventually, Visgoth could also vary the size of the response data, using the same data reduction factor that it predicts according to current profile information.

In addition to the network, Visgoth can collect profiling information about client-side computation, again using the window.performance API. Visgoth exposes a higher-level statistic API to the developer. The developer can produce instances of statistic, each of which has a tag and `begin()` and `end()` methods that the developer can use to mark the beginning and end of the procedure that they want to profile. When `statistic.end()` is called, the `statistic` dumps the measured time since `statistic.begin()` was called for the same instance to Visgoth’s global state.

This simple API is sufficient to measure throughput information as well as latency. For instance, in a visualization application, one can measure the frames rendered per second. This is done by counting the number of frames rendered within some time interval and dividing by the rendering time, which can be measured using the Visgoth statistic API.

5.2.2 Modeling

Visgoth’s goal overall is to provide a uniform experience across all clients, no matter the hardware or workload variation across the clients. To do this, Visgoth must be able to pre-
dict what the total latency of a particular application request will be. Here, latency does not refer to the network, but to the time between the client interaction, e.g. a click on a webpage, and the time when the results are visible in the browser, e.g. an image is rendered. We hope that providing a uniform latency, while still serving as high of a data resolution as possible, will translate to a more uniform experience across all clients.

Visgoth predicts latency using a set of pre-trained regression models, one for each value of downsample, the factor that the application server reduces its data by. The features of each model comprise of the statistics from the profiles collected on the application server and client. We train a single regression model by setting a default value for downsample and running an application request for a set number of trials to get variation in the profiles.

Once the regression models are trained, the learning goal can be framed as follows: Given a current profile of the application server and client and a target value for latency, predict the value of downsample that will reach the closest to target latency value. More formally, Visgoth starts with a set of regression models $\{M_1,\ldots, M_n\}$, where each $M_i$ is of the form:

$$M_i(P) = \text{latency}$$

Here, $i$ is the value of downsample during the application requests that model $M_i$ was trained on. $P$ is a feature vector, containing all the statistics from the application server and client profiles. latency is the predicted latency if the current profile is $P$ and the application server reduces data served by a factor of $i$.

Visgoth takes as input a profile $P$ from the application server and client. Visgoth does not simply minimize latency, or else it would always send the least data possible. Instead, our goal is to strike a balance between latency and the quality of the client’s results. To do this, we instead try to minimize the difference between the actual latency experienced and a target value $l$ for latency. To determine what value of downsample the application
server should use, Visgoth computes the $i$ that minimizes the distance between the predicted latency and $l$:

$$| M_i(P) - l |$$

### 5.3 Implementation

**Server Profiler**

The server profile collects statistics concerning the server state or calculations. For the EEG application we use statistics recorded by the `collectd` library. We keep statistics pertaining to free memory, user CPU, network and disk usage to monitor server load. Load could be increased by additional clients connecting or server resources being consumed by other server programs.

**Client Profiler**

The `window.performance` high-resolution time data API proved to be essential for gaining client profile information. `window.performance` is a widely supported API that provides functions for timing webpages. We used two main methods, `mark()` and `now()` to build the higher-level Visgoth statistic API. The `mark()` method can be used to set and name a mark at the time that it is called. The `now()` method returns the current timestamp, which can be used to measure against marks. These together can be used to measure different spans of application client code.

### 5.4 Results

We ran experiments using the CSAIL OpenStack infrastructure to host the visualization server and using our local machines as clients. The server, running Ubuntu 14.04.3, had 4 cores available with 8GB of RAM and Intel Xeon 2.27GHz processors. The client machine ran Ubuntu 14.04.3, and experiments were run in both Google Chrome (46.0.2490.86) and
We predicted that latency would correlate with downsample, since in general, if less data is sent, it is likely to take less time on the network. However, we also predicted that downsample wasn’t the only factor in latency. This would indicate that there are application server- and client-specific factors also at play.

To validate this prediction, we ran the same request on the visualization application 100 times each for downsample values from 1 to 16. We recorded the profile and total latency for each request. In Figure 5-3, there is indeed a downward trend in latency as downsample increases. However, there is also significant variation in latency across all columns. In fact, every value tested for downsample achieved a latency of about 3 seconds for at least one application request. We can conclude from this data that there are indeed other factors that determine the variation in latency besides just downsample.

Although we’d hoped to build a model on all of the features in the server and client profile, we chose for the sake of time to focus on a single feature in order to quickly validate the Visgoth system’s utility. We looked for a single feature that shared a strong correlation with latency.

We first looked at rendering throughput, measured by taking the inverse of the time needed to render a single frame of the spectrogram. The results are shown in Figure 5-4.
We can conclude that Firefox in general has a much more consistent and lower rendering throughput than Chrome. However, as was the case for downsampling, rendering throughput does not seem to correlate with latency. This shows us that for this particular application, rendering time is unlikely to be a bottleneck.

Finally, we did the same with latency versus bandwidth, shown in Figure 5-5. Just by visual inspection, bandwidth clearly had the strongest correlation with latency, so we decided to train regression models on this profile feature. We used a robust Theil-Sen regression, fit to an equation of the form:

\[ \text{latency} = a + \frac{b}{\text{bandwidth}} \]

We used 75% of our dataset of 1600 points to train the 16 regression models, one for each downsampling value. The coefficient of determination $R^2$ was 0.71 for model $M_1$, for which there was no data reduction, and over 0.99 for the other 15 models. Figure 5-5 shows a selection of these models, including $M_1$, along with the test dataset points.

The curve for $M_1$ predicts a much higher latency for higher bandwidths than the other models, while the other models also predict similar latency values. This indicates that at these bandwidths, Visgoth will almost always recommend some data reduction. This may not be so useful to an application developer, who could just set a default downsampling factor.
Figure 5-5: Overall latency of an application client’s interaction, plotted against approximate network bandwidth. Each curve represents the latency values predicted by one regression model. The points plotted are from the test dataset.

However, at lower bandwidths, around 500 KB/s, the same bandwidth is predicted to produce different latencies depending on which downsampling value is used. This is where the Visgoth system could have impact on the client’s experience. Within this range of bandwidths, it is unclear from a developer’s perspective which default value of downsampling would always produce the target latency value, or if there even is one.

To validate Visgoth’s utility at these lower bandwidths, we reran the same experiment, but this time using the downsampling value predicted by the Visgoth server. We set a target latency of 4 seconds. For this experiment, we hoped to see latency values that were uniform relative to those in Figure 5-3. The latencies also should have been centered at our target value across all application requests, regardless of the downsampling factor set by Visgoth.

Unfortunately, we did not see an appreciable difference in latency variation from Figure 5-3. However, we believe that this is due to our experiment setup. We were unable to implement frequent polling of the network in time for this experiment. Instead, we used the bandwidth measured during the previous request as the “current” bandwidth profile. We waited 9 seconds between requests to give the application time to receive a response, meaning that the bandwidth profile used to predict the downsampling for each request was
actually from 9 seconds ago. In order to get more meaningful results, it will be necessary to send Visgoth more recent profiling information.
Chapter 6

Discussion

6.1 Design Challenges

Working with the EEG data proved to be a challenge in itself. Starting with some Matlab processing scripts that worked for small datasets, we set out to build a scalable design. These initial programs would read data from an EDF file, perform the spectrogram calculations and render a static image of 1 to 2 hours of data. From this, our design separated the workflow into the three layers, storage, compute and visualization. Converting the implementations first to Python and then to C++ (see Section 6.2) and implementing a web based visualization was non-trivial since testing the correctness of the algorithms required all layers to be correctly implemented. While implementing any particular layer, we had to make backwards compatible methods test serialized Matlab data. Once we completed an initial implementation, it was easier to abstract different portions or change parts of the calculations for more optimized use.

6.1.1 Storage Layer

Creating a useful abstraction for the storage layer was essential for evaluating different datastores and maintaining scalability. The performance of this layer is likely the most critical for system performance and also the hardest to profile and analyze. The difficulty
arises from the scale of the datasets. Some bugs appear only for large files during testing, files which are larger than the disk space on development machines. To verify correctness with different backends, conversion between the backend data formats was crucial. Each backend can dump a dataset to a CSV format to import it into a different store for debugging.

### 6.1.2 Compute Layer

The main challenge with the compute layer was becoming familiar with the medical domain and the algorithms required for different computations. As discussed in Section 6.2, the performance of this layer can greatly affect the system as a whole since it connects the storage and visualizations layers to one another. The most important design decision here was the choice of language, a trade-off between development ease and performance. Having an efficient way to extract data from storage, compute on it, and send it to the client was essential. By choosing C++ over C we were able to take advantage of the Armadillo library for linear algebra processing and an open-source websocket library for sending data over the network. By choosing C++ over Python, we were able to achieve sane runtimes.

### 6.1.3 Visualization Layer

When designing the visualization layer, the main challenge we found was working with larger datasets in the browser. The network latency of the browser to receive such a dataset can be prohibitive in itself. The main fact we were able to take advantage of was that a human would be analyzing the final dataset. Analysts can view dramatically downsampled data to make the same conclusions. Because of this, we cap the size of a visualization density by the number of pixels in the client’s screen. This allows us to still give rich representations in our visualizations but without adding latency to the analysis.
6.2 Abandoned Designs

There were two main directions that the project could have taken that were abandoned. Initially we wanted to explore performing the EEG calculations on a GPU to reduce the computation overhead. It became clear that I/O was the primary bottleneck for analyzing the data and since the datasets cannot reside in memory, the overhead of transferring data to the GPU for processing would be prohibitive. In addition, after implementing an initial implementation in C++, the performance results were acceptable and allowed us to focus on the storage and visualization systems.

In addition, we had a version of the computation algorithms written in Python. The benefit of a Python implementation is that it allows rapid development and testing and also reduces the time for configure the system as it is more portable. HDF5 provides Python bindings to access array data, however, we found that the cost of serialization when sending data over the network dominated the cost of the calculation by several orders of magnitude. We believe that when trying to serialize the data to send a visualization over the network, Python converted the data to an internal type causing a massive slowdown. Rather than rewrite a websocket library, changing the language seemed appropriate.

6.3 Lessons Learned

The lessons here are a reflection of the overall project and throughout the development process. The first point is working with the open source community. Pinky makes use of the following open source projects [32] [13] [27] [31] [7] in each layer (in addition to many other larger community libraries) and without these, the project would not have been completed. These projects are often maintained by individuals as side projects and can contain bugs or be incomplete. Recognizing this fact early on was crucial to be able to contribute back to the projects and work with the project maintainers to improve the projects’ by submitting patches and helping pinpoint bugs.
Secondly, using the OpenStack infrastructure allowed us to easily allocate new machines for testing or experimentation. Initially, it was a struggle to setup the project and compile, however taking the time to automate this greatly helped to allow collaboration (such as the Visgoth project) and also to quickly add machines for demos or experimentation. By realizing that having an easy way to setup the infrastructure was necessary, we created Docker images of the project so it can easily be setup and used by others. Along with this vein, the entire project is available on Github in the hopes that other doctors will make use of the software.

6.4 Future Work

Pinky provides the most basic interaction that an analyst can use to work with EEG data, namely they can efficiently scroll through patient records on demand. In order to further the benefit of such analysis, we sketch the following potential designs for future projects.

6.4.1 Visgoth

Moving forward, the first step will be to rerun the last experiment from Section during which we tested the effect of predicting downsample on the consistency of the latency experienced by the client. For this run, the client profiler will have to be modified to send more up-to-date profile information. Specifically, the profiler should poll the network more frequently, so that the bandwidth information that Visgoth uses to predict downsample is still relevant. We’re optimistic that this will yield latency values close to our target, given the high $R^2$ scores of our regression models.

In future work, we hope to train similar regression models, but with a richer set of features that includes all of the statistics that we collect during profiling. Using bandwidth as the only feature produced high scores for our regression models because bandwidth also happened to be the single largest bottleneck during profile collection for our example application. However, if we had collected a more diverse dataset, it’s possible that one profile feature could have overtaken bandwidth as the bottleneck for some domain of inputs. Such
a pattern is impossible to learn if regression models are only trained on bandwidth.

Similarly, fitting the models to a linear basis in this first prototype worked well with a single bottleneck. However, in order to learn the regression pattern described above, it will be necessary to use more advanced non-linear regressor models.

Making the regression models dynamic is another possibility for producing latency values closer to the target. To do this, incoming application requests would also become part of the training or test dataset, and the models would update incrementally.

Visgoth can also be generalized to apply to generic applications that display large amounts of data to a user. Future work to make this possible would include creating a small library of statistics important to any application, such as network bandwidth. Developers could then take the following steps to integrate Visgoth into their own application:

- Install Visgoth profilers on application server and client.
- Use the Visgoth statistic API to record application-specific statistics.
- Define application-specific data reduction rules (e.g. serve text, but not images).

All other steps, including profile collection and training of the regression models, could be automated by the Visgoth system. In this way, Visgoth could remove much of the work that would have to be done per application and automatically provide a consistent experience across any such application’s clients.

6.4.2 Polystore

Pinky typically deals with the unique patient identifier known as their medical record number or mrn. The identifier is used throughout the system as an id, for example an analyst requests data by giving a patient’s mrn. Currently, the analyst would have to determine the mrn to use independently and would then able to query Pinky. An extension to the system
would be to allow analysts to query patient information to determine which cases to analyze. The current ad hoc system used at MGH involves using different storage systems, so this addition would expand Pinky to have a polystore architecture. BigDawg [14] addresses systems using polystore architectures.

In order to query patient information Pinky would have to incorporate a relational store for basic patient metadata and a free text store to keep doctor reports containing results of different examinations or medications. An analyst could then query across these stores, ultimately acquiring an mrn to query Pinky’s storage layer with. The importance of this lies in the ability to select patients with similar characteristics or treatments and analyze their data together.

6.4.3 Spectrogram Annotations

In addition to viewing the datasets, it would be beneficial for analysts to also be able to mark noteworthy sections of the spectrogram. This could be to make a reference for another analyst to see or to cross reference with other data, for example the raw EEG signals.

The implementation would involve selecting the frequencies of a spectrogram for a given time interval and storing these bounds of the matrix. The analyst would be able to associated a small amount of free text with the annotation, such as a note to why the event is interesting. Subsequent viewers would be able to see this annotation when browsing the spectrogram.

6.4.4 Change Point Detection and Clustering

As the corpus of EEG data grows, marking annotations by hand as described in Section 6.4.3 is rather tedious. A more automated way would be to scan the dataset and try to automatically cut the spectrogram signal into segments, characterized by changes in average power using the cumulative sum algorithm [18]. After generating the segments, one could cluster these segments and present them to an analyst for classification. During the
patient scan, the majority of the time goes without incident, giving rise to large periods of time with inactivity. By automatically clustering these sections together, an analyst would be able to quickly pinpoint areas of interest and mark them. The difficult part of this problem is properly extracting features from the dataset for classification. Some schemes using texture analysis for feature extraction are recommended in [34], [12], and [1].

### 6.4.5 Distributed Architecture

The current architecture allows an analyst to import and view data that can fit on a single machine. There are two downsides to this, the entire data corpus will not fit on a single machine, causing costly I/O to transfer files in for analysis or certain manual steps to modify the infrastructure, for example swapping out hard disks. In addition, there is no way to easily share information between other analysts for collaboration.

A more general solution would be to host Pinky in a distributed infrastructure giving analysts access to patient files across as many machines as necessary. The system is naturally partitionable by patient records, which could simplify the design which would only require data replication and access parts of an array over the network.

Given the changes proposed in Section 6.4.3 and Section 6.4.4 a distributed data service, similar to the DataHub [8] would be ideal. This service would allow hospitals and medical research institutions around the world to host patient data for collaborative analysis, using the Pinky computation and visualization layers to power analysis.

### 6.5 Conclusion

Pinky is a system for interactively analyzing EEG data at scale. We have evaluated different storage systems to address the need for optimized access to large array based datasets, and built an adaptive browser-based visualization system, Visgoth, on top of it. The processing layer of Pinky allows efficient mediation between the stored data and the data rendered for
an analyst. Using Pinky, doctors are now able to perform patient analysis more efficiently. Future work would allow doctors to collaborate, sharing their insights with doctors around the world.
Bibliography


[38] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets.